CS Exit Ethiopia

Questions & Worked Answers

Computer Science Exit Examination

141 complete questions and 4 source variants

1 selected course | 18 November 2026

Selected courses

01. Design and Analysis of Algorithms

https://csexitethiopia.com

abelasnake2003@gmail.com | Telegram: @Abechu7

Course contents

Each course begins on a new page. PDF bookmarks provide question, study-note section and revision-topic navigation.

Design and Analysis of Algorithms 3

Reading the answers

Every question includes its answer, explanation, choice-by-choice contrasts and available evidence. Primary note proof quotes the study notes. Related reading and module passages are labelled separately; a module reference is not presented as primary note proof.

Design and Analysis of Algorithms

Questions & Worked Answers

141 questions + 4 source variants

Question 1 | Q00065

Which one of the following is not an example of divide and conquer algorithm?

A. Quick sort

B. Binary searching

C. Merge sort

D. Heap sort

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: Heap sort

DAA16 explicitly classifies heap sort as outside standard divide and conquer: it builds a heap and repeatedly extracts its maximum. Quicksort partitions and recursively sorts ranges; merge sort recursively sorts halves; binary search recursively or iteratively reduces to a smaller half. Thus heap sort is the single exclusion among the listed algorithms.

Why each choice is right or wrong

A - Incorrect: Quicksort partitions its input and recursively sorts the resulting smaller ranges, a standard divide-and-conquer pattern.

B - Incorrect: Binary search repeatedly reduces a sorted search interval to one half; the note treats this as divide-and-conquer.

C - Incorrect: Merge sort divides into halves, sorts each recursively, and combines them by merging.

D - Correct: Heap sort builds a heap and repeatedly restores it after extraction, rather than independently sorting and combining subinstances.

Primary note proof

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results. DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it. Backtracking grows/retracts candidates when constraints fail; branch and bound also rejects candidates whose best possible objective cannot beat an incumbent.

Study-note section DAA16

Heap sort builds a max-heap in Θ(n), then extracts/restores in O(log n) per item: O(n log n) total, Θ(1) auxiliary space, ordinarily unstable. It is not standard D&C. Merge sort offers stability/worst-case guarantees; quicksort's practical advantage depends on implementation/workload.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 17 | lines 23-29

• Breaking large problems into smaller subproblems • Example of algorithms uses divide-and-conquer strategy : • Binary search, • Finding maximum and minimum • Merge sort • Quick sort • selection sort

Design and Analysis of Algorithms module | PDF page 23 | lines 19-29

QUICK SORT • The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. • In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged. • In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later. • This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n. • Thus the elements in a[1:m] & a[m+1:n] can be independently sorted. • No merge is needed. This rearranging is referred to as partitioning.

Design and Analysis of Algorithms module | PDF page 21 | lines 18-29

MERGE SORT • As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n) • This algorithm is called merge sort • We assume throughout that the elements are to be sorted in non-decreasing order. • Given a sequence of _n' elements a[1],...,a[n] the general idea is to imagine then split into 2 sets a[1],.....,a[n/2] and a[[n/2]+1],....a[n]. • Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of _n' elements. • Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Question sources

MOE Computer Science Exit Exam · 2016 · Question 65

Question 2 | Q00149

For n≥2, determine the tight asymptotic running time of this syntax-repaired C++-style pseudocode under mathematical-integer, unit-cost arithmetic and output:
function(n) { a=0; print("Enter a positive integer"); for(i=1; i<n; i*=2) a=i; return 0; }

A. Θ(n log n)

B. $\Theta(n^2)$

C. Θ(log n)

D. Θ(n)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: Θ(log n)

The loop visits i=1,2,4,... while i<n, so it executes ceil($\log_2 n$) iterations. Each iteration has constant work under the stated arithmetic and output model, and initialization, the fixed prompt and return add constant overhead. The total is therefore Θ(log n).

Why each choice is right or wrong

A - Incorrect: n log n grows faster and is not the tight order.

B - Incorrect: Quadratic grows faster and is not the tight order.

C - Correct: The doubling loop takes logarithmically many constant-cost iterations.

D - Incorrect: Linear is a looser upper bound rather than the tight order.

Primary note proof

Study-note section DAA06

> Sequential costs add; independent nested loop counts multiply. A dependent loop requires summation. n iterations of a constant body give Θ(n); ∑i=1.. n i=n(n+1)/2 gives Θ(n^2). Doubling i=1,2,4,... until i≥n takes ceil(log 2 n) iterations for n>1. A 10n loop is linear under constant-cost output.

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 10 | lines 6-20

> Analysis Rules: 1. We assume an arbitrary time unit. 2. Execution of one of the following operations takes time 1: • Assignment Operation • Single Input/output Operation • Single Boolean Operations • Single Arithmetic Operations • Function Return 3. Running time of a selection statement (if, switch) is the time for the condition evaluation plus the maximum of the running times for the individual clauses in the selection. 4. Loops: Running time for a loop is equal to the running time for the statements inside the loop multiplied by the number of iterations. The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops. For nested loops, analyze inside out.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching • Works on ordered/sorted list • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list • The computational time for this algorithm is proportional to log2 n.

Question sources

Addis Ababa University mock exam · Question 51

Question 3 | Q00173

Recursion is a method in which the solution of a problem depends on:

A. Larger instances of the same problem

B. Smaller instances of the same problem

C. Larger instances of different problems

D. Smaller instances of different problems

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: Smaller instances of the same problem

Recursion reduces a problem to simpler instances of that same problem (PDF p.14). Thus smaller same-problem instances, with a terminating base case, match the definition.

Why each choice is right or wrong

A - Incorrect: Larger instances do not implement the reducing step on p.14.

B - Correct: Smaller instances preserve the problem while reducing its size (p.14).

C - Incorrect: Changing both size upward and problem identity fails the recursive reduction (p.14).

D - Incorrect: Smaller instances of a different problem are not the same-problem recursion described on p.14.

Primary note proof

Study-note section DAA09

Include every recursive call, nonrecursive work and base case. Expansion gives $T(n)=T(n-1)+\Theta(1)\rightarrow\Theta(n)$, and $T(n)=T(n/2)+\Theta(1)\rightarrow\Theta(\log n)$. For $2T(n/2)+\Theta(n)$, logarithmically many levels each cost $\Theta(n)$, giving $\Theta(n \log n)$.

Study-note section DAA01

Prove an invariant initially, after each step and at termination; for recursion prove base cases and the inductive reduction. Testing finds failures but finite tests do not prove correctness for all inputs.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 30-33

> Recursion is a particularly powerful kind of reduction, which can be described loosely as follows: • If the given instance of the problem is small or simple enough, just solve it. • Otherwise, reduce the problem to one or more simpler instances of the same problem.

Question sources

Addis Ababa University mock exam · Question 75

Computer Science Exit Exam · Sene 2016 · Question 16-3

Computer Science Exit Exam practice collection · Question 96

Question 4 | Q00175

If an optimal solution can be created for a problem by constructing optimal solutions for its subproblems, the problem possesses property.

A. Overlapping subproblems

B. Optimal substructure

C. Memoization

D. Greedy

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: Optimal substructure

Optimal substructure means an optimal solution contains optimal solutions to subproblems (PDF p.28). The principle of optimality on p.49 expresses exactly the construction in the stem.

Why each choice is right or wrong

A - Incorrect: Overlap concerns repeated subproblems, not optimality of their solutions (pp.28,49).

B - Correct: This is the named property defined on p.28.

C - Incorrect: Memoization stores computed answers; it is not the optimality property (p.49).

D - Incorrect: Greedy is a design strategy requiring additional choice conditions (p.28).

Primary note proof

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be

missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Study-note section DAA19

A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 28 | lines 3-4

• Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems.

Design and Analysis of Algorithms module | PDF page 49 | lines 2-12

The Principle of Optimality The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal. The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its sub problems. The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which sub problems are relevant to the problem under consideration. General method-multistage graphs-all pair shortest path algorithm-Optimal Binary Search trees- 0/1 knapsack Problem-Reliability design-traveling salesman problem

Question sources

Addis Ababa University mock exam · Question 77

Question 5 | Q00176

If a problem can be broken into subproblems that are reused several times, the problem possesses property.

A. Overlapping subproblems

B. Optimal substructure

C. Memoization

D. Greedy

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Overlapping subproblems

Reusing a subproblem several times means subproblems share sub-subproblems, the contrast made on PDF p.48. This is overlapping subproblems; storing an answer is a response to that property, not the property itself.

Why each choice is right or wrong

A - Correct: Shared sub-subproblems and repeated work are described on p.48.

B - Incorrect: Optimal substructure concerns optimal solutions rather than repetition (p.28).

C - Incorrect: Saving results in a table addresses repeated work; it is not the reuse property (p.48).

D - Incorrect: A greedy choice is unrelated to whether subproblems recur (pp.28,48).

Primary note proof

Study-note section DAA25

> Overlapping states motivate reuse; optimal substructure supports optimization. These are textbook signatures, not requirements that every DP problem optimize or every state have multiple users. DP also counts/decides. Cost=states×work per state and can remain exponential.

Study-note section DAA26

> Memoization recursively checks a cache before solving a state and stores its result. Tabulation orders dependencies explicitly; chosen domains may exclude irrelevant states. Cache keys include all relevant parameters; recursion/locality/sparsity affect implementation cost.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 28 | lines 3-4

> • Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems.

Question sources

Addis Ababa University mock exam · Question 78

Question 6 | Q00177

If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called _____.

A. Dynamic programming

B. Divide and Conquer

C. Recursion

D. Greedy

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: Divide and Conquer

PDF p.48 explicitly contrasts independent subproblems in divide and conquer with shared sub-subproblems in dynamic programming. Combining independent solutions therefore identifies divide and conquer; optimality alone does not identify greedy.

Why each choice is right or wrong

A - Incorrect: Dynamic programming is motivated by non-independent shared subproblems (p.48).

B - Correct: Independent subproblems are recursively solved and combined (pp.17,48).

C - Incorrect: Recursion is a mechanism; the more specific independent-split-and-combine strategy is given on p.48.

D - Incorrect: Greedy needs a locally justified choice, which this stem does not specify (p.48 contrast).

Primary note proof

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results. DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it. Backtracking grows/retracts candidates when constraints fail; branch and bound also rejects candidates whose best possible objective cannot beat an incumbent.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 17 | lines 5-9

> 2.1 Divide and Conquer Algorithm In this approach, we solve a problem recursively by applying 3 steps 1. DIVIDE-break the problem into several sub problems of smaller size. 2. CONQUER-solve the problem recursively. 3. COMBINE-combine these solutions to create a solution to the original problem.

Question sources

Addis Ababa University mock exam · Question 79

Computer Science Exit Exam · Sene 2016 · Question 18-3

Computer Science Exit Exam practice collection · Question 117

Question 7 | Q00283

Suppose we want to determine the efficiency of the algorithm, then how we can measure the space factor

A. To count the maximum memory required by the algorithm

B. To count the average memory required by the algorithm

C. To count the minimum memory required by the algorithm

D. To count the maximum disk space needed by the algorithm

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: To count the maximum memory required by the algorithm

Space needed to run to completion must allow the program to finish without exhausting memory (PDF p.6). Hence the required allocation is peak simultaneously needed memory for that execution; average or minimum use may underallocate.

Why each choice is right or wrong

A - Correct: Maximum simultaneously required memory supports completion (p.6).

B - Incorrect: Average use can fall below a later peak and cannot guarantee enough memory (p.6).

C - Incorrect: Minimum use cannot supply the full execution requirement (p.6).

D - Incorrect: Disk capacity alone does not measure the program memory requirement defined on p.6.

Primary note proof

Study-note section DAA08

> Space complexity measures peak simultaneously live storage, not the sum of every allocation ever made. Count input, auxiliary structures and stack according to the requested convention. n active recursion levels with constant-size frames need Θ(n) stack; depth log n with constant frames needs Θ(log n). The number of total calls does not determine peak depth.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

> The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Question sources

Computer Science model exam · Question 14

Computer Science model exam 2 · Question 67

Computer Science model exam 2 · Question 89

University of Gondar model exam · Question 14

Question 8 | Q00284

Two main measures for the efficiency of an algorithm are

A. Processor and memory

B. Data and space

C. Complexity and capacity

D. Time and space

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: Time and space

Performance is stated in computer memory and time (PDF p.4), and complexity is expressed as space and time (p.6). The pair is therefore time and space.

Why each choice is right or wrong

A - Incorrect: Processor is a device, not the time measure paired with memory (p.4).

B - Incorrect: Data is the problem content, not one of the two resource measures (p.6).

C - Incorrect: Complexity and capacity are not the stated two measures (p.6).

D - Correct: Time and space are explicitly named on p.6.

Primary note proof

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

> 1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

> The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Question sources

Computer Science model exam · Question 15

University of Gondar model exam · Question 15

Question 9 | Q00285

For the standard Floyd-Warshall implementation on an n-vertex weighted graph, using constant-time distance arithmetic and the full k,i,j loops, what is the tight running-time bound?

A. $\Theta(n^2)$

B. $\Theta(n^3)$

C. $\Theta(n)$

D. $\Theta(n^4)$

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: $\Theta(n^3)$

Floyd-Warshall considers each intermediate vertex k and updates every ordered pair i,j. The three independent n-sized loop ranges perform $\Theta(n^3)$ constant-cost updates. The n^2 output table explains storage, not the time of this implementation. Other graph assumptions can support different algorithms, so this is the cost of the explicitly named method rather than a universal best-achievable APSP bound.

Why each choice is right or wrong

A - Incorrect: Quadratic counts pair storage but misses the intermediate-vertex loop.

B - Correct: Three full n-sized loops give cubic time.

C - Incorrect: Linear is too small even for the explicit pair table.

D - Incorrect: Quartic is a loose upper bound rather than the tight cost.

Primary note proof

Study-note section DAA28

> For all-pairs distances initialize D[i, i]=0, D[i, j] to the minimum direct edge weight, and missing edges to∞. Preserve a negative self-loop by taking min(0, its weight). For k outermost, update D[i, j]=min(D[i, j], D[i, k]+D[k, j]); before phase k, intermediates are restricted to earlier vertices. Guard unreachable sums and numeric overflow.

Study-note section DAA28

> Three loops give $\Theta(V^3)$ time and $\Theta(V^2)$ storage. A negative diagonal afterward identifies a negative cycle; pairs able to traverse it have no finite minimum walk distance. Warshall reachability replaces min/+ by OR/AND. Floyd's cost is not a universal best-achievable bound for all APSP settings; sparse nonnegative graphs permit repeated Dijkstra and other models permit other methods.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 61 | lines 14-23

> 4.2. ALL PAIR SHORTEST PATH • Let G=<V,E> be a directed graph 'V' is a set of nodes and _E' is the set of edges. • Each edge has an associated non-negative length. • We want to calculate the length of the shortest path between each pair of nodes. • Suppose the nodes of G are numbered from 1 to n, so V={1,2,...N},and suppose G matrix L gives the length of each edge, with L(i,j)=0 for i=1,2...n,L(i,j)>=for all i & j, and L(i,j)=infinity, if the edge (i,j) does not exist. • The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.

Design and Analysis of Algorithms module | PDF page 64 | lines 18-28

> Function AllPair (L[1..r,1..r]):array[1..n,1..n] array D[1..n,1..n] D = L For k = 1 to n do For i = 1 to n do For j = 1 to n do D [i , j] = min (D[i, j], D[i, k] + D[k, j] Return D ANALYSIS: This algorithm takes a time of O(n3)

Question sources

Computer Science model exam · Question 16

University of Gondar model exam · Question 16

Question 10 | Q00287

A recursive execution has n simultaneously active call levels, each retaining Theta(1) storage. Count only live call-frame storage. What is the tightest listed asymptotic upper bound on its peak space?

A. O(n'2)

B. O(n)

C. O(1)

D. O(log n)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: B. O(n)

All n call frames are live at the deepest point. Each frame retains a fixed positive amount of storage, so their combined peak space is proportional to n. This gives Theta(n) space and makes O(n) the tightest listed upper bound. $O(n^2)$ is a looser valid upper bound, while constant and logarithmic space cannot hold for n simultaneously live constant-size frames.

Why each choice is right or wrong

A - Incorrect: Quadratic is a valid but looser upper bound than the requested tightest one.

B - Correct: Linear space tightly bounds n live constant-size frames.

C - Incorrect: Constant space cannot hold as the number of live frames grows.

D - Incorrect: Logarithmic space cannot hold for n live constant-size frames.

Primary note proof

Study-note section DAA08

> Space complexity measures peak simultaneously live storage, not the sum of every allocation ever made. Count input, auxiliary structures and stack according to the requested convention. n active recursion levels with constant-size frames need Θ(n) stack; depth log n with constant frames needs Θ(log n). The number of total calls does not determine peak depth.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 6 | lines 28-31

> • Environmental stack space:- is the space used to save information needed to resume execution of partially completed methods (functions). For example, if method func1 invokes or calls method func2, then we must at least save a pointer to the instruction of func1 to be executed when func2 terminates. Each time a function or method is called or invoked, the following data are stored on the

Design and Analysis of Algorithms module | PDF page 8 | lines 2-3

> ii. The values of all local variables and formal parameters in the method or function being invoked or called. Time Complexity of a program: - is the amount of computer time it needs to run to completion. We are

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It’s only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Question sources

Computer Science model exam · Question 18

University of Gondar model exam · Question 18

Question 11 | Q00295

Which of the following is a Divide and Conquer algorithm?

A. Merge Sort

B. Heap Sort

C. Selection Sort

D. Bubble Sort

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Merge sort

DAA14 splits into halves, recursively sorts each, and merges the results, directly implementing divide and conquer. DAA16 excludes standard heap sort from that category and describes selection's repeated minimum and bubble's adjacent passes. Only Merge Sort among these options has the specified independent recursive decomposition and combination mechanism.

Why each choice is right or wrong

A - Correct: Merge sort recursively solves halves and combines them through a merge.

B - Incorrect: Standard heapsort builds and repeatedly restores a heap rather than applying the standard independent-subproblem decomposition.

C - Incorrect: Selection repeatedly finds a suffix minimum instead of recursively sorting halves.

D - Incorrect: Bubble repeatedly swaps adjacent inversions instead of recursively decomposing subinstances.

Primary note proof

Study-note section DAA14

> Split by indices, recursively sort each half, then repeatedly copy the smaller remaining front element into a temporary result. The merge invariant says the output prefix is sorted and contains the smallest elements already consumed. Append the unexhausted suffix. Equal keys must be taken from the left half first to preserve stability.

Study-note section DAA16

> Insertion shifts larger sorted-prefix keys: stable with strict comparisons, $\Theta(n)$ best and $\Theta(n^2)$ worst. Optimized bubble sort swaps adjacent inversions and stops after a no-swap pass: stable, $\Theta(n)$ best, $\Theta(n^2)$ worst. Selection repeatedly finds a suffix minimum: $\Theta(n^2)$ comparisons in every order, normally unstable. These use $\Theta(1)$ auxiliary array space.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 17 | lines 23-29

> • Breaking large problems into smaller subproblems • Example of algorithms uses divide-and-conquer strategy : • Binary search, • Finding maximum and minimum • Merge sort • Quick sort • selection sort

Design and Analysis of Algorithms module | PDF page 21 | lines 18-29

> MERGE SORT • As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n) • This algorithm is called

merge sort • We assume throughout that the elements are to be sorted in non-decreasing order. • Given a sequence of _n' elements a[1],...,a[n] the general idea is to imagine then split into 2 sets a[1],.....,a[n/2] and a[[n/2]+1],....a[n]. • Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of _n' elements. • Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items. • How many comparisons? => O(n2) and How many swaps? => O(n)

Question sources

Computer Science model exam · Question 26

University of Gondar model exam · Question 26

Question 12 | Q00373

In Average case, if operation takes ƒ(n) time in execution, then m operations will take?

A. ƒ(n)

B. f(m)

C. mf(n)

D. nf(m)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: mf(n)

Adding m execution costs each with mean f(n) gives mean total m*f(n) by the consecutive-block addition rule (PDF p.12), using the average-case meaning on p.14. This assumes n is fixed and each of the m operations has the stated mean; no independence is needed to add expected costs.

Why each choice is right or wrong

A - Incorrect: f(n) counts only one mean execution cost (pp.12,14).

B - Incorrect: f(m) changes the argument instead of adding m costs (p.12).

C - Correct: m equal mean costs sum to m*f(n) (p.12).

D - Incorrect: n*f(m) swaps operation count and input size without justification (p.12).

Primary note proof

Study-note section DAA06

If m operations each have expected cost f(n) at the same fixed size, expected total is m f(n), by linearity of expectation; independence is unnecessary. If sizes change, sum f(n_i). Exact coefficients depend on whether tests, increments, arithmetic and assignments count separately.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 12 | lines 6-20

for Loops: Formally • In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop. ∑1 = N i=1 • Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence N additions in total. Nested Loops: Formally • Nested for loops translate into multiple summations, one for each for loop. N M N ∑∑2 = ∑2M = 2MN i=1 j=1 i=1 • Again, count the number of additions. The outer summation is for the outer for loop. Consecutive Statements: Formally • Add the running times of the separate blocks of your code

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Question sources

Data Structures & Algorithms practice questions · Question 3

Question 13 | Q00376

O(1) means computing time is _____

A. Constant

B. Quadratic

C. Linear

D. Cubic

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Constant

O(1) is a bounded constant upper growth rate independent of input size. The related PDF p.3 visually orders constant K before log n, n, n log n and n^2; the assigned PDF p.14 explains comparison by growth rate.

Why each choice is right or wrong

A - Correct: A constant bound does not increase with n (cross-module p.3 growth diagram).

B - Incorrect: Quadratic corresponds to n squared, which grows with n (assigned p.14).

C - Incorrect: Linear is proportional to n, unlike 1 (assigned p.14).

D - Incorrect: Cubic is proportional to n cubed, unlike 1 (assigned p.14).

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \le c \cdot g(n)$ for all $n \ge n_0$, with fixed $c > 0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n * log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Data Structures & Algorithms practice questions · Question 6

Question 14 | Q00378

Which of the following sorting algorithm is of divide-and-conquer type?

A. Bubble sort

B. Insertion sort

C. Quick sort

D. Selection sort

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: Quicksort

DAA15 partitions around a pivot and recursively sorts smaller ranges, which is the divide-and-conquer strategy. The other ordinary methods in DAA16 use adjacent passes, sorted-prefix insertion, or repeated suffix-minimum selection. They do not use the same recursive partition decomposition. Thus Quick sort is the unique matching main choice.

Why each choice is right or wrong

A - Incorrect: Bubble uses repeated adjacent comparisons rather than recursive partitioning.

B - Incorrect: Insertion grows a sorted prefix by shifting larger preceding values.

C - Correct: Quicksort partitions and recursively sorts the resulting smaller ranges.

D - Incorrect: Selection repeatedly picks a suffix minimum rather than recursively partitioning and combining.

Primary note proof

Study-note section DAA15

> Choose a pivot, partition, then sort the resulting smaller ranges. In a two-pointer scheme, advance i while values are below pivot and j while above; when i≤j, swap and advance both. Bound the scans and ensure progress on equal values. Recurse on the resulting ranges. Hoare-style partition boundaries need not put the pivot at its final position; Lomuto-style partition commonly does. Do not mix their return conventions.

Study-note section DAA16

> Insertion shifts larger sorted-prefix keys: stable with strict comparisons, $\Theta(n)$ best and $\Theta(n^2)$ worst. Optimized bubble sort swaps adjacent inversions and stops after a no-swap pass: stable, $\Theta(n)$ best, $\Theta(n^2)$ worst. Selection repeatedly finds a suffix minimum: $\Theta(n^2)$ comparisons in every order, normally unstable. These use $\Theta(1)$ auxiliary array space.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 23 | lines 19-29

QUICK SORT • The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. • In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged. • In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later. • This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n. • Thus the elements in a[1:m] & a[m+1:n] can be independently sorted. • No merge is needed. This rearranging is referred to as partitioning.

Design and Analysis of Algorithms module | PDF page 21 | lines 18-29

MERGE SORT • As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n) • This algorithm is called merge sort • We assume throughout that the elements are to be sorted in non-decreasing order. • Given a sequence of _n' elements a[1],...,a[n] the general idea is to imagine then split into 2 sets a[1],.....,a[n/2] and a[[n/2]+1],....a[n]. • Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of _n' elements. • Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items. • How many comparisons? => O(n2) and How many swaps? => O(n)

Question sources

Data Structures & Algorithms practice questions · Question 8

Question 15 | Q00718

of an algorithm refers to defining the mathematical bound/framing of its run-time performance?

A. Space complexity

B. Asymptotic analysis

C. Time complexity

D. none of the above

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: Asymptotic analysis

Asymptotic analysis studies the limiting growth of running time with input size and groups functions by growth bounds (PDF p.14). Time complexity is the resource function being studied; the bounding analysis is asymptotic analysis.

Why each choice is right or wrong

A - Incorrect: Space concerns memory, not bounding the stated runtime (p.14).

B - Correct: The growth-rate and bounding analysis is explicitly titled Asymptotic Analysis (p.14).

C - Incorrect: Time complexity describes runtime; asymptotic analysis classifies its growth as asked (p.14).

D - Incorrect: The bounding concept is provided by the p.14 section, so “none” fails.

Primary note proof

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \le c \cdot g(n)$ for all $n \ge n_0$, with fixed $c > 0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Computer Science model exam 2 · Question 59

Question 16 | Q00719

Which described implementation uses dynamic programming through stored dependent subproblem values?

A. Multistage DAG shortest paths using stored reverse-topological cost-to-go values

B. Single-source shortest paths by Dijkstra on nonnegative edges

C. Merge sort

D. Chess game-tree search without cached subproblem values

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Multistage DAG shortest paths using stored reverse-topological cost-to-go values

Multistage DAG paths can be solved by storing the best cost-to-go for each vertex and evaluating vertices in reverse topological order. Each value combines outgoing-edge costs with already solved successor values, which is dynamic programming. The specified Dijkstra implementation uses greedy finalization, merge sort uses independent halves, and uncached chess tree search does not reuse stored subproblem values as described.

Why each choice is right or wrong

A - Correct: The reverse-topological cost recurrence reuses solved successor values.

B - Incorrect: Dijkstra uses greedy minimum-distance finalization under nonnegative weights.

C - Incorrect: Merge sort divides into independent halves and merges them.

D - Incorrect: The stipulated uncached search does not reuse solved state values.

Primary note proof

Study-note section DAA12

> DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it.

Study-note section DAA27

> For a DAG or multistage graph, compute cost-to-go in reverse topological order: F(t)=0 and F(u)=min over outgoing(u, v) of[w(u, v)+F(v)].

Study-note section DAA24

> Repeatedly extract the unsettled vertex u of minimum distance; relax each edge by dist[v]=min(dist[v], dist[u]+w(u, v)), updating predecessors when improved.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 49 | lines 2-12

The Principle of Optimality The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal. The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its sub problems. The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which sub problems are relevant to the problem under consideration. General method-multistage graphs-all pair shortest path algorithm-Optimal Binary Search trees- 0/1 knapsack Problem-Reliability design-traveling salesman problem

Design and Analysis of Algorithms module | PDF page 46 | lines 2-4

o Single Source Shortest Path Algorithm (Dijkstra's) Single-source shortest path: Graphs can be used to represent the highway structure of a state or country with

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Question sources

Computer Science model exam 2 · Question 60

Question 17 | Q00720

Optimal merge pattern is an example of _____ Algorithm design Approach

A. Dynamic Programming

B. Divide and Conquered

C. Greedy

D. Back tracking

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: Greedy

The optimal-merge construction explicitly chooses the two smallest files at each step (PDF p.43). That repeated local selection is called a greedy attempt on the page and gives the listed example an optimal merge pattern.

Why each choice is right or wrong

A - Incorrect: No table of overlapping subproblems drives the p.43 construction.

B - Incorrect: The pairing order is chosen by repeated smallest-file selection, not a predetermined independent split (p.43).

C - Correct: The module explicitly calls this selection rule greedy (p.43).

D - Incorrect: The described rule does not backtrack over earlier pairings (p.43).

Primary note proof

Study-note section DAA22

> Combine the two smallest weights, add their sum to total cost, and reinsert it. A min-heap gives O(n log n) time/O(n) space. Cost p+q counts moved records; worst-case comparisons are p+q−1 for nonempty lists, and actual comparisons may be fewer.

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 43 | lines 5-12

> A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n record file and an m record file requires possibly n + m records moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files F , ••• , F 5) with sizes (20, 30, 10, 5, 30) our greedy rule would generate the following merge pattern: merge F4 and FJ to get Z1 (|Z1| = 15); merge Z1 and F1 to get Z2 (|Z2| = 35); merge Fi and Fs to get ZJ (|ZJ| = 60); merge Z2 and ZJ to get the answer Z4. The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.

Question sources

Computer Science model exam 2 · Question 61

Question 18 | Q00898

For each input length n, a report records the smallest number of comparisons attained over all legal inputs of that length. Which case-analysis function is the report recording?

A. Average time

B. Small time

C. .

Worst time

D. Best time

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: D. Best time

At each fixed length n, the report selects the legal input with the smallest comparison count. This records best-case time under a comparison-count model. Worst case would record the largest count, and average case would take an expectation under a specified distribution. Comparing different input lengths is a separate growth analysis, so the application preserves the fixed-size meaning of Best time.

Why each choice is right or wrong

A - Incorrect: Average time is an expectation, not the minimum.

B - Incorrect: Small time is not the standard case-analysis term.

C - Incorrect: Worst time is the maximum over inputs of the fixed size.

D - Correct: Best time is the minimum over legal inputs of that fixed size.

Primary note proof

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the

smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Question sources

Computer Science Exit Exam · 2018 · Question 11

Question 19 | Q00925

Which statement about proving worst-case lower bounds for all comparison-based sorting algorithms is FALSE?

A. A comparison decision tree must distinguish n! possible orders of distinct keys.

B. Solving a recurrence for one chosen sorting algorithm, by itself, proves a lower bound for every algorithm in the comparison class.

C. Suppressing constant factors in asymptotic notation does not by itself invalidate a lower-bound proof.

D. A decision-tree argument can establish an Ω(n log n) worst-case comparison bound even when array indexing is allowed.

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: B. A single algorithm’s recurrence alone does not prove a class-wide lower bound.

A recurrence describes the running time of the particular algorithm it models. An upper or tight bound for that algorithm does not exclude a different comparison algorithm using fewer operations, so B is false. The class-wide decision-tree argument instead distinguishes n! orders of distinct keys and needs height at least $\log_2(n!)$, giving Ω(n log n). Ignoring constants and allowing ordinary array access do not defeat that comparison-count argument.

Why each choice is right or wrong

A - Incorrect: This is a true premise of the comparison decision-tree lower-bound argument, so it is not the requested false statement.

B - Correct: This is false: a recurrence for one algorithm does not lower-bound every other algorithm.

C - Incorrect: This is true: asymptotic lower bounds intentionally disregard constant factors.

D - Incorrect: This is true: the bound concerns comparisons, not a prohibition on array indexing.

Primary note proof

Study-note section DAA11

A comparison decision tree needs at least n! leaves for distinct-key orderings. Height h allows at most 2^h leaves, so h≥log 2(n!)=Ω(n log n); merge sort matches this worst-case bound. Array indexing is not forbidden.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 21 | lines 18-29

> MERGE SORT • As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n) • This algorithm is called merge sort • We assume throughout that the elements are to be sorted in non-decreasing order. • Given a sequence of _n' elements a[1],...,a[n] the general idea is to imagine then split into 2 sets a[1],.....,a[n/2] and a[[n/2]+1],....a[n]. • Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of _n' elements. • Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Design and Analysis of Algorithms module | PDF page 2 | lines 9-11

> - Describe Θ(f), Ω(f), O(f), growth rate, and algorithm order. - Use decision tree to determine a lower bound on complexity - Convert a simple recurrence relation into its closed form.

Question sources

Computer Science Exit Exam · 2018 · Question 38

Question 20 | Q00929

For integer n≥0, treat the code below as mathematical-integer RAM pseudocode: arithmetic is exact and unit-cost, with no overflow. Charge 1 unit for each initialization, comparison, loop increment and return; charge 1 for the addition and 1 for assignment in sum=sum+1. Ignore declarations, parameter passing and function-call overhead. What is T(n)?
total(n) { sum=0; for(i=1;i<=n;i++) sum=sum+1; return sum; }

A. $5n+5$

B. $4n^2+4$

C. $6n+2$

D. 4n+4

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: 4n+4

In this exact-integer model, the two initializations cost 2 units. The test i<=n runs n+1 times, the increment runs n times, and addition plus assignment costs 2n units. The return costs one unit. Thus T(n)=2+(n+1)+n+2n+1=4n+4, including n=0. Unbounded mathematical arithmetic keeps the final increment defined throughout the stated domain.

Why each choice is right or wrong

A - Incorrect: 5n+5 does not match the specified operation counts.

B - Incorrect: Quadratic growth does not match a single n-iteration constant-body loop.

C - Incorrect: 6n+2 charges operations differently from the stated model.

D - Correct: The complete count is 2+(n+1)+n+2n+1=4n+4.

Primary note proof

Study-note section DAA06

> If m operations each have expected cost f(n) at the same fixed size, expected total is m f(n), by linearity of expectation; independence is unnecessary. If sizes change, sum f(n_i). Exact coefficients depend on whether tests, increments, arithmetic and assignments count separately.

Study-note section DAA06

> For sum=0; for(i=1;i<=n;i++) sum=sum+1; return sum: counting two initializations, n+1 tests, n increments,2n body operations and one return gives 4n+4. A different primitive-cost convention changes this exact expression, not Θ(n).

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 10 | lines 6-20

> Analysis Rules: 1. We assume an arbitrary time unit. 2. Execution of one of the following operations takes time 1: • Assignment Operation • Single Input/output Operation • Single Boolean Operations • Single Arithmetic Operations • Function Return 3. Running time of a selection statement (if, switch) is the time for the condition evaluation plus the maximum of the running times for the individual clauses in the selection. 4. Loops: Running time for a loop is equal to the running time for the statements inside the loop multiplied by the number of iterations. The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops. For nested loops, analyze inside out.

Design and Analysis of Algorithms module | PDF page 10 | lines 34-41

1 for the assignment statement: int k=0 1 for the output statement. 1 for the input statement. In the for loop: 1 assignment, n+1 tests, and n increments. n loops of 2 units for an assignment, and an addition. 1 for the return statement. T (n)= 1+1+1+(1+n+1+n)+2n+1 = 4n+6 = O(n)

Question sources

Computer Science Exit Exam · 2018 · Question 42

Question 21 | Q01211

Which option most completely covers the listed factors that affect a program's exact elapsed running time, including its workload and execution environment?

A. Processor speed

B. Current processor load

C. Input size of the given algorithm

D. All

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: All listed factors affect observed runtime.

Input size influences how many operations the algorithm performs. Processor speed influences how quickly that work executes, and competing processor load influences the resources available during the run. All three can therefore affect exact elapsed time. Each individual factor is true but incomplete as an answer to the combined-factor question; D includes the whole set without confusing elapsed measurements with an abstract operation-count bound.

Why each choice is right or wrong

A - Incorrect: True but incomplete: processor speed matters, but it omits current load and input size.

B - Incorrect: True but incomplete: competing load matters, but it omits processor speed and input size.

C - Incorrect: True but incomplete: input size matters, but it omits the execution-environment factors.

D - Correct: All three listed factors contribute to exact elapsed time.

Primary note proof

Study-note section DAA02

Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs

need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Design and Analysis of Algorithms module | PDF page 6 | lines 18-30

Components of space complexity: - the space needed by a program has the following components: • Instruction space: - is the amount of space needed to store the compiled version of the program instructions. The amount of instruction space that is needed depends on factors such as: - The compiler used to compile the program into machine code. - The compiler options in effect at time of compilation. - The target computer. • Data space: - is the space needed to store all constant and variable values. Data space has two components: i. Space needed by constants (e.g. the numbers 0, 1, 2...) and simple variables. ii. Space needed by dynamically allocated objects such as arrays and class instances. • Environmental stack space:- is the space used to save information needed to resume execution of partially completed methods (functions). For

example, if method func1 invokes or calls method func2, then we must at least save a pointer to the instruction of func1 to be executed when func2

Question sources

Computer Science model exam 2 · Question 32

Question 22 | Q01230

Which of the following is an example of a Dynamic Programming algorithm?

A. Breadth First Search

B. Depth First Search

C. Dijkstra's Algorithm

D. Knapsack Problem

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: Knapsack Problem

The module develops 0/1 knapsack in the dynamic-programming unit (PDF p.71; application list p.49), with alternatives of including or excluding an item. Dijkstra is placed in the greedy unit (p.46). BFS and DFS are traversal techniques, rather than the shared-subproblem optimization example intended here (p.85).

Why each choice is right or wrong

A - Incorrect: Breadth-first search is a traversal; p.85 separates graph search from the knapsack optimization example.

B - Incorrect: Depth-first search is a traversal; the dynamic-programming item-selection example is p.71.

C - Incorrect: Dijkstra appears under the greedy unit on p.46, not as the intended dynamic-programming example.

D - Correct: The 0/1 knapsack variant is explicitly developed in the dynamic-programming unit on p.71.

Primary note proof

Study-note section DAA29

For integer capacity W, K[i, w] is maximum value using the first i items within capacity w. Initialize K[0, w]=0. If weight_i>w, use K[i−1, w]; otherwise take max(K[i−1, w], value_i+K[i−1, w−weight_i]). The previous-row reference ensures each item is used at most once. Backtrack changed values to recover chosen items.

Study-note section DAA34

BFS/DFS being printed under a DP unit does not make ordinary traversal a DP optimization algorithm.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 71 | lines 3-9

4.4. 0/1 KNAPSACK PROBLEM • This problem is similar to ordinary knapsack problem but we may not take a fraction of an object. • We are given _ N _ objects with weight Wi and profits Pi where i varies from l to N and also a knapsack with capacity _ M _. • The problem is, we have to fill the bag with the help of _ N _ objects and the resulting profit has to be maximum.

Design and Analysis of Algorithms module | PDF page 46 | lines 2-4

o Single Source Shortest Path Algorithm (Dijkstra’s) Single-source shortest path: Graphs can be used to represent the highway structure of a state or country with

Design and Analysis of Algorithms module | PDF page 85 | lines 14-20

4.7. BASIC SEARCH AND TRAVERSAL TECHNIQUE DEFINING GRAPH: A graphs g consists of a set V of vertices (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges. Therefore, V(G) is read as V of G, is a set of vertices and E(G),read as E of G is a set of edges. An edge e= (v, w) is a pair of vertices v and w, and to be incident with v and w.

Question sources

Computer Science model exam 2 · Question 51

Computer Science model exam 2 · Question 58

Question 23 | Q01232

When designing an algorithm for a specified implementation environment, which option most completely covers the listed resource, correctness and alternative-method considerations?

A. Whether the method respects the chosen hardware/resource cost model.

B. Whether the software implementation correctly realizes the specified method.

C. If there is more than one way to solve the problem.

D. All of the above

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: All of the above

Algorithm design starts with inputs, outputs, constraints and correctness, then evaluates resource use under a stated model. For an implementation, respecting the selected hardware/resource assumptions and ensuring the program implements the method correctly both matter. Comparing alternative solution methods is also legitimate. Thus the all-of-the-above option combines all three considerations; none of the individual choices alone covers the entire set.

Why each choice is right or wrong

A - Incorrect: Respecting the resource model matters but is only one consideration.

B - Incorrect: Implementation correctness matters but is only one consideration.

C - Incorrect: Comparing alternatives matters but is only one consideration.

D - Correct: All three stated considerations belong in the implementation-aware design.

Primary note proof

Study-note section DAA01

> State inputs, output, constraints and empty-solution policy. Algorithms are finite, definite executable methods independent of source language; feasible satisfies constraints, optimal is best feasible. Conditional branches are legitimate control flow.

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 6-19

• Finiteness: Algorithm must complete after a finite number of steps. • Definiteness: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next. • Sequence: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted. • Feasibility: It must be possible to perform each instruction. • Correctness: It must compute correct answer for all possible legal inputs. • Language Independence: It must not depend on any one programming language. • Completeness: It must solve the problem completely. • Effectiveness: It must be possible to perform each step exactly and in a finite amount of time. • Efficiency: It must solve with the least amount of computational resources such as time and space. • Generality: Algorithm should be valid on all possible inputs • Input/Output: There must be a specified number of input values, and one or more result values.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Design and Analysis of Algorithms module | PDF page 6 | lines 18-30

Components of space complexity: - the space needed by a program has the following components: • Instruction space: - is the amount of space needed to store the compiled version of the program instructions. The amount of instruction space that is needed depends on factors such as: - The compiler used to compile the program into machine code. - The compiler options in effect at time of compilation. - The target computer. • Data space: - is the space needed to store all constant and variable values. Data space has two components: i. Space needed by constants (e.g. the numbers 0, 1, 2...) and simple variables. ii. Space needed by dynamically allocated objects such as arrays and class instances. • Environmental stack space:- is the space used to save information needed to resume execution of partially completed methods (functions). For example, if method func1 invokes or calls method func2, then we must at least save a pointer to the instruction of func1 to be executed when func2

Question sources

Computer Science model exam 2 · Question 53

Question 24 | SM0007

Assume you have a weighted graph G with a set of vertices V and set of edges E. Which one of the following is an optimization problem on the graph?

A. Traverse all vertices in graph G

B. Is vertex p in V reachable from vertex q in V in graph G

C. Find minimum spanning tree from graph G

D. How many cycles are there in graph G

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: Find minimum spanning tree from graph G

An optimization problem seeks a feasible solution minimizing or maximizing an objective (PDF p.27). The minimum spanning tree minimizes total edge weight while connecting every vertex (p.36); the other tasks ask for traversal, a yes/no fact or a count.

Why each choice is right or wrong

A - Incorrect: Traversal asks to visit vertices rather than optimize a weight (pp.27,36).

B - Incorrect: Reachability is a decision question rather than minimization (pp.27,85).

C - Correct: Minimum total tree weight is the optimization objective (p.36).

D - Incorrect: Counting cycles asks for a number, with no minimization or maximization criterion (p.27).

Primary note proof

Study-note section DAA01

> State inputs, output, constraints and empty-solution policy. Algorithms are finite, definite executable methods independent of source language; feasible satisfies constraints, optimal is best feasible. Conditional branches are legitimate control flow.

Study-note section DAA23

> For a connected weighted undirected graph, an MST spans every vertex without cycles and minimizes total edge weight; it has V−1 edges. Negative edges are allowed. Equal weights may yield multiple MSTs; distinct weights suffice for uniqueness. A tree is itself a graph.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 27 | lines 7-18

> • Greedy method is the most straightforward designed technique. • As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decisions may have in the future. DEFINITION:

• A problem with N inputs will have some constraints. Any subsets that satisfy these constraints are called a feasible solution. • A feasible solution that either maximize can minimize a given objectives function is called an optimal solution. Greedy is a strategy that works well on optimization problems with the following characteristics: • Greedy-choice property: A global optimum can be arrived at by selecting a local

Design and Analysis of Algorithms module | PDF page 36 | lines 16-33

o Minimum Cost Spanning Tree- Prim's Algorithm ✓ Let G(V,E) be an undirected connected graph with vertices v and edge E . ✓ A sub-graph t=(V,E) of the G is a Spanning tree of G iff t is a tree.3 ✓ The problem is to generate a graph G = (V,E) where E is the subset of E,G is a Minimum spanning tree. ✓ Each and every edge will contain the given non-negative length. connect all the nodes with edge present in set E and weight has to be minimum. NOTE: ✓ We have to visit all the nodes. ✓ The subset tree (i.e) any connected graph with N vertices must have at least N-1 edges and also it does not form a cycle. Definition: 1. A spanning tree of a graph is an undirected tree consisting of only those edge that are necessary to connect all the vertices in the original graph. 2. A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Design and Analysis of Algorithms module | PDF page 85 | lines 14-20

4.7. BASIC SEARCH AND TRAVERSAL TECHNIQUE DEFINING GRAPH: A graphs g consists of a set V of vertices (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges. Therefore, V(G) is read as V of G, is a set of vertices and E(G),read as E of G is a set of edges. An edge e= (v, w) is a pair of vertices v and w, and to be incident with v and w.

Question sources

MOE Computer Science Exit Exam · Question 2-3

National Computer Science Exit Exam · Question 90

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 107

Question 25 | SM0031

For an average-cost function f(n)=E[T(n)] under a specified input distribution, which notation expresses a tight constant-factor growth bound relative to g(n)?

A. $f(n) \in \Omega(g(n))$

B. $f(n) \in O(g(n))$

C. $f(n) \in \omega(g(n))$

D. $f(n) \in \Theta(g(n))$

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: $f(n) \in \Theta(g(n))$

Average case first takes an expectation under the specified distribution. A tight growth relation for that resulting function requires both an eventual upper and lower constant-factor bound relative to g, which is Θ. Big-O supplies only an upper bound, Ω only a lower bound, and little-ω a strict asymptotic lower relation. None of the symbols intrinsically means average case.

Why each choice is right or wrong

A - Incorrect: Ω supplies a lower bound without the matching upper bound.

B - Incorrect: O supplies an upper bound without the matching lower bound.

C - Incorrect: Little-ω means f/g tends to infinity rather than constant-factor equivalence.

D - Correct: Θ states both constant-factor upper and lower bounds.

Primary note proof

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \le c \cdot g(n)$ for all $n \ge n_0$, with fixed $c > 0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It’s only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An

upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Data Structure and Algorithms module | PDF page 3 | lines 6-12

II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n). III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth. IV. Little-o Notation (o): f(n)=o(g(n)) means f(n) has less growth rate compared to g(n). V. Little-Omega Notation (ω): ω notation used to denote a lower bound that is not asymptotically tight.

Question sources

MOE Computer Science Exit Exam · Question 9-3

National Computer Science Exit Exam · Question 11

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 110

Question 26 | SM0065

Size of large data files can be reduced to save storage disk space. Which algorithm is appropriate to reduce the size of large files?

A. Heap sort algorithm

B. Prim's algorithm

C. Merge sort algorithm

D. Huffman encoding algorithm

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: D. Huffman encoding algorithm

Huffman coding assigns prefix-free bit strings to symbols using their frequencies, minimizing the weighted code length in its model. It can therefore reduce the storage required for data represented with those codes. Heap sort and merge sort instead reorder items, while Prim builds a minimum spanning tree of a weighted graph. Those other objectives do not encode file contents more compactly. Metadata overhead still prevents a universal compression guarantee for every file.

Why each choice is right or wrong

A - Incorrect: Heap sort reorders items by repeatedly extracting a heap extreme; it does not encode file symbols.

B - Incorrect: Prim minimizes graph-connection weight, not encoded file length.

C - Incorrect: Merge sort orders records by recursively sorting and merging, rather than encoding symbols.

D - Correct: Huffman builds a frequency-based prefix code minimizing its weighted length.

Primary note proof

Study-note section DAA22

> Huffman uses frequencies as weights and 0/1 branches to leaf symbols, making a prefix-free code. Cost is∑frequency×depth.

Study-note section DAA16

> Heap sort builds a max-heap in Θ(n), then extracts/restores in O(log n) per item: O(n log n) total, Θ(1) auxiliary space, ordinarily unstable.

Study-note section DAA23

> For a connected weighted undirected graph, an MST spans every vertex without cycles and minimizes total edge weight; it has V−1 edges.

Study-note section DAA14

> Split by indices, recursively sort each half, then repeatedly copy the smaller remaining front element into a temporary result.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 45 | lines 21-29

> The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If qi is the relative frequency with which message Mi will be transmitted, then the expected decode time is Σ ≤i≤n+ qidi. where di is the distance of the external node for message Mi from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that Σ ≤i≤n+ qidi is also the expected length of a transmitted message. Hence the code which minimizes expected decode time also minimizes the expected length of a message.

Design and Analysis of Algorithms module | PDF page 37 | lines 4-11

> Minimum cost spanning tree: o The cost of a spanning tree is the sum of cost of the edges in that trees. o There are 2 method to determine a minimum cost spanning tree are • Kruskal s Algorithm • Prom s Algorithm. • PRIM'S ALGORITHM Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in

Design and Analysis of Algorithms module | PDF page 21 | lines 18-29

> MERGE SORT • As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n) • This algorithm is called

merge sort • We assume throughout that the elements are to be sorted in non-decreasing order. • Given a sequence of _n‘ elements a[1],…,a[n] the general idea is to imagine then split into 2 sets a[1],…..,a[n/2] and a[[n/2]+1],….a[n]. • Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of _n‘ elements. • Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Question sources

MOE Computer Science Exit Exam · Question 19-1

National Computer Science Exit Exam · Question 2

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 111

Question 27 | SM0083

For the worst-cost function f(n)=max over size-n inputs T(input), which notation expresses a tight constant-factor bound relative to g(n)?

A. $f(n) \in \omega(g(n))$

B. $f(n) \in \Theta(g(n))$

C. $f(n) \in o(g(n))$

D. $f(n) \in \Omega(g(n))$

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: $f(n) \in \Theta(g(n))$

Worst case maximizes cost over all inputs of a fixed size. Once that function is defined, Θ expresses matching constant-factor upper and lower bounds relative to g. Little-o and little-ω describe strict asymptotic relations, and Ω alone supplies only a lower bound. The answer concerns tightness of the bound, not a claim that any notation inherently selects the worst input case.

Why each choice is right or wrong

A - Incorrect: Little-ω gives a strict lower-growth relation.

B - Correct: Θ gives matching constant-factor upper and lower bounds.

C - Incorrect: Little-o gives a strict upper-growth relation.

D - Incorrect: Ω alone lacks the required upper bound.

Primary note proof

Study-note section DAA03

At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Study-note section DAA04

For eventually nonnegative f, g, f∈O(g) means f(n)≤c·g(n) for all n≥n 0, with fixed c>0. Ω reverses that inequality; Θ requires both bounds. Little-o means f/g→0, and little-ω means f/g→∞ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

I. Big-Oh Notation (O): It’s only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Data Structure and Algorithms module | PDF page 3 | lines 6-12

II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n). III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth. IV. Little-o Notation (o): f(n)=o(g(n)) means f(n) has less growth rate compared to g(n). V. Little-Omega Notation (ω): ω notation used to denote a lower bound that is not asymptotically tight.

Question sources

MOE Computer Science Exit Exam · Question 23-4

National Computer Science Exit Exam · Question 71

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 112

Question 28 | SM0087

Which one of the following algorithms can be used to extract a Minimum Spanning Tree from an input graph?

A. Prim's algorithm

B. Huffman encoding algorithm

C. Dijkstra's algorithm

D. Merge sort algorithm

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Prim's algorithm

The module explicitly names Prim among the two minimum-cost spanning-tree methods (PDF p.37). Huffman codes optimize expected message length (p.45), Dijkstra is single-source path search (p.46), and merge sort orders arrays (p.21).

Why each choice is right or wrong

A - Correct: Prim is explicitly a minimum-cost spanning-tree method (p.37).

B - Incorrect: Huffman builds codes, not a spanning tree of the input graph (p.45).

C - Incorrect: Dijkstra targets source-to-vertex paths, a different objective (p.46).

D - Incorrect: Merge sort combines sorted sequences, not graph edges (p.21).

Primary note proof

Study-note section DAA23

> For a connected weighted undirected graph, an MST spans every vertex without cycles and minimizes total edge weight; it has V−1 edges. Negative edges are allowed. Equal weights may yield multiple MSTs; distinct weights suffice for uniqueness. A tree is itself a graph.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 37 | lines 4-11

> Minimum cost spanning tree: o The cost of a spanning tree is the sum of cost of the edges in that trees. o There are 2 method to determine a minimum cost spanning tree are • Kruskal s Algorithm • Prom s Algorithm. • PRIM'S ALGORITHM Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in

Design and Analysis of Algorithms module | PDF page 45 | lines 21-29

The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If qi is the relative frequency with which message Mi will be transmitted, then the expected decode time is Σ ≤i≤n+ qidi. where di is the distance of the external node for message Mi from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that Σ ≤i≤n+ qidi is also the expected length of a transmitted message. Hence the code which minimizes expected decode time also minimizes the expected length of a message.

Design and Analysis of Algorithms module | PDF page 46 | lines 2-4

o Single Source Shortest Path Algorithm (Dijkstra's) Single-source shortest path: Graphs can be used to represent the highway structure of a state or country with

Design and Analysis of Algorithms module | PDF page 21 | lines 18-29

MERGE SORT • As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n) • This algorithm is called merge sort • We assume throughout that the elements are to be sorted in non-decreasing order. • Given a sequence of _n' elements a[1],...,a[n] the general idea is to imagine then split into 2 sets a[1],.....,a[n/2] and a[[n/2]+1],....a[n]. • Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of _n' elements. • Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Question sources

MOE Computer Science Exit Exam · Question 24-4

National Computer Science Exit Exam · Question 19

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 113

Question 29 | SM0214

Which analysis case calculates the upper bound on running time of an algorithm?

A. Optimal case

B. Best case

C. Average case

D. Worst case

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: Worst case

Worst-case time is the maximum time over the inputs under consideration and therefore bounds all their runtimes (PDF p.14). This is the case-analysis upper bound, distinct from the separate question of an asymptotic bound on that maximum.

Why each choice is right or wrong

A - Incorrect: Optimal case is not a defined case function on p.14.

B - Incorrect: Best case minimizes rather than bounds all runtime values from above (p.14).

C - Incorrect: An average need not exceed every observation (p.14).

D - Correct: Worst case supplies the all-input bound explicitly described on p.14.

Primary note proof

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 5-1

Computer Science Exit Exam practice collection · Question 118

Question 30 | SM0249

An algorithm has unit-cost running-time function $T(n)=3n^k+7n+12$, where k is a fixed integer with $k\geq 2$ independent of n. Which listed growth class describes it?

A. Linear

B. Polynomial

C. Constant

D. Exponential

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: Polynomial

A fixed exponent $k \geq 2$ makes the leading term $3n^k$ dominate $7n+12$. The resulting function has $\Theta(n^k)$ growth, which is polynomial. It is not linear because k exceeds one, and the positive growing terms exclude constant growth. Exponential growth would instead put the input in an exponent of a fixed base, a different form from the fixed-degree polynomial here.

Why each choice is right or wrong

A - Incorrect: $k \geq 2$ prevents linear growth.

B - Correct: A fixed degree k gives polynomial growth.

C - Incorrect: The positive n-dependent terms prevent constant growth.

D - Incorrect: A fixed-degree power is not exponential growth.

Primary note proof

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n * log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 16-2

Question 31 | SM0302

The amount of memory needed to run an algorithm to completion is known as:

A. Space complexity

B. Time complexity

C. Worst case

D. Best case

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Space complexity

Space complexity is explicitly defined as the memory needed to run to completion (PDF p.6). This is a resource quantity, not an input-case category.

Why each choice is right or wrong

A - Correct: This is the exact definition on p.6.

B - Incorrect: Time measures duration rather than memory (p.6 contrast).

C - Incorrect: Worst case selects inputs; it is not the memory-resource name (p.6).

D - Incorrect: Best case selects inputs; it is not the memory-resource name (p.6).

Primary note proof

Study-note section DAA08

> Space complexity measures peak simultaneously live storage, not the sum of every allocation ever made. Count input, auxiliary structures and stack according to the requested convention. n active recursion levels with constant-size frames need Θ(n) stack; depth log n with constant frames needs Θ(log n). The number of total calls does not determine peak depth.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

> The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 33-2

Computer Science Exit Exam practice collection · Question 120

Question 32 | SS116

Interpret the following as mathematical-integer pseudocode with constant-cost arithmetic and print operations. Which offered Big-O upper bounds on the full running time are valid? Select all that apply; the bounds need not be tight.
for i=0,...,n−1:
for j=i,...,i^2−1:
if j mod i == 0:
for k=0,...,j−1: print("*")
An upper limit below the lower limit makes an empty loop.

A. $O(n^5)$

B. $O(n^7)$

C. O(n log n)

D. $O(n^2)$

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A, B: $O(n^5)$; $O(n^7)$

For i=0 and i=1 the j loop is empty, so no division by zero occurs. For i≥2, successful j values are i,2i,...,(i−1)i. Their k-loop work totals $i^2(i-1)/2=\Theta(i^3)$, and summing over i yields $\Theta(n^4)$. Consequently both $O(n^5)$ and $O(n^7)$ are valid loose upper bounds, while the two smaller offered rates cannot bound the work.

Why each choice is right or wrong

A - Correct: A quartic function is also $O(n^5)$.

B - Correct: A quartic function is also $O(n^7)$.

C - Incorrect: n log n is too small to upper-bound quartic work.

D - Incorrect: n^2 is too small to upper-bound quartic work.

Primary note proof

Study-note section DAA07

> For the loop i=0.. n−1, j=i.. i^2−1, and k=0.. j−1 only when j%i==0, i=0 and 1 have empty j ranges, so no remainder by zero is evaluated. For i≥2, successful j values are i,2i,...,(i−1)i. The inner work is $i\sum m=1..\ i-1\ m=i^2(i-1)/2=\Theta(i^3)$. Summing over i gives $\Theta(n^4)$; condition checks add only $\Theta(n^3)$.

Study-note section DAA07

> Neither n^5 nor n^7 is tight for this loop, though both are valid Big-O upper bounds. This assumes mathematical integers and constant-cost output; literal signed C++ overflow in i*i is undefined,

and executable runtime analysis requires valid syntax in a specified language. Redeclaring parameter n and using return @;3 are syntax errors; neither establishes logarithmic complexity.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 12 | lines 6-20

for Loops: Formally • In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop. ∑1 = N i=1 • Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence N additions in total. Nested Loops: Formally • Nested for loops translate into multiple summations, one for each for loop. N M N ∑∑2 = ∑2M = 2MN i=1 j=1 i=1 • Again, count the number of additions. The outer summation is for the outer for loop. Consecutive Statements: Formally • Add the running times of the separate blocks of your code

Design and Analysis of Algorithms module | PDF page 10 | lines 6-20

Analysis Rules: 1. We assume an arbitrary time unit. 2. Execution of one of the following operations takes time 1: • Assignment Operation • Single Input/output Operation • Single Boolean Operations • Single Arithmetic Operations • Function Return 3. Running time of a selection statement (if, switch) is the time for the condition evaluation plus the maximum of the running times for the individual clauses in the selection. 4. Loops: Running time for a loop is equal to the running time for the statements inside the loop multiplied by the number of iterations. The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops. For nested loops, analyze inside out.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Question sources

Computer Science model exam collection · Question 23

Question 33 | SS120

Which claims about abstract algorithms are NOT true? Select all that apply. In C, platform considerations mean choosing realistic resource constraints, not making source language part of the definition.

A. Is step wise logical description of how to solve the problem

B. ls equivalent to the programming language code

C. Should be developed by considering the platforms on which it runs

D. Its abstract definition requires the details of a particular programming language.

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B, D: Is equivalent to the programming language code; Its abstract definition requires the details of a particular programming language.

An algorithm is a definite executable method independent of a particular source language. Equating it with programming-language code, or making that language's details essential to its abstract definition, is therefore false. A stepwise logical description is appropriate. Resource constraints and a platform-aware cost model can legitimately inform implementation choices without changing the underlying method's language independence. Thus B and D are the false claims.

Why each choice is right or wrong

A - Incorrect: A stepwise logical description is compatible with an algorithm.

B - Correct: The abstract method is not identical to a source-language implementation.

C - Incorrect: Realistic resource constraints may inform design, so C is not selected.

D - Correct: Programming-language details are not essential to the abstract method.

Primary note proof

Study-note section DAA01

> State inputs, output, constraints and empty-solution policy. Algorithms are finite, definite executable methods independent of source language; feasible satisfies constraints, optimal is best feasible. Conditional branches are legitimate control flow.

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 6-19

> • Finiteness: Algorithm must complete after a finite number of steps. • Definiteness: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next. • Sequence: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted. • Feasibility: It must be possible to perform each instruction. • Correctness: It must compute correct answer for all possible legal inputs. • Language Independence: It must not depend on any one programming language. • Completeness: It must solve the problem completely. • Effectiveness: It must be possible to perform each step exactly and in a finite

amount of time. • Efficiency: It must solve with the least amount of computational resources such as time and space. • Generality: Algorithm should be valid on all possible inputs • Input/Output: There must be a specified number of input values, and one or more result values.

Question sources

Computer Science model exam collection · Question 27

Question 34 | SS132

For integer n>=0, determine the tightest listed growth bound for this mathematical-integer pseudocode. Arithmetic, loop-control operations and each print operation have unit cost; ignore output-digit length.

```
function someFunction(n) {
for (i=0; i<10*n; i=i+1) {
print(n);
}
}
```

A. O(n)

B. $O(n^2)$

C. O(log n)

D. O(log(2n))

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: A. O(n)

The loop body runs 10n times. Each iteration performs a constant amount of loop-control work and one unit-cost print, while initialization and the final failed condition check contribute only constant additional work. The total therefore has linear growth. Quadratic is a looser upper bound, while logarithmic growth is too small. The explicit mathematical and output-cost model avoids syntax and representation-cost distractions.

Why each choice is right or wrong

A - Correct: A constant multiple of n iterations gives the tight linear bound.

B - Incorrect: Quadratic is a looser bound rather than the requested tightest one.

C - Incorrect: Logarithmic growth is too small for 10n body executions.

D - Incorrect: log(2n) differs from log n only by a constant and is also too small.

Primary note proof

Study-note section DAA06

Sequential costs add; independent nested loop counts multiply. A dependent loop requires summation. n iterations of a constant body give $\Theta(n)$; $\sum i=1..\ n\ i=n(n+1)/2$ gives $\Theta(n^2)$. Doubling i=1,2,4,... until i≥n takes ceil(log 2 n) iterations for n>1. A 10n loop is linear under constant-cost output.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 10 | lines 6-20

Analysis Rules: 1. We assume an arbitrary time unit. 2. Execution of one of the following operations takes time 1: • Assignment Operation • Single Input/output Operation • Single Boolean Operations • Single Arithmetic Operations • Function Return 3. Running time of a selection statement (if, switch) is the time for the condition evaluation plus the maximum of the running times for the individual clauses in the selection. 4. Loops: Running time for a loop is equal to the running time for the statements inside the loop multiplied by the number of iterations. The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops. For nested loops, analyze inside out.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Question sources

Computer Science model exam collection · Question 39

Question 35 | SS141

Which of the following analysis mechanism defines the input for which the algorithm takes the least time (fastest time to complete)?

A. Standard Case

B. Average case

C. Worst case

D. Best case

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: Best case

The least runtime among inputs of a fixed size is best-case time, the fastest case in the p.14 three-case classification. It is not the average or worst case.

Why each choice is right or wrong

A - Incorrect: Standard case is not a defined category on p.14.

B - Incorrect: Average case concerns a distribution or average set, not a minimum (p.14).

C - Incorrect: Worst case takes the slowest input (p.14).

D - Correct: Best case is the minimum-time case intended by p.14.

Primary note proof

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Question sources

Computer Science model exam collection · Question 48

Question 36 | AM0344

Which described activity is outside analysis of an algorithm’s resource costs? Interpret dynamic analysis as measuring executed operations or storage.

A. Asymptotic analysis

B. Experimental analysis

C. Dynamic analysis

D. Graph connectivity analysis of the input data, without examining algorithm resource costs

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: Graph connectivity analysis of the input data, without examining algorithm resource costs

Asymptotic analysis studies resource growth, experimental analysis measures runs, and dynamic observation can measure executed operations or storage. The graph alternative asks only a mathematical connectivity question about input data, expressly without examining algorithm costs. It is outside the resource-analysis task as described, although graph-based representations can certainly support valid resource analysis in other contexts. This scope avoids an unsupported universal taxonomy.

Why each choice is right or wrong

A - Incorrect: Asymptotic analysis studies resource growth.

B - Incorrect: Experimental analysis measures resource costs of actual runs.

C - Incorrect: The stipulated dynamic analysis observes executed costs.

D - Correct: The stated connectivity-only task does not examine algorithm resource costs.

Primary note proof

Study-note section DAA02

> Experimental/dynamic measurement observes actual runs; asymptotic analysis abstracts growth; probabilistic analysis studies expectations. There is no universal taxonomy proving “graph analysis” or “dynamic analysis” is invalid. Auxiliary space excludes input; I/O complexity counts transfers. Representation affects costs; “data type complexity” is not a standard metric name.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

> 1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some

function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n * log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Design and Analysis of Algorithms module | PDF page 85 | lines 14-20

4.7. BASIC SEARCH AND TRAVERSAL TECHNIQUE DEFINING GRAPH: A graphs g consists of a set V of vertices (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges. Therefore, V(G) is read as V of G, is a set of vertices and E(G),read as E of G is a set of edges. An edge e= (v, w) is a pair of vertices v and w, and to be incident with v and w.

Question sources

Computer Science MCQ practice collection · Question 344

Question 37 | AM0345

What is the primary growth-related purpose of asymptotic analysis?

A. To measure the actual running time of an algorithm

B. To determine the best-case scenario for an algorithm

C. To analyze the behavior of an algorithm as the input size increases

D. To evaluate the correctness of an algorithm

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: To analyze the behavior of an algorithm as the input size increases

Asymptotic analysis abstracts how resource use grows with input size. It can analyze best-, average- or worst-cost functions, but does not itself choose one input case or measure actual elapsed seconds. Those measurements depend on environment and workload. Correctness is a separate obligation to meet the specification. Hence C states the primary growth-related purpose among these choices.

Why each choice is right or wrong

A - Incorrect: Actual elapsed-time measurement is experimental.

B - Incorrect: Asymptotic analysis is not restricted to the best case.

C - Correct: It analyzes resource behavior as input size grows.

D - Incorrect: Growth bounds do not prove correctness.

Primary note proof

Study-note section DAA02

Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Study-note section DAA03

At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Variant 1 | Alternative version 1

Which are valid uses of asymptotic resource analysis? Select all that apply. In B, performance comparison means comparing growth under a common cost model, rather than predicting exact elapsed time.

A. To determine the actual running time of an algorithm

B. To compare the performance of different algorithms

C. To evaluate the correctness of an algorithm

D. To analyze the behavior of an algorithm as the input size increases

Answer B, D: To compare the performance of different algorithms; To analyze the behavior of an algorithm as the input size increases

Asymptotic analysis studies how cost grows with input size and can compare that scaling across algorithms under a common model. Thus both the growth description and the performance-comparison application are valid. It does not give actual elapsed seconds, and a favorable resource bound does not establish correctness. The explicit growth-based meaning of performance makes the two valid variant choices definite.

Why each choice is right or wrong

A - Incorrect: Actual elapsed time needs execution-environment measurements.

B - Correct: Comparing cost growth under a common model is a valid use.

C - Incorrect: Resource bounds do not establish correctness.

D - Correct: Behavior as input size grows is the direct purpose.

Primary note proof

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Question sources

Computer Science MCQ practice collection · Question 345

Computer Science MCQ practice collection · Question 356

Question 38 | AM0346

Which notation expresses a general asymptotic upper bound on an algorithm’s worst-cost function, allowing equality up to constant factors and not requiring a matching lower bound?

A. Big O notation (O)

B. Theta notation (Θ)

C. Omega notation (Ω)

D. Small o notation (o)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Big O notation (O)

Big-O states an eventual constant-factor upper bound and allows the bounded function to have the same order as its comparator. Θ additionally requires a lower bound, Ω is a lower-bound relation, and little-o requires strictly slower growth. The worst-case function is selected separately by maximizing over equal-size inputs; Big-O itself does not mean worst case. The explicit upper-bound requirement uniquely identifies A.

Why each choice is right or wrong

A - Correct: O gives the requested non-strict upper bound.

B - Incorrect: Θ requires a matching lower bound as well.

C - Incorrect: Ω gives a lower bound.

D - Incorrect: Little-o requires strictly slower growth.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \le c \cdot g(n)$ for all $n \ge n_0$, with fixed $c > 0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Related reading (context, not primary note proof)

Study-note section DAA16

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It’s only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Data Structure and Algorithms module | PDF page 3 | lines 6-12

> II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n). III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth. IV. Little-o Notation (o): f(n)=o(g(n)) means f(n) has less growth rate compared to g(n). V. Little-Omega Notation (ω): ω notation used to denote a lower bound that is not asymptotically tight.

Variant 1 | Alternative version 1

For ordinary array insertion sort on n distinct keys, count comparisons and shifts as constant-time operations. Which is the tightest listed Big-O upper bound for its worst case?

A. O(1)

B. O(n)

C. O(log n)

D. O(n^2)

Answer D: O(n^2)

In the worst input order, each inserted key shifts past the entire sorted prefix. The total work is proportional to $1+2+\cdots+(n-1)=\Theta(n^2)$. Thus $O(n^2)$ is the tightest listed worst-case upper bound. Constant, logarithmic and linear bounds are too small. The supplied algorithm and cost model make this growth-class variant answerable without inventing a universal worst-case rate.

Why each choice is right or wrong

A - Incorrect: Constant cannot bound the quadratic worst-case work.

B - Incorrect: Linear cannot bound the summed prefix shifts.

C - Incorrect: Logarithmic is too small for quadratic work.

D - Correct: Insertion sort has $\Theta(n^2)$ worst work, so $O(n^2)$ is tight.

Primary note proof

Study-note section DAA16

> Insertion shifts larger sorted-prefix keys: stable with strict comparisons, Θ(n) best and $\Theta(n^2)$ worst.

Related reading (context, not primary note proof)

Study-note section DAA04

Question sources

Computer Science MCQ practice collection · Question 346

Computer Science MCQ practice collection · Question 357

Computer Science MCQ practice collection · Question 386

Question 39 | AM0347

For binary search on a nonempty sorted random-access array, what is the tight best-case time when the key may be at the initial midpoint? Use constant-time comparisons.

A. $\Theta(1)$

B. $\Theta(\log n)$

C. $\Theta(n)$

D. $\Theta(n^2)$

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: Θ(1)

If the sought key is at the first midpoint, binary search finds it after one constant-time comparison and returns. This is $\Theta(1)$ best-case time, regardless of the array length. The logarithmic bound describes its worst case, not this fastest legal input at fixed size. The linear and quadratic alternatives are likewise not tight for the midpoint-match case.

Why each choice is right or wrong

A - Correct: An initial midpoint match takes constant work.

B - Incorrect: Logarithmic describes worst-case search depth, not the tight best case.

C - Incorrect: Linear is not tight for a one-comparison match.

D - Incorrect: Quadratic is not tight for a one-comparison match.

Primary note proof

Study-note section DAA13

> Binary search requires sorted random-access data. Maintain inclusive low/high bounds, compute mid=low+(high−low)//2, and replace high with mid−1 or low with mid+1 according to the comparison. Stop when found or low>high. T(n)=T(n/2)+Θ(1): worst Θ(log n), best Θ(1); iterative auxiliary space Θ(1), recursive Θ(log n).

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Question sources

Computer Science MCQ practice collection · Question 347

Question 40 | AM0348

For ordinary array insertion sort on n distinct keys, with constant-time comparisons and shifts, what is the tight worst-case running time?

A. $\Theta(\log n)$

B. $\Theta(n)$

C. $\Theta(n \log n)$

D. $\Theta(n^2)$

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: $\Theta(n^2)$

A descending input makes each inserted key move across every earlier key in the sorted prefix. The work sums as $1+2+\cdots+(n-1)$, which is $\Theta(n^2)$. This establishes the tight worst case for the specified insertion-sort implementation. The smaller alternatives cannot upper-bound that quadratic work, and the answer is not a general claim about worst-case complexity of every algorithm.

Why each choice is right or wrong

A - Incorrect: Logarithmic is too small for the summed shifts.

B - Incorrect: Linear is too small for the summed shifts.

C - Incorrect: n log n is too small for quadratic worst-case shifts.

D - Correct: The sum of prefix lengths gives quadratic work.

Primary note proof

Study-note section DAA16

> Insertion shifts larger sorted-prefix keys: stable with strict comparisons, $\Theta(n)$ best and $\Theta(n^2)$ worst. Optimized bubble sort swaps adjacent inversions and stops after a no-swap pass: stable, $\Theta(n)$ best, $\Theta(n^2)$ worst. Selection repeatedly finds a suffix minimum: $\Theta(n^2)$ comparisons in every order, normally unstable. These use $\Theta(1)$ auxiliary array space.

Study-note section DAA06

> Sequential costs add; independent nested loop counts multiply. A dependent loop requires summation. n iterations of a constant body give $\Theta(n)$; $\sum i=1..\ n\ i=n(n+1)/2$ gives $\Theta(n^2)$. Doubling i=1,2,4,... until i≥n takes ceil(log 2 n) iterations for n>1. A 10n loop is linear under constant-cost output.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Question sources

Computer Science MCQ practice collection · Question 348

Question 41 | AM0349

Which of the following time complexities represents an algorithm with logarithmic growth?

A. O(1)

B. O(log n)

C. O(n)

D. O(n^2)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer B: O(log n)

Binary search takes time proportional to log_2 n. B matches logarithmic growth.

Why each choice is right or wrong

A - Incorrect: 1 is a constant, not a logarithm (p.14 growth comparison).

B - Correct: log n is the logarithmic class named on p.14 and instantiated in cross-module p.3.

C - Incorrect: n is linear rather than logarithmic (p.14).

D - Incorrect: n^2 is quadratic rather than logarithmic (p.14).

Primary note proof

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the

size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching • Works on ordered/sorted list • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list • The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science MCQ practice collection · Question 349

Computer Science MCQ practice collection · Question 361

Question 42 | AM0350

Which of the following time complexities represents an algorithm with linear growth?

A. O(1)

B. O(log n)

C. O(n)

D. O(n^2)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: O(n)

Linear growth means work proportional to input size n; sequential search takes O(n) time. The growth classes n, log n and n^2 are distinct.

Why each choice is right or wrong

A - Incorrect: 1 stays bounded instead of increasing proportionally to n (p.14).

B - Incorrect: log n increases more slowly than n (p.14 growth classes).

C - Correct: O(n) is the linear class illustrated by sequential search (cross-module p.3).

D - Incorrect: n^2 scales with the square rather than the first power (p.14).

Primary note proof

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Study-note section DAA06

> n iterations of a constant body give Θ(n);

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching • Works with unsorted List • Loop through the list starting at the first element • Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science MCQ practice collection · Question 350

Question 43 | AM0351

Which tight running-time growth rates are strictly superlinear, meaning f(n)/n→∞? Select all that apply.

A. Θ(log n)

B. Θ(n)

C. Θ(n log n)

D. Θ(n^2)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C, D: Θ(n log n); Θ(n^2)

For n log n, dividing by n gives log n, which grows without bound. For n^2, the ratio is n, also unbounded. Both are therefore superlinear. The logarithmic representative is sublinear and n itself has constant ratio one, so neither meets the strict condition. Using tight rates prevents a loose upper-bound label from being mistaken for an algorithm's actual growth.

Why each choice is right or wrong

A - Incorrect: log n grows more slowly than n.

B - Incorrect: n is linear, with ratio one.

C - Correct: The ratio to n is log n and diverges.

D - Correct: The ratio to n is n and diverges.

Primary note proof

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Question sources

Computer Science MCQ practice collection · Question 351

Question 44 | AM0352

What is the purpose of experimental analysis in algorithm analysis?

A. To measure the actual running time of an algorithm

B. To determine the best-case scenario for an algorithm

C. To analyze the behavior of an algorithm as the input size increases

D. To evaluate the correctness of an algorithm

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: To measure the actual running time of an algorithm

The module distinguishes analytical performance analysis from conducting experiments for performance measurement (PDF p.4). Experimental analysis therefore measures actual performance on chosen executions rather than deriving a limiting rate or proving correctness.

Why each choice is right or wrong

A - Correct: Direct runtime measurement is the performance-measurement purpose (p.4).

B - Incorrect: An experiment might include favorable inputs but does not inherently determine a universal best case (p.4).

C - Incorrect: Growth analysis is analytical; experiments can sample sizes but are not the defining limiting argument (p.4).

D - Incorrect: Correctness tests can accompany experiments, but the stated performance-measurement purpose is separate (p.4).

Primary note proof

Study-note section DAA02

> Experimental/dynamic measurement observes actual runs; asymptotic analysis abstracts growth; probabilistic analysis studies expectations. There is no universal taxonomy proving “graph analysis” or “dynamic analysis” is invalid. Auxiliary space excludes input; I/O complexity counts transfers. Representation affects costs; “data type complexity” is not a standard metric name.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Variant 1 | Alternative version 1

What is the purpose of experimental analysis in algorithm analysis?

A. To determine the best-case time complexity of an algorithm.

B. To compare algorithms based on their theoretical analysis.

C. To evaluate the correctness of an algorithm.

D. To measure the actual performance of an algorithm on real data.

Answer D: To measure the actual performance of an algorithm on real data.

Apply the same study-note rule to this alternative source set; the choice contrasts below show the derivation.

Why each choice is right or wrong

A - Incorrect: Experiments do not by themselves prove the minimum cost for every input size (p.4 distinction).

B - Incorrect: Comparison based on theoretical analysis uses the analytical approach (p.4).

C - Incorrect: Functional correctness is not the stated performance-measurement objective (p.4).

D - Correct: Measured performance on real data is the experimental approach (p.4).

Primary note proof

Study-note section DAA02

Experimental/dynamic measurement observes actual runs; asymptotic analysis abstracts growth; probabilistic analysis studies expectations. There is no universal taxonomy proving “graph analysis” or “dynamic analysis” is invalid. Auxiliary space excludes input; I/O complexity counts transfers. Representation affects costs; “data type complexity” is not a standard metric name.

Question sources

Computer Science MCQ practice collection · Question 352

Computer Science MCQ practice collection · Question 362

Question 45 | AM0353

Which offered phrase is not a standard resource-metric name in the course's time/space/I-O cost model? This does not ask whether representation affects cost.

A. Space complexity

B. Auxiliary space

C. Input-output complexity

D. Data type complexity

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer D: Data type complexity

Space complexity measures storage, auxiliary space isolates additional storage beyond input, and I/O complexity counts transfers. These are resource metrics in the stated model. Data type complexity is not the standard metric name used for representation effects; those effects are accounted for through arithmetic, storage or bit-operation costs. Therefore D is the terminology exception without asserting that data representation is irrelevant.

Why each choice is right or wrong

A - Incorrect: Space complexity is a standard storage metric.

B - Incorrect: Auxiliary space is a standard additional-storage metric.

C - Incorrect: I/O complexity is a standard transfer metric.

D - Correct: Data type complexity is not the standard metric name in this model.

Primary note proof

Study-note section DAA02

> Experimental/dynamic measurement observes actual runs; asymptotic analysis abstracts growth; probabilistic analysis studies expectations. There is no universal taxonomy proving "graph analysis" or "dynamic analysis" is invalid. Auxiliary space excludes input; I/O complexity counts transfers. Representation affects costs; "data type complexity" is not a standard metric name.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

> The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to

be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Question sources

Computer Science MCQ practice collection · Question 353

Question 46 | AM0354

What is the purpose of algorithm analysis in computer science?

A. To design algorithms for specific problems

B. To implement algorithms using programming languages

C. To analyze the efficiency and performance of algorithms

D. To test algorithms for correctness

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: To analyze the efficiency and performance of algorithms

Performance analysis estimates computer memory and time requirements (PDF p.4), and studying those requirements helps choose between algorithms (p.6). Thus efficiency/performance is the direct purpose asked for.

Why each choice is right or wrong

A - Incorrect: Design creates a method; p.4 analysis estimates resources of a method.

B - Incorrect: Implementation expresses a method in code; p.4 analysis estimates performance.

C - Correct: This directly matches the memory-and-time performance definition (p.4).

D - Incorrect: Correctness is necessary but is distinct from the p.4 performance analysis objective.

Primary note proof

Study-note section DAA02

Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to

bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Question sources

Computer Science MCQ practice collection · Question 354

Question 47 | AM0355

Which can be legitimate techniques for analyzing algorithm resource costs? Select all that apply. Dynamic means observing actual executions; empirical means measuring runs.

A. Asymptotic analysis

B. Empirical analysis

C. Dynamic analysis

D. Probabilistic analysis

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A, B, C, D: Asymptotic analysis; Empirical analysis; Dynamic analysis; Probabilistic analysis

Asymptotic reasoning studies growth, empirical measurements observe actual runs, and dynamic analysis under the stated meaning also observes execution costs. Probabilistic analysis studies expected cost under a distribution or internal randomness. All four can therefore be legitimate resource-analysis techniques. Empirical and dynamic may overlap; they do not have to be disjoint categories for their methods to be useful.

Why each choice is right or wrong

A - Correct: Asymptotic analysis studies cost growth.

B - Correct: Empirical analysis measures actual runs.

C - Correct: Dynamic observation measures actual execution costs under the given definition.

D - Correct: Probabilistic analysis studies expectations.

Primary note proof

Study-note section DAA02

> Experimental/dynamic measurement observes actual runs; asymptotic analysis abstracts growth; probabilistic analysis studies expectations. There is no universal taxonomy proving “graph analysis” or “dynamic analysis” is invalid. Auxiliary space excludes input; I/O complexity counts transfers. Representation affects costs; “data type complexity” is not a standard metric name.

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

> 1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the

> worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Computer Science MCQ practice collection · Question 355

Question 48 | AM0358

What does a time complexity of O(1) indicate?

A. The algorithm runs in constant time.

B. The algorithm runs in logarithmic time.

C. The algorithm runs in linear time.

D. The algorithm runs in quadratic time.

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: The algorithm runs in constant time.

A constant upper bound O(1) does not grow with input size; it is the constant K at the start of the visually inspected growth ordering in Data Structures and Algorithms PDF p.3. The assigned PDF p.14 motivates classifying by growth rather than exact operation count.

Why each choice is right or wrong

A - Correct: Constant time has a fixed upper bound independent of n (cross-module p.3).

B - Incorrect: Logarithmic time grows with n, unlike 1 (assigned p.14).

C - Incorrect: Linear time grows proportionally to n (assigned p.14).

D - Incorrect: Quadratic time grows as n^2 (assigned p.14).

Primary note proof

Study-note section DAA04

For eventually nonnegative f, g, f∈O(g) means f(n)≤c·g(n) for all n≥n 0, with fixed c>0. Ω reverses that inequality; Θ requires both bounds. Little-o means f/g→0, and little-ω means f/g→∞ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA05

Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Computer Science MCQ practice collection · Question 358

Question 49 | AM0359

Which of the following time complexities indicates the most efficient algorithm?

A. O(1)

B. O(log n)

C. O(n)

D. O(n^2)

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer A: O(1)

When comparing the listed representative growth rates for the same task and cost model as n becomes large, constant growth is smallest. The visually inspected related-module PDF p.3 ordering places K before log n, n and n^2; assigned p.14 focuses on large-input growth. This is not a guarantee about clock time for every finite input.

Why each choice is right or wrong

A - Correct: Constant is the smallest representative listed growth (cross-module p.3 diagram).

B - Incorrect: log n grows beyond any fixed constant (p.14 growth interpretation).

C - Incorrect: n grows faster than constant and logarithmic rates (p.14).

D - Incorrect: n^2 grows fastest among these listed rates (p.14).

Primary note proof

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What

happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Computer Science MCQ practice collection · Question 359

Question 50 | AM0360

What does a time complexity of O(n^2) signify?

A. The algorithm runs in linear time.

B. The algorithm runs in logarithmic time.

C. The algorithm runs in quadratic time.

D. The algorithm runs in exponential time.

Hint

Identify the specification and cost model, then apply the relevant bound, invariant or strategy rule.

Answer C: The algorithm runs in quadratic time.

The exponent 2 identifies the quadratic representative growth class n^2, listed on PDF p.14. Strictly O(n^2) is an upper bound (cross-module p.3), so it does not prove that an algorithm actually requires quadratic time. C is the conventional class-name answer.

Why each choice is right or wrong

A - Incorrect: Linear is the n class, not the name of n^2 (p.14).

B - Incorrect: Logarithmic is the log n class, distinct from n^2 (p.14).

C - Correct: Quadratic names the squared-input growth class (p.14).

D - Incorrect: Exponential uses a growing exponent, such as 2^n, separately listed on p.14.

Primary note proof

Study-note section DAA04

For eventually nonnegative f, g, f∈O(g) means f(n)≤c·g(n) for all n≥n 0, with fixed c>0. Ω reverses that inequality; Θ requires both bounds. Little-o means f/g→0, and little-ω means f/g→∞ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA05

Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Question sources

Computer Science MCQ practice collection · Question 360

Question 51 | AM0363

In fixed unit-cost RAM analysis of operation count and storage as functions of input size, which factor is abstracted away rather than being a parameter of the analysis?

A. Time complexity

B. Space complexity

C. Input size

D. Hardware configuration

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: Hardware configuration

The fixed RAM model counts operations and storage relative to input size. Time complexity, space complexity and input size are therefore central to the analysis. The particular hardware configuration is abstracted away by the stipulated unit costs. It would matter for actual elapsed measurements, alongside compiler, cache, I/O and load, but that is a different analysis setting from the one explicitly requested.

Why each choice is right or wrong

A - Incorrect: Operation-count time complexity is a central resource.

B - Incorrect: Space complexity is a central resource.

C - Incorrect: Costs are expressed as functions of input size.

D - Correct: The fixed unit-cost abstraction hides the specific hardware configuration.

Primary note proof

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

> 1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

> The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Design and Analysis of Algorithms module | PDF page 10 | lines 6-20

> Analysis Rules: 1. We assume an arbitrary time unit. 2. Execution of one of the following operations takes time 1: • Assignment Operation • Single Input/output Operation • Single Boolean

Operations • Single Arithmetic Operations • Function Return 3. Running time of a selection statement (if, switch) is the time for the condition evaluation plus the maximum of the running times for the individual clauses in the selection. 4. Loops: Running time for a loop is equal to the running time for the statements inside the loop multiplied by the number of iterations. The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops. For nested loops, analyze inside out.

Design and Analysis of Algorithms module | PDF page 6 | lines 18-30

Components of space complexity: - the space needed by a program has the following components: • Instruction space: - is the amount of space needed to store the compiled version of the program instructions. The amount of instruction space that is needed depends on factors such as: - The compiler used to compile the program into machine code. - The compiler options in effect at time of compilation. - The target computer. • Data space: - is the space needed to store all constant and variable values. Data space has two components: i. Space needed by constants (e.g. the numbers 0, 1, 2...) and simple variables. ii. Space needed by dynamically allocated objects such as arrays and class instances. • Environmental stack space:- is the space used to save information needed to resume execution of partially completed methods (functions). For example, if method func1 invokes or calls method func2, then we must at least save a pointer to the instruction of func1 to be executed when func2

Question sources

Computer Science MCQ practice collection · Question 363

Question 52 | AM0364

Which technique solves smaller largely independent instances recursively and then combines their results, without relying on reuse of overlapping states?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A: Divide and conquer

Divide and conquer separates a problem into smaller largely independent instances, solves them and combines the results. Dynamic programming instead emphasizes defined reusable states when subproblems overlap. Greedy commits to safe local choices, while backtracking explores and retracts partial candidates. The independence and combination conditions distinguish the requested technique from these other ways of reducing or extending a problem.

Why each choice is right or wrong

A - Correct: Independent recursive subinstances and combination define divide and conquer.

B - Incorrect: Greedy is defined by safe irrevocable local choices.

C - Incorrect: DP emphasizes reusable subproblem states rather than this independent structure.

D - Incorrect: Backtracking is defined by candidate extension and retraction.

Primary note proof

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results. DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it. Backtracking grows/retracts candidates when constraints fail; branch and bound also rejects candidates whose best possible objective cannot beat an incumbent.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 17 | lines 5-9

> 2.1 Divide and Conquer Algorithm In this approach, we solve a problem recursively by applying 3 steps 1. DIVIDE-break the problem into several sub problems of smaller size. 2. CONQUER-solve the problem recursively. 3. COMBINE-combine these solutions to create a solution to the original problem.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 95 | lines 3-17

> 5. Introduction to Backtracking 5.1 Backtracking: • It is one of the most general algorithm design techniques. • Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation. • To apply backtracking method, the desired solution must be expressible as an n- tuple (x1…xn) where xi is chosen from some finite set Si. • The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn). • The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (mi+1…….mn) possible test

vectors may be ignored entirely. • Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

Question sources

Computer Science MCQ practice collection · Question 364

Question 53 | AM0365

Which algorithm design technique guarantees finding the optimal solution at each step but may not lead to the overall optimal solution?

A. Greedy algorithm

B. Divide and conquer

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A: Greedy algorithm

Greedy makes the best available local choice based on current information (PDF p.27), and the module explicitly warns that such choices can fail to yield a global optimum (p.30). “Optimal at each step” must mean locally best, not globally guaranteed.

Why each choice is right or wrong

A - Correct: Local short-sighted choices and possible global failure match pp.27,30.

B - Incorrect: Independent splitting and combining does not define a locally best choice (p.27 contrast).

C - Incorrect: Dynamic programming retains subproblem alternatives rather than irrevocably picking only the local best (p.30 contrast).

D - Incorrect: Backtracking revisits choices, unlike non-recoverable greedy choices on p.30.

Primary note proof

Study-note section DAA19

A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 27 | lines 7-18

• Greedy method is the most straightforward designed technique. • As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decisions may have in the future. DEFINITION: • A problem with N inputs will have some constraints. Any subsets that satisfy these constraints are called a feasible solution. • A feasible solution that either maximize can minimize a given objectives function is called an optimal solution. Greedy is a strategy that works well on optimization problems with the following characteristics: • Greedy-choice property: A global optimum can be arrived at by selecting a local

Design and Analysis of Algorithms module | PDF page 30 | lines 10-14

Greedy algorithms sometimes fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the travelling salesman problem. Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that

Question sources

Computer Science MCQ practice collection · Question 365

Question 54 | AM0366

Which algorithm design technique is used to solve problems by exploring all possible solutions?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: Backtracking

Backtracking generates solution tuples and explores choices with constraints, returning to earlier positions when needed (PDF pp.95-96). It can enumerate all valid solutions while pruning branches proved unpromising, so “all possible solutions” should not be read as requiring every invalid tuple to be visited.

Why each choice is right or wrong

A - Incorrect: Divide and conquer combines independent solved parts rather than enumerating candidate tuples (pp.95-96 contrast).

B - Incorrect: Greedy follows one selection sequence rather than revisiting alternatives (pp.95-96 contrast).

C - Incorrect: Dynamic programming caches subproblems rather than being the tuple-enumeration method shown on p.96.

D - Correct: The p.96 schema generates all solutions and backtracks over choices.

Primary note proof

Study-note section DAA36

> Represent a partial assignment x[1.. k]. For each allowed next choice: assign it, test whether an extension remains possible, recurse if promising, and undo the assignment on return. Report complete feasible candidates; stop after one only if the task requests existence/a single answer. Explicit constraints restrict each variable’s domain; implicit constraints relate choices.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 95 | lines 3-17

> 5. Introduction to Backtracking 5.1 Backtracking: • It is one of the most general algorithm design techniques. • Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation. • To apply backtracking method, the desired solution must be expressible as an n- tuple (x1…xn) where xi is chosen from some finite set Si. • The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn). • The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (mi+1…….mn) possible test vectors may be ignored entirely. • Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

Design and Analysis of Algorithms module | PDF page 96 | lines 3-21

> Algorithm Backtracking (n) // This schema describes the backtracking procedure. All solutions are generated in X[1:n] //and printed as soon as they are determined. { k=1; While (k ≠ 0) do { if (there remains all untried X[k] ∈ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true) then { if(X[1],……X[k])is the path to the answer node) Then write(X[1:k]); k=k+1; //consider the next step. } else k=k-1; //consider backtracking to the previous set. }} • All solutions are generated in X[1:n] and printed as soon as they are determined. • T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],……..X[k-1] have already been chosen.

Question sources

Computer Science MCQ practice collection · Question 366

Question 55 | AM0367

Which algorithm design technique is suitable for problems with overlapping subproblems and optimal substructure?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: Dynamic programming

Shared sub-subproblems motivate saving results once (PDF p.48); optimal substructure makes an optimal solution a combination of optimal subsolutions (pp.28,49). Together these are the properties for the module's dynamic-programming optimization method.

Why each choice is right or wrong

A - Incorrect: Independent subproblems distinguish divide and conquer in p.48.

B - Incorrect: Optimal substructure alone does not prove the greedy-choice property (pp.28,49).

C - Correct: Caching shared subproblem solutions with optimal substructure is exactly the method on pp.48-49.

D - Incorrect: Backtracking searches constrained tuples; it need not exploit repeated subproblems via a table (p.48 contrast).

Primary note proof

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Study-note section DAA26

> Memoization recursively checks a cache before solving a state and stores its result. Tabulation orders dependencies explicitly; chosen domains may exclude irrelevant states. Cache keys include all relevant parameters; recursion/locality/sparsity affect implementation cost.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 28 | lines 3-4

> • Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems.

Design and Analysis of Algorithms module | PDF page 49 | lines 2-12

> The Principle of Optimality The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal. The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its sub problems. The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which sub problems are relevant to the problem under consideration. General method-multistage graphs-all pair shortest path algorithm-Optimal Binary Search trees- 0/1 knapsack Problem-Reliability design-traveling salesman problem

Question sources

Computer Science MCQ practice collection · Question 367

Computer Science MCQ practice collection · Question 378

Question 56 | AM0368

Which technique builds an optimum by repeatedly making a locally attractive feasible choice, never undoing it, when an exchange or safety proof justifies each commitment?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer B: Greedy algorithm

Greedy algorithms repeatedly commit to locally attractive feasible choices without later undoing them. An exchange or related safety proof is needed to show that these commitments can form an optimum. Dynamic programming compares reusable states, backtracking revises candidates, and divide and conquer combines independent subinstances. The explicit safety and irrevocability conditions distinguish greedy from the broad idea of an optimal choice sequence.

Why each choice is right or wrong

A - Incorrect: Divide and conquer relies on independent decomposition and combination.

B - Correct: Safe irrevocable local commitments define greedy.

C - Incorrect: DP evaluates state alternatives rather than requiring irrevocable local choices.

D - Incorrect: Backtracking undoes choices, contrary to the stated rule.

Primary note proof

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 27 | lines 7-18

> • Greedy method is the most straightforward designed technique. • As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decisions may have in the future. DEFINITION: • A problem with N inputs will have some constraints. Any subsets that satisfy these constraints are called a feasible solution. • A feasible solution that either maximize can minimize a given objectives function is called an optimal solution. Greedy is a strategy that works well on optimization problems with the following characteristics: • Greedy-choice property: A global optimum can be arrived at by selecting a local

Design and Analysis of Algorithms module | PDF page 49 | lines 2-12

> The Principle of Optimality The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal. The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its sub problems. The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which sub problems are relevant to the problem under consideration. General method-multistage graphs-all pair shortest path algorithm-Optimal Binary Search trees- 0/1 knapsack Problem-Reliability design-traveling salesman problem

Design and Analysis of Algorithms module | PDF page 49 | lines 27-31

• The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated. • In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

Question sources

Computer Science MCQ practice collection · Question 368

Question 57 | AM0369

Which algorithm design technique is suitable for solving problems with optimal substructure but does not involve overlapping subproblems?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A: Divide and conquer is the supported independent-subproblem strategy.

DAA12 characterizes divide and conquer by smaller largely independent instances. The stated absence of overlapping subproblems fits that structure. Optimal substructure alone does not establish greedy safety, as DAA19 warns, while DP's main reuse motivation is overlap. Backtracking needs a candidate-and-constraint search description absent here. A is therefore the supported textbook structural match, rather than a proof that only it can ever apply.

Why each choice is right or wrong

A - Correct: Independent smaller instances with combinable optimal results fit divide and conquer.

B - Incorrect: Optimal substructure alone does not prove a safe greedy local choice.

C - Incorrect: Without overlapping states, the stated premise does not motivate DP reuse.

D - Incorrect: No candidate retraction or constraint-based enumeration is specified to identify backtracking.

Primary note proof

Study-note section DAA12

Divide and conquer solves smaller largely independent instances and combines results. DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it. Backtracking grows/retracts candidates when constraints fail;

branch and bound also rejects candidates whose best possible objective cannot beat an incumbent.

Study-note section DAA19

Optimal substructure alone does not prove this safe.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 27 | lines 7-18

• Greedy method is the most straightforward designed technique. • As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decisions may have in the future. DEFINITION: • A problem with N inputs will have some constraints. Any subsets that satisfy these constraints are called a feasible solution. • A feasible solution that either maximize can minimize a given objectives function is called an optimal solution. Greedy is a strategy that works well on optimization problems with the following characteristics: • Greedy-choice property: A global optimum can be arrived at by selecting a local

Design and Analysis of Algorithms module | PDF page 28 | lines 3-4

• Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems.

Question sources

Computer Science MCQ practice collection · Question 369

Question 58 | AM0370

Which technique explores a series of candidate choices, retracting a choice when constraints show that its partial candidate cannot extend to a solution?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: Backtracking

Backtracking extends a partial assignment, tests whether a feasible completion remains possible, and undoes the assignment when returning to try alternatives. That matches the stated constraint-driven retraction. Greedy keeps its choices, DP stores reusable state values, and divide and conquer combines independent instances. Merely having a series of choices would not distinguish them; the explicit undo-and-retry mechanism does.

Why each choice is right or wrong

A - Incorrect: Independent decomposition is not the described candidate retraction.

B - Incorrect: Greedy does not undo committed choices.

C - Incorrect: DP is characterized by reuse of states rather than this search rule.

D - Correct: Backtracking extends, checks, retracts and tries alternatives.

Primary note proof

Study-note section DAA36

> Represent a partial assignment x[1.. k]. For each allowed next choice: assign it, test whether an extension remains possible, recurse if promising, and undo the assignment on return. Report complete feasible candidates; stop after one only if the task requests existence/a single answer. Explicit constraints restrict each variable’s domain; implicit constraints relate choices.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 17-22

> Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions. The most attractive property of this strategy is that during the search for a solution it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution. In many practical situations, this strategy hits the optimal solution in a polynomial number of decision steps. However, in the worst case, such a strategy may end up performing full enumeration.

Design and Analysis of Algorithms module | PDF page 95 | lines 3-17

> 5. Introduction to Backtracking 5.1 Backtracking: • It is one of the most general algorithm design techniques. • Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation. • To apply backtracking method, the desired solution must be expressible as an n- tuple (x1…xn) where xi is chosen from some finite set Si. • The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn). • The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (mi+1…….mn) possible test vectors may be ignored entirely. • Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

Design and Analysis of Algorithms module | PDF page 49 | lines 27-31

> • The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated. • In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

Question sources

Computer Science MCQ practice collection · Question 370

Question 59 | AM0371

Which listed techniques can support exact optimization when their required structural or exhaustive-search conditions hold? Select all that apply; consider maximum subarray, fractional knapsack, 0/1 knapsack and complete tour enumeration as examples.

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A, B, C, D: Divide and conquer; Greedy algorithm; Dynamic programming; Backtracking

Divide and conquer optimizes maximum subarray by comparing left, right and crossing results. Greedy optimizes fractional knapsack under divisibility, DP optimizes 0/1 knapsack states, and complete backtracking can enumerate tours and retain the cheapest. All four techniques therefore have exact optimization applications under their own conditions. None becomes universally appropriate merely because the task asks for an optimum.

Why each choice is right or wrong

A - Correct: Maximum-subarray divide and conquer is an exact optimizer.

B - Correct: Density-greedy is exact for divisible fractional knapsack.

C - Correct: 0/1 knapsack DP computes an exact optimum.

D - Correct: Complete tour enumeration can retain the exact minimum.

Primary note proof

Study-note section DAA18

> Divide-and-conquer compares the best left, best right and best crossing result.

Study-note section DAA20

> For positive weights/profits and divisible items, sort by decreasing value/weight, take complete items while they fit, then the fitting fraction of the next.

Study-note section DAA29

> For integer capacity W, K[i, w] is maximum value using the first i items within capacity w.

Study-note section DAA33

> Greedy nearest-neighbor is a heuristic without a general exactness guarantee; backtracking and branch and bound can also solve exact TSP.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 27 | lines 7-18

> • Greedy method is the most straightforward designed technique. • As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decisions may have in the future. DEFINITION: • A problem with N inputs will have some constraints. Any subsets that satisfy these constraints are called a feasible solution. • A feasible solution that either maximize can minimize a given objectives function is called an optimal solution. Greedy is a strategy that works well on optimization problems with the following characteristics: • Greedy-choice property: A global optimum can be arrived at by selecting a local

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 95 | lines 3-17

> 5. Introduction to Backtracking 5.1 Backtracking: • It is one of the most general algorithm design techniques. • Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation. • To apply backtracking method, the desired solution must be expressible as an n- tuple (x1…xn) where xi is chosen from some finite set Si. • The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn). • The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (mi+1…….mn) possible test vectors may be ignored entirely. • Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

Question sources

Computer Science MCQ practice collection · Question 371

Question 60 | AM0372

An exponential recursive optimization repeatedly solves the same well-defined states. Optimal substructure holds and there are only polynomially many distinct states with polynomial work per state. Which technique avoids the repeated work by storing state results?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: Dynamic programming

Dynamic programming stores the result for each distinct state, through memoization or tabulation, and reuses it instead of repeating equivalent recursive work. With the stated polynomial state count and polynomial work per state, this yields polynomial total work. Greedy needs an additional local-choice safety argument, divide and conquer targets independent instances, and plain backtracking does not provide the specified reuse.

Why each choice is right or wrong

A - Incorrect: Independent subproblem decomposition does not address the stated repeated states.

B - Incorrect: No safe irrevocable local-choice rule is supplied.

C - Correct: DP stores and reuses the repeated state results.

D - Incorrect: Plain candidate backtracking does not supply the requested cache reuse.

Primary note proof

Study-note section DAA25

Overlapping states motivate reuse; optimal substructure supports optimization. These are textbook signatures, not requirements that every DP problem optimize or every state have multiple users. DP also counts/decides. Cost=states×work per state and can remain exponential.

Study-note section DAA26

Memoization recursively checks a cache before solving a state and stores its result. Tabulation orders dependencies explicitly; chosen domains may exclude irrelevant states. Cache keys include all relevant parameters; recursion/locality/sparsity affect implementation cost.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 48 | lines 17-22

Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions. The most attractive property of this strategy is that during the search for a solution it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution. In many practical situations, this strategy hits the optimal solution in a polynomial number of decision steps. However, in the worst case, such a strategy may end up performing full enumeration.

Design and Analysis of Algorithms module | PDF page 95 | lines 3-17

5. Introduction to Backtracking 5.1 Backtracking: • It is one of the most general algorithm design techniques. • Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation. • To apply backtracking method, the desired solution must be expressible as an n- tuple (x1…xn) where xi is chosen from some finite set Si. • The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn). • The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (mi+1…….mn) possible test vectors may be ignored entirely. • Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

Question sources

Computer Science MCQ practice collection · Question 372

Question 61 | AM0373

Which technique combines solutions of overlapping subproblems using a defined state recurrence and reuses each solved state rather than solving it repeatedly?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: Dynamic programming

Dynamic programming defines states and a recurrence, solves dependencies in a valid order and reuses the resulting values. Combining those values forms larger solutions without repeating the same state work. Divide and conquer combines largely independent instances instead, greedy keeps safe local commitments, and backtracking grows and retracts candidates. The explicit overlap and reuse conditions make DP the specific match.

Why each choice is right or wrong

A - Incorrect: Divide and conquer emphasizes independent instances rather than reused overlapping states.

B - Incorrect: Greedy is defined by safe local commitments.

C - Correct: DP combines stored dependent state values.

D - Incorrect: Backtracking is defined by candidate search and retraction.

Primary note proof

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Study-note section DAA26

> Memoization recursively checks a cache before solving a state and stores its result. Tabulation orders dependencies explicitly; chosen domains may exclude irrelevant states. Cache keys include all relevant parameters; recursion/locality/sparsity affect implementation cost.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 17 | lines 5-9

> 2.1 Divide and Conquer Algorithm In this approach, we solve a problem recursively by applying 3 steps 1. DIVIDE-break the problem into several sub problems of smaller size. 2. CONQUER-solve the problem recursively. 3. COMBINE-combine these solutions to create a solution to the original problem.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Question sources

Computer Science MCQ practice collection · Question 373

Question 62 | AM0374

For nonnegative weighted edges, an algorithm repeatedly finalizes the unsettled vertex with minimum current source distance and relaxes its outgoing edges. Which design technique describes this Dijkstra procedure?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer B: Greedy algorithm

Dijkstra chooses the unsettled vertex of minimum distance and commits to its finalized value. Nonnegative edges ensure that a later path through other unsettled vertices cannot improve it. That safe irrevocable minimum choice is greedy. DP methods also solve other shortest-path settings, but they are not the explicitly described finalization procedure; independent decomposition and retracting path search are likewise different designs.

Why each choice is right or wrong

A - Incorrect: The procedure does not solve independent graph subinstances and combine them.

B - Correct: Safe minimum-distance finalization is a greedy choice.

C - Incorrect: A DP path recurrence is a different method from the specified Dijkstra procedure.

D - Incorrect: The algorithm finalizes distances rather than retracting path candidates.

Primary note proof

Study-note section DAA24

> Initialize dist[source]=0, others∞ and predecessors unset. Repeatedly extract the unsettled vertex u of minimum distance; relax each edge by dist[v]=min(dist[v], dist[u]+w(u, v)), updating predecessors when improved. Once the minimum is∞, remaining vertices are unreachable. A destination may be finalized before stopping, not merely discovered.

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 46 | lines 2-4

> o Single Source Shortest Path Algorithm (Dijkstra's) Single-source shortest path: Graphs can be used to represent the highway structure of a state or country with

Design and Analysis of Algorithms module | PDF page 61 | lines 14-23

> 4.2. ALL PAIR SHORTEST PATH • Let G=<V,E> be a directed graph 'V' is a set of nodes and _E' is the set of edges. • Each edge has an associated non-negative length. • We want to calculate the length of the shortest path between each pair of nodes. • Suppose the nodes of G are numbered from 1 to n, so V={1,2,...N},and suppose G matrix L gives the length of each edge, with L(i,j)=0 for i=1,2...n,L(i,j)>=for all i & j, and L(i,j)=infinity, if the edge (i,j) does not exist. • The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.

Design and Analysis of Algorithms module | PDF page 49 | lines 2-12

> The Principle of Optimality The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal. The

principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its sub problems. The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which sub problems are relevant to the problem under consideration. General method-multistage graphs-all pair shortest path algorithm-Optimal Binary Search trees- 0/1 knapsack Problem-Reliability design-traveling salesman problem

Question sources

Computer Science MCQ practice collection · Question 374

Question 63 | AM0375

Which algorithm design technique is commonly used for sorting algorithms?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A: Divide and conquer

Merge sort splits into independent halves and merges (PDF p.21), and quick sort partitions into independently sorted subarrays (p.23). These provide the concrete common sorting examples for divide and conquer. The answer is about those taught examples, not a claim that every sort uses this technique.

Why each choice is right or wrong

A - Correct: Both merge sort and quick sort are explicitly divide and conquer (pp.21,23).

B - Incorrect: No greedy-choice sorting construction is the taught example in pp.21-23.

C - Incorrect: No tabulated shared-subproblem sorting construction is presented in pp.21-23.

D - Incorrect: No candidate-solution backtracking sort is the taught example in pp.21-23.

Primary note proof

Study-note section DAA14

Split by indices, recursively sort each half, then repeatedly copy the smaller remaining front element into a temporary result. The merge invariant says the output prefix is sorted and contains the smallest elements already consumed. Append the unexhausted suffix. Equal keys must be taken from the left half first to preserve stability.

Study-note section DAA15

Choose a pivot, partition, then sort the resulting smaller ranges. In a two-pointer scheme, advance i while values are below pivot and j while above; when i≤j, swap and advance both. Bound the scans and ensure progress on equal values. Recurse on the resulting ranges. Hoare-style partition boundaries need not put the pivot at its final position; Lomuto-style partition commonly does. Do not mix their return conventions.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 21 | lines 18-29

MERGE SORT • As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n) • This algorithm is called merge sort • We assume throughout that the elements are to be sorted in non-decreasing order. • Given a sequence of _n' elements a[1],...,a[n] the general idea is to imagine then split into 2 sets a[1],.....,a[n/2] and a[[n/2]+1],....a[n]. • Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of _n' elements. • Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Design and Analysis of Algorithms module | PDF page 23 | lines 19-29

QUICK SORT • The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. • In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged. • In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later. • This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n. • Thus the elements in a[1:m] & a[m+1:n] can be independently sorted. • No merge is needed. This rearranging is referred to as partitioning.

Question sources

Computer Science MCQ practice collection · Question 375

Question 64 | AM0376

A sequence-decision problem has overlapping state subproblems and optimal substructure. Which technique evaluates all legal next choices through a recurrence and stores the best value for each state for reuse?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: Dynamic programming

Dynamic programming defines enough state to summarize the remaining decision problem, evaluates all legal choices and stores the best resulting value. Optimal substructure justifies using the stored optimal subsolutions, and overlap makes reuse useful. A mere decision sequence would not identify a technique; the explicit recurrence and reuse distinguish DP from greedy commitments, independent divide-and-conquer instances and backtracking without cached state values.

Why each choice is right or wrong

A - Incorrect: Independent decomposition does not describe the stated state reuse.

B - Incorrect: Greedy does not generally evaluate all choices through cached state recurrences.

C - Correct: DP evaluates and reuses best state values.

D - Incorrect: Backtracking without state caching revisits candidate choices rather than implementing this reuse rule.

Primary note proof

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Study-note section DAA26

> Memoization recursively checks a cache before solving a state and stores its result. Tabulation orders dependencies explicitly; chosen domains may exclude irrelevant states. Cache keys include all relevant parameters; recursion/locality/sparsity affect implementation cost.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 17-22

> Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions. The most attractive property of this strategy is that during the search for a solution it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution. In many practical situations, this strategy hits the optimal solution in a polynomial number of decision steps. However, in the worst case, such a strategy may end up performing full enumeration.

Design and Analysis of Algorithms module | PDF page 49 | lines 27-31

> • The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated. • In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

Design and Analysis of Algorithms module | PDF page 95 | lines 3-17

5. Introduction to Backtracking 5.1 Backtracking: • It is one of the most general algorithm design techniques. • Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation. • To apply backtracking method, the desired solution must be expressible as an n- tuple (x1…xn) where xi is chosen from some finite set Si. • The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn). • The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (mi+1…….mn) possible test vectors may be ignored entirely. • Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

Question sources

Computer Science MCQ practice collection · Question 376

Question 65 | AM0377

Which technique builds a candidate incrementally, checks constraints after each addition and undoes the last addition when it cannot lead to a complete feasible solution?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: Backtracking

Backtracking keeps a partial candidate, adds a choice, checks whether completion remains possible, and recursively explores promising extensions. On return it undoes the choice so other alternatives can be tried. Greedy also grows solutions but does not undo committed choices; DP builds reusable states rather than this candidate stack. Independent divide-and-conquer combination does not describe the specified constraint-search mechanism.

Why each choice is right or wrong

A - Incorrect: Independent subinstance combination is a different mechanism.

B - Incorrect: Greedy keeps rather than undoes its commitments.

C - Incorrect: DP is defined by state reuse rather than this candidate undo rule.

D - Correct: Constraint checks and undoing partial choices are backtracking.

Primary note proof

Study-note section DAA36

Represent a partial assignment x[1.. k]. For each allowed next choice: assign it, test whether an extension remains possible, recurse if promising, and undo the assignment on return. Report complete feasible candidates; stop after one only if the task requests existence/a single answer. Explicit constraints restrict each variable’s domain; implicit constraints relate choices.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 29 | lines 5-13

Algorithm Greedy (a,n) //a[1:n] contain the n inputs { solution =0;//Initialise the solution. For i=1 to n do { x=select(a); if(feasible(solution,x))then solution=union(solution,x); } return solution;

Design and Analysis of Algorithms module | PDF page 96 | lines 3-21

Algorithm Backtracking (n) // This schema describes the backtracking procedure. All solutions are generated in X[1:n] //and printed as soon as they are determined. { k=1; While (k ≠ 0) do { if (there remains all untried X[k] ∈ T (X[1],[2],.....X[k-1]) and Bk (X[1],.....X[k])) is true) then { if(X[1],.....X[k])is the path to the answer node) Then write(X[1:k]); k=k+1; //consider the next step. } else k=k-1; //consider backtracking to the previous set. }} • All solutions are generated in X[1:n] and printed as soon as they are determined. • T(X[1].....X[k-1]) is all possible values of X[k] gives that X[1],.....X[k-1] have already been chosen.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Question sources

Computer Science MCQ practice collection · Question 377

Question 66 | AM0379

Which technique finds maximum contiguous subarray by recursively comparing the best left-half, right-half and crossing subarray, with a linear-time crossing computation?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A: Divide and conquer

The stated construction splits the array, recursively solves each half and combines those results with the best crossing interval. This is divide and conquer, with recurrence $2T(n/2)+\Theta(n)=\Theta(n \log n)$. Kadane also solves the same problem, but uses a best-sum-ending-at-index DP recurrence rather than this construction. Greedy commitments and extend/retract search do not describe the specified three-way comparison.

Why each choice is right or wrong

A - Correct: Independent halves plus the crossing combination define divide and conquer.

B - Incorrect: No irrevocable local-choice rule defines this construction.

C - Incorrect: Kadane is DP, but it is not the described left/right/crossing algorithm.

D - Incorrect: The construction does not extend and retract candidate assignments.

Primary note proof

Study-note section DAA18

> A maximum subarray is contiguous; it is not an arbitrary subsequence. Divide-and-conquer compares the best left, best right and best crossing result. The crossing result is the best suffix of the left half plus best prefix of the right, found in $\Theta(n)$. This yields $2T(n/2)+\Theta(n)=\Theta(n \log n)$.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 17 | lines 5-9

> 2.1 Divide and Conquer Algorithm In this approach, we solve a problem recursively by applying 3 steps 1. DIVIDE-break the problem into several sub problems of smaller size. 2. CONQUER-solve the problem recursively. 3. COMBINE-combine these solutions to create a solution to the original problem.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Question sources

Computer Science MCQ practice collection · Question 379

Question 67 | AM0380

Which listed techniques describe exact TSP algorithms using either (i) cached subset-and-endpoint states or (ii) complete extend-and-retract enumeration of feasible tours while retaining the cheapest? Select all that apply.

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C, D: Dynamic programming; Backtracking

Held-Karp stores the best cost for a visited subset and endpoint, which is dynamic programming. Complete backtracking extends feasible tours, retracts choices and retains the least-cost complete tour. Both are exact constructions described by the stem. Greedy nearest-neighbor can produce a tour but has no general optimality guarantee, while independent divide-and-conquer combination is not either specified construction.

Why each choice is right or wrong

A - Incorrect: Neither specified construction is independent divide-and-conquer combination.

B - Incorrect: A greedy tour heuristic is not a general exact optimizer.

C - Correct: Cached subset-and-endpoint states form Held-Karp DP.

D - Correct: Complete extend-and-retract tour search is backtracking.

Primary note proof

Study-note section DAA33

Let F(S, j) be minimum cost from s through exactly S, ending at j∈S.

Study-note section DAA33

Held-Karp uses O(V²2^V) time and O(V 2^V) space, improving factorial enumeration but remaining exponential. Greedy nearest-neighbor is a heuristic without a general exactness guarantee; backtracking and branch and bound can also solve exact TSP.

Study-note section DAA36

For each allowed next choice: assign it, test whether an extension remains possible, recurse if promising, and undo the assignment on return.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 78 | lines 6-13

4.6. TRAVELLING SALESMAN PROBLEM Let G(V,E) be a directed graph with edge cost cij is defined such that cij >0 for all i and j and Cij =∞ ,if <i,j> € E. Let |V|=n and assume n>1. The traveling salesman problem is to find a tour of minimum cost. A tour of G is a directed cycle that include every vertex in V. The cost of the tour is the sum of cost of the edges on the tour. The tour is the shortest path that starts and ends at the same vertex i.e., 1.

Design and Analysis of Algorithms module | PDF page 49 | lines 2-12

The Principle of Optimality The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal. The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its sub problems. The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which sub problems are relevant to the problem under consideration. General method-multistage graphs-all pair shortest path algorithm-Optimal Binary Search trees- 0/1 knapsack Problem-Reliability design-traveling salesman problem

Design and Analysis of Algorithms module | PDF page 30 | lines 10-14

Greedy algorithms sometimes fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the travelling salesman problem. Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that

Design and Analysis of Algorithms module | PDF page 130 | lines 18-23

TRAVELLING SALESMAN PROBLEM INTRODUCTION: It is algorithmic procedures similar to backtracking in which a new branch is chosen and is there (bound there) until new branch is choosing for advancing. Travelling salesman is a minimization problem. It can be solved using Branch and bound by generating state space tree by calculating cost and upper bound.

Question sources

Computer Science MCQ practice collection · Question 380

Question 68 | AM0381

Which algorithm design technique is used for solving problems that can be solved by making the best local choice at each step?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer B: Greedy algorithm

The greedy-choice property selects a local optimum, and the method uses immediately available information (PDF p.27). The stem's best local choice identifies greedy. PDF p.30 warns that this is not automatically a global optimality proof.

Why each choice is right or wrong

A - Incorrect: Independent splitting is not the p.27 local-choice criterion.

B - Correct: Greedy directly matches the p.27 short-sighted local-selection description.

C - Incorrect: Dynamic programming compares subproblem solutions rather than committing solely to the best current choice (p.30 contrast).

D - Incorrect: Backtracking revisits choices rather than making the non-recoverable choices described on p.30.

Primary note proof

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 27 | lines 7-18

> • Greedy method is the most straightforward designed technique. • As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decisions may have in the future. DEFINITION:

• A problem with N inputs will have some constraints. Any subsets that satisfy these constraints are called a feasible solution. • A feasible solution that either maximize can minimize a given objectives function is called an optimal solution. Greedy is a strategy that works well on optimization problems with the following characteristics: • Greedy-choice property: A global optimum can be arrived at by selecting a local

Design and Analysis of Algorithms module | PDF page 30 | lines 10-14

Greedy algorithms sometimes fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the travelling salesman problem. Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that

Question sources

Computer Science MCQ practice collection · Question 381

Question 69 | AM0382

Which algorithm design technique is used for finding the longest common subsequence between two sequences?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: Dynamic programming using prefix-pair states.

DAA30 defines a state for each pair of prefix lengths. A match extends a diagonal state; a mismatch takes the larger of two neighboring prefix solutions. Filling states in dependency order avoids recomputing overlapping subproblems and gives $\Theta(mn)$ time. This is the note's dynamic-programming solution; no greedy safety rule or backtracking enumeration is supplied.

Why each choice is right or wrong

A - Incorrect: The given LCS construction reuses overlapping prefix states rather than only independently solving and combining disjoint pieces.

B - Incorrect: No rule for committing to a locally best character is justified; prefix alternatives must be compared.

C - Correct: The prefix-pair table and recurrence reuse optimal subproblem lengths, exactly the dynamic-programming technique described.

D - Incorrect: The note computes stored prefix states instead of exhaustively extending and undoing candidate subsequences.

Primary note proof

Study-note section DAA30

A subsequence preserves order but need not be contiguous. For prefixes of lengths i, j define L[i, j]. Empty-prefix rows/columns are 0. If X[i−1]=Y[j−1], set L[i, j]=1+L[i−1, j−1]; otherwise use max(L[i−1, j], L[i, j−1]). Keep prefix lengths i and j distinct from the zero-based character indices i−1 and j−1.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Design and Analysis of Algorithms module | PDF page 49 | lines 2-12

The Principle of Optimality The dynamic programming relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal. For example, in matrix chain multiplication problem, not only the value we are interested in is optimal but all the other entries in the table are also represent optimal. The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its sub problems. The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which sub problems are relevant to the problem under consideration. General method-multistage graphs-all pair shortest path algorithm-Optimal Binary Search trees- 0/1 knapsack Problem-Reliability design-traveling salesman problem

Question sources

Computer Science MCQ practice collection · Question 382

Question 70 | AM0383

Which algorithm design technique is used for solving problems that can be divided into smaller independent subproblems?

A. Divide and conquer

B. Greedy algorithm

C. Dynamic programming

D. Backtracking

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A: Divide and conquer

PDF p.48 explicitly describes divide and conquer as partitioning into independent subproblems, solving recursively, and combining solutions. The independence qualification makes this stem more specific than merely saying “subproblems.”

Why each choice is right or wrong

A - Correct: A directly matches independent partitioning and recursive solution (pp.17,48).

B - Incorrect: Greedy is distinguished by local feasible selection, not independent recursive partitioning (p.48 contrast).

C - Incorrect: Dynamic programming is motivated by subproblems sharing sub-subproblems (p.48).

D - Incorrect: Backtracking searches constrained assignments rather than the independent partition framework (p.48).

Primary note proof

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results. DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it. Backtracking grows/retracts candidates when constraints fail; branch and bound also rejects candidates whose best possible objective cannot beat an incumbent.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 17 | lines 5-9

> 2.1 Divide and Conquer Algorithm In this approach, we solve a problem recursively by applying 3 steps 1. DIVIDE-break the problem into several sub problems of smaller size. 2. CONQUER-solve the problem recursively. 3. COMBINE-combine these solutions to create a solution to the original problem.

Design and Analysis of Algorithms module | PDF page 48 | lines 5-16

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. As we saw in Chapter 2 of binary search, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Question sources

Computer Science MCQ practice collection · Question 383

Question 71 | AM0384

Which term refers to the amount of resources, such as time and memory, required by an algorithm to solve a problem?

A. Efficiency

B. Complexity

C. Optimality

D. Completeness

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer B: Complexity

Complexity expresses algorithm requirements in time and space (PDF p.6), with performance defined as needed computer memory and time (p.4). Efficiency is an evaluation of those requirements; complexity names the resource requirement being quantified.

Why each choice is right or wrong

A - Incorrect: Efficiency evaluates how economically resources are used, while p.6 calls their expression complexity.

B - Correct: Complexity is explicitly expressed in time and space requirements (p.6).

C - Incorrect: Optimality concerns the objective value of a solution, not the resources used to find it (p.6 contrast).

D - Incorrect: Completeness concerns fully solving a problem, not its memory/time quantity (p.4 context).

Primary note proof

Study-note section DAA02

Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 4 | lines 6-19

• Finiteness: Algorithm must complete after a finite number of steps. • Definiteness: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next. • Sequence: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted. • Feasibility: It must be possible to perform each instruction. • Correctness: It must compute correct answer for all possible legal inputs. • Language Independence: It must not depend on any one programming language. • Completeness: It must solve the problem completely. • Effectiveness: It must be possible to perform each step exactly and in a finite amount of time. • Efficiency: It must solve with the least amount of computational resources such as time and space. • Generality: Algorithm should be valid on all possible inputs • Input/Output: There must be a specified number of input values, and one or more result values.

Question sources

Computer Science MCQ practice collection · Question 384

Question 72 | AM0385

What is the purpose of analyzing the computational complexity of algorithms?

A. To determine the correctness of an algorithm

B. To measure the size of the input data

C. To evaluate the performance of an algorithm

D. To compare different programming languages

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: To evaluate the performance of an algorithm

Analysis estimates memory and time needed to solve an input problem (PDF p.4); these estimates help select between possible algorithms (p.6). Evaluating performance is therefore the direct purpose.

Why each choice is right or wrong

A - Incorrect: Correctness concerns whether the result is right; the p.4 performance definition concerns resources.

B - Incorrect: Input size is an argument of the analysis, not the ultimate performance-evaluation purpose (pp.4,6).

C - Correct: Evaluating resource performance directly matches pp.4,6.

D - Incorrect: The comparison is between algorithms and their requirements, not primarily programming languages (p.6).

Primary note proof

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 20-28

> 1.3. Complexity analysis 1.3.1. Space and Time Complexity Performance Analysis By performance of a program, we mean the amount of computer memory and time needed to run a program. There are two approaches to determine the performance of a program: - Analytical approach- used for performance analysis, - Experimental approach - conduct experiments for performance

measurement. What is Analysis: - The analysis of algorithm provides background information that a general idea of how long an algorithm will take for a given set. For each algorithm considered, we will come up with an estimate of how

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Design and Analysis of Algorithms module | PDF page 4 | lines 6-19

• Finiteness: Algorithm must complete after a finite number of steps. • Definiteness: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next. • Sequence: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted. • Feasibility: It must be possible to perform each instruction. • Correctness: It must compute correct answer for all possible legal inputs. • Language Independence: It must not depend on any one programming language. • Completeness: It must solve the problem completely. • Effectiveness: It must be possible to perform each step exactly and in a finite amount of time. • Efficiency: It must solve with the least amount of computational resources such as time and space. • Generality: Algorithm should be valid on all possible inputs • Input/Output: There must be a specified number of input values, and one or more result values.

Question sources

Computer Science MCQ practice collection · Question 385

Question 73 | AM0387

What does f(n)∈O(g(n)) mean, independently of whether f measures best, average, worst or space cost?

A. The best-case time complexity of an algorithm

B. The average-case time complexity of an algorithm

C. An eventual constant-factor upper bound: f(n)≤c·g(n) for fixed c>0 and all sufficiently large n.

D. The space complexity of an algorithm

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: An eventual constant-factor upper bound: f(n)≤c·g(n) for fixed c>0 and all sufficiently large n.

Big-O gives an eventual constant-factor upper bound: for some fixed c>0 and n_0, f(n)≤c·g(n) whenever n≥n_0. It does not choose best, average or worst inputs, nor does it specify time rather than space. Those choices determine the function being bounded. Option C states the general mathematical relation that applies across all of those analyses.

Why each choice is right or wrong

A - Incorrect: Best case is a way to define f, not the meaning of Big-O.

B - Incorrect: Average case defines an expectation, not the bound relation.

C - Correct: The eventual upper inequality is the definition of Big-O.

D - Incorrect: Space is one possible resource for f, not the definition of the notation.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, f∈O(g) means f(n)≤c·g(n) for all n≥n 0, with fixed c>0. Ω reverses that inequality; Θ requires both bounds. Little-o means f/g→0, and little-ω means f/g→∞ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the ―Big-Oh‖ estimate.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

> I. Big-Oh Notation (O): It’s only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Data Structure and Algorithms module | PDF page 3 | lines 6-12

II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n). III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth. IV. Little-o Notation (o): f(n)=o(g(n)) means f(n) has less growth rate compared to g(n). V. Little-Omega Notation (ω): ω notation used to denote a lower bound that is not asymptotically tight.

Question sources

Computer Science MCQ practice collection · Question 387

Question 74 | AM0388

Which time complexity indicates an algorithm that executes in constant time regardless of the input size?

A. O(1)

B. O(n)

C. O(log n)

D. O(n^2)

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A: O(1)

A bound independent of input size is constant O(1). This is the constant K in the visually inspected related-module p.3 growth diagram and agrees with classifying growth rather than exact operation count on assigned PDF p.14.

Why each choice is right or wrong

A - Correct: 1 is independent of n and gives the constant upper bound (cross-module p.3).

B - Incorrect: n grows as the input increases, unlike a constant (assigned p.14).

C - Incorrect: log n still grows, though slowly (assigned p.14).

D - Incorrect: n^2 increases quadratically rather than staying fixed (assigned p.14).

Primary note proof

Study-note section DAA04

For eventually nonnegative f, g, f∈O(g) means f(n)≤c·g(n) for all n≥n 0, with fixed c>0. Ω reverses that inequality; Θ requires both bounds. Little-o means f/g→0, and little-ω means f/g→∞ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA05

Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Computer Science MCQ practice collection · Question 388

Question 75 | AM0389

Which term describes the amount of memory required by an algorithm to solve a problem?

A. Time complexity

B. Space complexity

C. Computational complexity

D. Algorithmic complexity

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer B: Space complexity

PDF p.6 explicitly defines space complexity as the amount of memory space needed to run a program to completion. B is the specific resource name, while the other complexity terms are broader or refer to duration.

Why each choice is right or wrong

A - Incorrect: Time complexity measures time rather than memory (p.6).

B - Correct: Space complexity is the precise definition on p.6.

C - Incorrect: Computational complexity is broader than the memory-specific measure asked for (p.6 distinguishes resources).

D - Incorrect: Algorithmic complexity is broader; p.6 names the particular memory measure space complexity.

Primary note proof

Study-note section DAA08

> Space complexity measures peak simultaneously live storage, not the sum of every allocation ever made. Count input, auxiliary structures and stack according to the requested convention. n active recursion levels with constant-size frames need Θ(n) stack; depth log n with constant frames needs Θ(log n). The number of total calls does not determine peak depth.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 6 | lines 4-10

> The complexity of an algorithm can be expressed in terms of its space and time complexity. Space Complexity of a program: - is the amount of memory space it needs to run to completion. We are interested in the space complexity of a program for the following reasons: • If a program is to run on a multiuser computer system, then we may need to specify the amount of memory to be allocated for the program. • For any computer system, we would like to know in advance whether or not sufficient memory is available to run the program.

Design and Analysis of Algorithms module | PDF page 4 | lines 6-19

> • Finiteness: Algorithm must complete after a finite number of steps. • Definiteness: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next. • Sequence: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted. • Feasibility: It must be possible to perform each instruction. • Correctness: It must compute correct answer for all possible legal inputs. • Language Independence: It must not depend on any one programming language. • Completeness: It must solve the problem completely. • Effectiveness: It must be possible to perform each step exactly and in a finite amount of time. • Efficiency: It must solve with the least amount of computational resources such as time and space. • Generality: Algorithm should be valid on all possible inputs • Input/Output: There must be a specified number of input values, and one or more result values.

Question sources

Computer Science MCQ practice collection · Question 389

Question 76 | AM0390

What does the term "worst-case analysis" refer to in computational complexity?

A. Analyzing the best-case performance of an algorithm

B. Analyzing the average-case performance of an algorithm

C. Analyzing the worst possible input for an algorithm

D. Analyzing the expected performance of an algorithm

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: Analyzing the worst possible input for an algorithm

Worst-case analysis considers the worst possible input among inputs of a fixed size, giving a bound covering all their runtimes (PDF p.14). C describes that selection; average and expected costs answer different questions.

Why each choice is right or wrong

A - Incorrect: Best case considers the fastest rather than slowest input (p.14).

B - Incorrect: Average case uses an average input rather than the maximizing one (p.14).

C - Correct: The worst possible input is precisely the p.14 definition.

D - Incorrect: Expected performance requires averaging rather than selecting the slowest input (p.14).

Primary note proof

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 3-9

> In order to determine the running time of an algorithm it is possible to define three functions Tbest(n), Tavg(n) and Tworst(n) as the best, the average and the worst case running time of the algorithm respectively. Average Case (Tavg): The amount of time the algorithm takes on an "average" set of inputs. Worst Case (Tworst): The amount of time the algorithm takes on the worst possible set of inputs. Best Case (Tbest): The amount of time the algorithm takes on the

smallest possible set of inputs. We are interested in the worst-case time, since it provides a bound for all input - this is called the —Big-Oh‖ estimate.

Question sources

Computer Science MCQ practice collection · Question 390

Question 77 | AM0391

What is the significance of logarithmic time complexity, such as O(log n)?

A. It indicates linear time complexity.

B. It indicates constant time complexity.

C. It indicates exponential time complexity.

D. It indicates sublinear time complexity.

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: It indicates sublinear time complexity.

For increasing positive n, log n grows more slowly than n: repeated halving reaches unit size in only log_2 n stages, as illustrated by binary search (Data Structures and Algorithms PDF p.3). Assigned PDF p.14 separately lists logarithmic and linear classes. Therefore logarithmic growth is sublinear.

Why each choice is right or wrong

A - Incorrect: Linear is proportional to n, faster than log n (cross-module p.3 search comparison).

B - Incorrect: Logarithms remain unbounded, so they are not constant (assigned p.14).

C - Incorrect: Exponential growth is represented by 2^n, not log n (assigned p.14).

D - Correct: Log n grows slower than n, so it is sublinear (cross-module p.3 search comparison).

Primary note proof

Study-note section DAA05

Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the

size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching • Works on ordered/sorted list • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list • The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching • Works with unsorted List • Loop through the list starting at the first element • Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science MCQ practice collection · Question 391

Question 78 | AM0392

What does the term "P versus NP problem" refer to in computational complexity theory?

A. The problem of distinguishing between parallel and non-parallel algorithms.

B. The problem of determining the runtime of algorithms.

C. The problem of comparing the efficiency of programming languages.

D. The problem of determining if certain types of problems have efficient algorithms.

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: Whether efficiently verifiable decision problems also have efficient solving algorithms.

DAA44 defines P by deterministic polynomial-time decision solving and NP by polynomial-time certificate verification, equivalently nondeterministic polynomial solving. The P-versus-NP problem asks whether these classes coincide. Option D captures the question of efficient algorithms for this particular class of problems, although its wording is broad. It does not concern parallelism, language comparisons, or timing one program.

Why each choice is right or wrong

A - Incorrect: P and NP refer to polynomial solving and verification classes, not parallel versus non-parallel execution.

B - Incorrect: The question concerns equality of decision-problem classes, not merely determining a particular algorithm runtime.

C - Incorrect: Programming-language efficiency is unrelated to whether polynomial-time solvability and verifiability coincide.

D - Correct: This broadly expresses whether NP problems have deterministic polynomial-time solving algorithms, the P-versus-NP question.

Primary note proof

Study-note section DAA44

> P contains decision problems solvable deterministically in polynomial input-bit time. NP contains decision problems with polynomial-length certificates verifiable in polynomial time, equivalently solvable by a nondeterministic polynomial-time machine. NP does not mean non-polynomial; P⊆NP. Whether P=NP remains open.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 30 | lines 10-14

> Greedy algorithms sometimes fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the travelling salesman problem. Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

> In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Question sources

Computer Science MCQ practice collection · Question 392

Question 79 | AM0393

Which complexity class includes problems that can be solved by a non-deterministic Turing machine in polynomial time?

A. P

B. NP

C. NP-complete

D. Exponential

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer B: NP

DAA44 identifies NP with decision problems solvable by a nondeterministic polynomial-time machine. P is a contained class defined using deterministic machines; NP-complete adds hardness, which the stem never requires. Exponential is a time-growth description rather than the requested definition. The alternate set replaces Exponential with NP-hard, which likewise does not change the defining NP answer.

Why each choice is right or wrong

A - Incorrect: P is defined through deterministic polynomial time; it is contained in NP but does not name the full defined class.

B - Correct: NP exactly names decision problems solvable by a nondeterministic polynomial-time Turing machine.

C - Incorrect: NP-complete requires both NP membership and NP-hardness; hardness is absent from the stem.

D - Incorrect: Exponential describes time growth, not the class defined by nondeterministic polynomial-time computation.

Primary note proof

Study-note section DAA44

> P contains decision problems solvable deterministically in polynomial input-bit time. NP contains decision problems with polynomial-length certificates verifiable in polynomial time, equivalently solvable by a nondeterministic polynomial-time machine. NP does not mean non-polynomial; P⊆NP. Whether P=NP remains open.

Study-note section DAA44

> A polynomial many-one reduction A≤p B transforms each A-instance into a B-instance preserving yes/no. To prove B NP-hard, reduce a known hard A to B, not the reverse.
> NP-complete means NP-hard and in NP; optimization TSP is normally called NP-hard, while its bounded-tour decision form is NP-complete. A valid proposed tour and its cost are easy to check,

but certifying global optimality is another task. Pseudo-polynomial O(nW) knapsack is not polynomial in the bit length of W.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 30 | lines 10-14

Greedy algorithms sometimes fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the travelling salesman problem. Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Variant 1 | Alternative version 1

Which complexity class includes problems that can be solved by a non-deterministic Turing machine in polynomial time?

A. P

B. NP

C. NP-complete

D. NP-hard

Answer B: NP

The variant retains P, NP, and NP-complete and replaces the fourth option with NP-hard. DAA44 still defines NP by nondeterministic polynomial-time computation. NP-hard is a reduction-based hardness property and need not imply NP membership, so the answer remains NP after independently examining this set.

Why each choice is right or wrong

A - Incorrect: P specifies deterministic polynomial solving, a subclass rather than the full nondeterministic polynomial class.

B - Correct: NP is exactly the class defined by nondeterministic polynomial-time decision computation.

C - Incorrect: NP-complete additionally requires NP-hardness; the stem imposes no such condition.

D - Incorrect: NP-hardness alone does not imply membership in NP or a nondeterministic polynomial-time algorithm.

Primary note proof

Study-note section DAA44

> P contains decision problems solvable deterministically in polynomial input-bit time. NP contains decision problems with polynomial-length certificates verifiable in polynomial time, equivalently solvable by a nondeterministic polynomial-time machine. NP does not mean non-polynomial; P⊆NP. Whether P=NP remains open.

Study-note section DAA44

> A polynomial many-one reduction A≤p B transforms each A-instance into a B-instance preserving yes/no. To prove B NP-hard, reduce a known hard A to B, not the reverse. NP-complete means NP-hard and in NP; optimization TSP is normally called NP-hard, while its bounded-tour decision form is NP-complete. A valid proposed tour and its cost are easy to check, but certifying global optimality is another task. Pseudo-polynomial O(nW) knapsack is not polynomial in the bit length of W.

Question sources

Computer Science MCQ practice collection · Question 393

Computer Science MCQ practice collection · Question 750

Question 80 | CS0114

For p=10 and n=5, execute if(n>p){print(n+p);}. Treat the comparison, branch and possible output as executable constant-cost operations. Which assessment is correct without adding an unstated output requirement?

A. Feasibility is violated.

B. Finiteness is violated.

C. Sequential control flow is violated.

D. No violation is demonstrated by this fragment.

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: No violation is demonstrated by this fragment.

The condition 5>10 is false, so the print is skipped and the fragment terminates after constant work. Its operations are executable and its conditional control flow is legitimate. Nothing in this specified fragment violates feasibility or finiteness, and an unrequired output is not a correctness obligation. Thus the accurate assessment is that no property violation is demonstrated by this fragment.

Why each choice is right or wrong

A - Incorrect: The specified operations are executable, so feasibility is satisfied.

B - Incorrect: The fragment terminates, so finiteness is satisfied.

C - Incorrect: A conditional branch is legitimate algorithmic control flow.

D - Correct: The false branch skips output and terminates without the claimed property violations.

Primary note proof

Study-note section DAA45

> For p=10,n=5, if(n>p) cout<<(n+p) skips output and terminates. Skipping this output does not by itself violate feasibility, finiteness or control flow. Efficiency or a missing required output needs a specification.

Study-note section DAA01

> State inputs, output, constraints and empty-solution policy. Algorithms are finite, definite executable methods independent of source language; feasible satisfies constraints, optimal is best feasible. Conditional branches are legitimate control flow.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 4 | lines 6-19

> • Finiteness: Algorithm must complete after a finite number of steps. • Definiteness: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next. • Sequence: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted. • Feasibility: It must be possible to perform each instruction. • Correctness: It must compute correct answer for all possible legal inputs. • Language Independence: It must not depend on any one programming language. • Completeness: It must solve the problem completely. • Effectiveness: It must be possible to perform each step exactly and in a finite amount of time. • Efficiency: It must solve with the least amount of computational resources such as time and space. • Generality: Algorithm should be valid on all possible inputs • Input/Output: There must be a specified number of input values, and one or more result values.

Question sources

Computer Science Exit Exam practice collection · Question 114

Question 81 | CS0115

Assume it is given that f(n) = 4n^2, g(n) = 8n. Which asymptotic notation is correct?

A. f(n) = Big-Oh(g(n))

B. f(n) = Little-Oh(g(n))

C. f(n) = Big-Omega(g(n))

D. f(n) = Theta(g(n))

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: f(n) = Big-Omega(g(n))

Compare f/g=(4n^2)/(8n)=n/2, which grows without bound. Thus f eventually exceeds a constant multiple of g, establishing Omega(g) and excluding O(g), little-o(g) and Theta(g). The assigned PDF p.14 introduces growth comparison; the precise upper/lower/tight definitions are in Data Structures and Algorithms PDF p.3. For example n>=2 gives f>=g.

Why each choice is right or wrong

A - Incorrect: O(g) requires an eventual constant upper multiple; n/2 is unbounded, so it fails (cross-module p.3).

B - Incorrect: Little-o(g) requires slower growth; the ratio instead grows without bound (cross-module p.3).

C - Correct: Omega(g) is a lower growth bound; f>=g for every n>=2 establishes it (cross-module p.3).

D - Incorrect: Theta(g) requires the same growth rate; quadratic and linear differ by unbounded n/2 (cross-module p.3).

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \le c \cdot g(n)$ for all $n \ge n_0$, with fixed $c>0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA05

> For $f=4n^2$ and $g=8n$, $f/g=n/2 \to \infty$. Therefore $f \in \Omega(g)$ and even $\omega(g)$, while f is neither $O(g)$ nor $\Theta(g)$.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 14 | lines 11-21

In analysis of algorithms, it is not important to know how many operations an algorithm does. Of greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to us the rate of growth of the algorithm. What happens with small sets of input data is not as interesting as what happens when the data set gets large. The rate of growth of an algorithm is dominated by the largest term in an equation; we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the order of the function or related algorithm. We then group algorithms based on their order. We group in three categories - those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster. The following are some of the classes of algorithms: - log2 n, n, n ∗ log2 n, n2,n3, 2n. Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

Data Structure and Algorithms module | PDF page 3 | lines 2-5

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:

Data Structure and Algorithms module | PDF page 3 | lines 6-12

II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n). III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth. IV. Little-o Notation (o): f(n)=o(g(n)) means f(n) has less growth rate compared to g(n). V. Little-Omega Notation (ω): ω notation used to denote a lower bound that is not asymptotically tight.

Question sources

Computer Science Exit Exam practice collection · Question 115

Question 82 | CS0116

For a connected weighted undirected graph, which statements about its minimum spanning trees are true? Select all that apply; weights may tie.

A. Minimum spanning tree contains all vertices from an input graph

B. We can have only one unique minimum spanning tree for a graph

C. Minimum spanning trees are graphs

D. Minimum spanning trees form a cycle

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer A, C: Minimum spanning tree contains all vertices from an input graph; Minimum spanning trees are graphs

An MST contains every vertex and is a tree, hence is itself a graph. It is acyclic and has V−1 edges. Equal edge weights may permit multiple different MSTs, so uniqueness is not guaranteed. These properties make A and C true, while B's universal uniqueness claim and D's cycle claim are false. Distinct edge weights would be a sufficient additional uniqueness condition.

Why each choice is right or wrong

A - Correct: Spanning includes every input vertex.

B - Incorrect: Tied weights may yield multiple MSTs.

C - Correct: A tree is a particular kind of graph.

D - Incorrect: An MST is acyclic.

Primary note proof

Study-note section DAA23

> For a connected weighted undirected graph, an MST spans every vertex without cycles and minimizes total edge weight; it has V−1 edges. Negative edges are allowed. Equal weights may yield multiple MSTs; distinct weights suffice for uniqueness. A tree is itself a graph.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Design and Analysis of Algorithms module | PDF page 36 | lines 16-33

> o Minimum Cost Spanning Tree- Prim's Algorithm ✓ Let G(V,E) be an undirected connected graph with vertices v and edge E . ✓ A sub-graph t=(V,E) of the G is a Spanning tree of G iff t is a tree.3 ✓ The problem is to generate a graph G = (V,E) where E is the subset of E,G is a Minimum spanning tree. ✓ Each and every edge will contain the given non-negative length. connect all the nodes with edge present in set E and weight has to be minimum. NOTE: ✓ We have to visit all the nodes. ✓ The subset tree (i.e) any connected graph with N vertices must have at least N-1 edges and also it does not form a cycle. Definition: 1. A spanning tree of a graph is an undirected tree consisting of only those edge that are necessary to connect all the vertices in the original graph. 2. A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Design and Analysis of Algorithms module | PDF page 85 | lines 14-20

> 4.7. BASIC SEARCH AND TRAVERSAL TECHNIQUE DEFINING GRAPH: A graphs g consists of a set V of vertices (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges. Therefore, V(G) is read as V of G, is a set of vertices and E(G),read as E of G is a set of edges. An edge e= (v, w) is a pair of vertices v and w, and to be incident with v and w.

Question sources

Computer Science Exit Exam practice collection · Question 116

Question 83 | NS0294

At a fixed input size n, what is the minimum running time over all legal inputs of that size called?

A. Average time

B. Small time

C. Worst time

D. Best time

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: Best time

Best-case time is the minimum cost among legal inputs of a fixed size. Worst case takes the maximum, and average case takes an expectation under a specified distribution. Choosing a smaller input size would ask a different question and would not define best case. The revised wording explicitly holds n fixed, making best time the unique standard case term.

Why each choice is right or wrong

A - Incorrect: Average time takes an expectation rather than a minimum.

B - Incorrect: Small time is not the defined input-case category.

C - Incorrect: Worst time takes the maximum at fixed size.

D - Correct: Best time takes the minimum at fixed size.

Primary note proof

Study-note section DAA03

> At fixed size n, best case minimizes cost, worst case maximizes it, and average case takes an expectation under a specified input distribution. Randomized expected cost may average internal choices for one fixed input. Smallest input size is not best case.

Question sources

Computer Science Exit Exam · 2018 · Question 11

Question 84 | NS0314

A researcher solves mergesort's recurrence $T(n)=2T(n/2)+\Theta(n)$, obtaining $\Theta(n \log n)$, and claims that this calculation alone proves every comparison-based sorting algorithm requires $\Omega(n \log n)$ worst-case time. What is the flaw in that reasoning?

A. The proof fails because comparison decision trees allow only array indexing.

B. The recurrence describes one algorithm; a separate argument is needed to lower-bound the whole comparison class.

C. The proof fails only because Θ notation ignores constant factors.

D. There is no flaw: any upper bound for one comparison sort automatically lower-bounds all comparison sorts.

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer B: B. One algorithm's recurrence is not a universal lower-bound argument.

Solving the given recurrence establishes mergesort's $\Theta(n \log n)$ running time, but it does not rule out a different comparison algorithm with less work. The unsupported step is extending one algorithm's recurrence to a lower bound for the whole class. A separate decision-tree argument supplies the universal result: $n!$ possible orders require height at least $\log_2(n!)$. Discarding constants and permitting array indexing do not invalidate that argument.

Why each choice is right or wrong

A - Incorrect: The comparison model does not impose the stated array-indexing-only restriction.

B - Correct: The recurrence bounds mergesort; the $n!$-leaf decision-tree argument is what supplies a class-wide lower bound.

C - Incorrect: Ignoring constants is not the logical error; the unsupported extension from one algorithm to every algorithm is.

D - Incorrect: An upper bound for one available algorithm does not exclude a faster different algorithm, so it is not automatically a universal lower bound.

Primary note proof

Study-note section DAA11

> A comparison decision tree needs at least n! leaves for distinct-key orderings. Height h allows at most 2^h leaves, so h≥log 2(n!)=Ω(n log n); merge sort matches this worst-case bound. Array indexing is not forbidden.

Study-note section DAA14

> Ordinary array merge sort has $T(n)=2T(n/2)+\Theta(n)=\Theta(n \log n)$ across input orders, $\Theta(n)$ auxiliary buffer and $\Theta(\log n)$ recursion depth. A written $+O(n)$ recurrence alone establishes an upper bound; the tight result also uses linear merge work. Empty/singleton subarrays are base cases.

Question sources

Computer Science Exit Exam · 2018 · Question 38

Question 85 | NS0317

A routine sums the integers from n down to 1. For integer n≥0, use mathematical-integer RAM pseudocode with exact unit-cost arithmetic and no overflow. Charge 1 unit for each initialization, comparison, compound decrement i-- and return; charge 1 for addition and 1 for assignment in sum=sum+i. Ignore declarations, parameter passing and function-call overhead. What is T(n)? total(n) { sum=0; for(i=n;i>0;i--) sum=sum+i; return sum; }

A. 5n+5

B. $4n^2+4$

C. 6n+2

D. 4n+4

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer D: 4n+4

The descending loop executes n times. Initializing sum and i costs 2 units; i>0 is tested n+1 times; the compound decrement costs n units; and adding i then assigning the result costs 2n. One return gives T(n)=2+(n+1)+n+2n+1=4n+4. Although the returned sum grows quadratically with n, the number of unit-cost operations grows linearly. Exact mathematical integers prevent accumulator or loop-control overflow.

Why each choice is right or wrong

A - Incorrect: This overcounts the specified model by one unit per iteration and one fixed unit.

B - Incorrect: The numeric sum grows quadratically, but there are only n constant-cost loop iterations.

C - Incorrect: The stated iteration has one test, one compound decrement and two accumulator operations, not six units of iteration work.

D - Correct: Two initializations, n+1 tests, n decrements, 2n body units and one return total 4n+4.

Primary note proof

Study-note section DAA06

> Exact coefficients depend on whether tests, increments, arithmetic and assignments count separately.

Study-note section DAA06

For sum=0; for(i=1;i<=n;i++) sum=sum+1; return sum: counting two initializations, n+1 tests, n increments,2n body operations and one return gives 4n+4. A different primitive-cost convention changes this exact expression, not Θ(n).

Study-note section DAA02

A RAM model commonly gives constant cost to bounded-word arithmetic/access.

Question sources

Computer Science Exit Exam · 2018 · Question 42

Question 86 | NS0468

Which strategy breaks a problem into smaller largely independent subinstances, solves them recursively and combines their results, without caching repeated overlapping states?

A. Greedy

B. Backtracking

C. Divide and Conquer

D. Dynamic Programming

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer C: Divide and Conquer

Divide and conquer solves smaller largely independent instances and combines their results into the original solution. Dynamic programming can also recurse and combine values, but its distinguishing mechanism is reusable state values when subproblems overlap. Greedy is defined by safe irrevocable local choices, and backtracking by revising partial candidates. Independence and the lack of overlap-based caching resolve the original broad decomposition wording.

Why each choice is right or wrong

A - Incorrect: Greedy is defined by safe local choices.

B - Incorrect: Backtracking extends and retracts candidates.

C - Correct: Independent recursive subinstances and combination define divide and conquer.

D - Incorrect: The specified independence excludes the overlapping-state reuse emphasized by DP.

Primary note proof

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results. DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it. Backtracking grows/retracts candidates when constraints fail; branch and bound also rejects candidates whose best possible objective cannot beat an incumbent.

Question sources

Algorithm Design & Analysis review questions · Question 1

Question 87 | NS0469

What is the formal definition of Big-O notation? State the roles of constants c and k.

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: $f \in O(g)$ iff $\exists c>0, k$ such that $0 \le f(n) \le c\, g(n)$ for all $n \ge k$.

DAA04 requires one fixed positive multiplier and one threshold covering the entire tail of the eventually nonnegative functions. c permits constant-factor scaling; k excludes a finite initial prefix. Neither may be chosen anew for each n. This defines an eventual upper bound, independently of input case or whether the resource is time or space.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \le c \cdot g(n)$ for all $n \ge n_0$, with fixed $c>0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Question sources

Algorithm Design & Analysis review questions · Question 2

Question 88 | NS0470

Show that f(n) = 10n + 5 is O(n). What values of c and k satisfy the definition?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: c=15 and k=1 work.

For every $n \geq 1$, $5 \leq 5n$, so $10n+5 \leq 15n$. Choosing $c=15$ and $k=1$ therefore satisfies the Big-O upper inequality for the whole tail. DAA04 gives this exact worked example. These witnesses need not be unique; the goal is one valid fixed constant and threshold, not the smallest pair.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, with fixed $c>0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA04

> $10n+5 \leq 15n$ for $n \geq 1$, so $c=15, n_0=1$ works.

Question sources

Algorithm Design & Analysis review questions · Question 3

Question 89 | NS0471

Show that $f(n) = 3n^2 + 4n + 1$ is $O(n^2)$. What are the constants?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: c=8 and k=1 work.

For $n \geq 1$, replace $4n$ by at most $4n^2$ and 1 by at most n^2. The sum is then at most $8n^2$, satisfying DAA04's upper-bound definition. Thus $c=8$ and $k=1$ are valid witnesses. They are not uniquely required constants; larger multipliers or thresholds can also satisfy the inequality.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, with fixed $c>0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA04

> Also $3n^2+4n+1 \leq 8n^2$ for $n \geq 1$.

Question sources

Algorithm Design & Analysis review questions · Question 4

Question 90 | NS0472

Rank these complexity classes from slowest to fastest growing: O(n!), O(n^2), O(log n), O(2^n), O(n log n)

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Increasing representative growth: log n, n log n, n^2, 2^n, n!.

DAA05 orders logarithmic, polynomial, exponential, and factorial rates. A log factor above n is still dominated by n^2; fixed-base exponential eventually exceeds any fixed polynomial, and factorial exceeds fixed-base exponential. This ranks representative functions, not mutually exclusive Big-O sets, since a slower function can also satisfy a looser larger upper bound.

Primary note proof

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Question sources

Algorithm Design & Analysis review questions · Question 5

Question 91 | NS0473

Big-Theta Θ(h(n)) provides what kind of bound on an algorithm's complexity?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: A tight asymptotic bound: both upper and lower constant-factor bounds.

DAA04 requires positive constants c1,c2 and a threshold k with c1 h(n)≤f(n)≤c2 h(n) for all n≥k. Theta therefore specifies growth up to constant factors, not exact seconds. It can describe best, average, worst, or space functions; choosing Theta does not itself choose an input case.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, f∈O(g) means f(n)≤c·g(n) for all n≥n 0, with fixed c>0. Ω reverses that inequality; Θ requires both bounds. Little-o means f/g→0, and little-ω means f/g→∞ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Question sources

Algorithm Design & Analysis review questions · Question 6

Question 92 | NS0474

What does Big-Omega Ω(g(n)) represent and how does it differ from Big-O?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Ω is an eventual lower bound; O is an eventual upper bound.

DAA04 gives $f \in \Omega(g)$ when $f(n) \geq c\, g(n)$ beyond a fixed threshold for some $c>0$, reversing the Big-O inequality. The relations describe functions independently of best or worst case. A function may satisfy both, giving Theta, or only one for a particular comparator; they are not literal fastest-versus-slowest input labels.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f \in O(g)$ means $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, with fixed $c>0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g \to 0$, and little-ω means $f/g \to \infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Question sources

Algorithm Design & Analysis review questions · Question 7

Question 93 | NS0475

State the Transitivity Theorem of Big-O and explain why it is useful.

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: If f∈O(g) and g∈O(h), then f∈O(h).

DAA04 permits chained upper bounds. If f≤c g past k1 and g≤d h past k2, then f≤cd h for every n beyond max(k1,k2). Multiplying constants and combining thresholds proves transitivity. It is useful for replacing a complicated intermediate expression by a simpler known bound without redoing the entire comparison.

Primary note proof

Study-note section DAA04

> Prove the inequality for the whole tail, not selected values. Transitivity allows chained upper bounds; sums are dominated by the largest nonnegative growth term.

Study-note section DAA04

> For eventually nonnegative f, g, f∈O(g) means f(n)≤c·g(n) for all n≥n 0, with fixed c>0. Ω reverses that inequality; Θ requires both bounds. Little-o means f/g→0, and little-ω means f/g→∞ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Question sources

Algorithm Design & Analysis review questions · Question 8

Question 94 | NS0476

According to Big-O Theorem 6, order the following correctly: n!, 2^n, n^2, n log n, n, log n, constant k

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Increasing growth: k, log n, n, n log n, n^2, 2^n, n!.

For fixed positive k, DAA05 places constant growth first, followed by logarithmic and linear rates, then n log n, quadratic, exponential, and factorial growth. The ordering is eventual, not a promise of speed for every small input. It concerns representative functions; Big-O memberships can overlap when upper bounds are loose.

Primary note proof

Study-note section DAA05

> Representative growth is 1,log n,n,n log n,n^2,n^3,...,b^n,n! for fixed b>1. Log bases differ by a constant. Both n log n and n^2 are superlinear; log n is sublinear. Polynomial growth has a fixed exponent in input size.

Question sources

Algorithm Design & Analysis review questions · Question 9

Question 95 | NS0477

How is algorithm validation different from algorithm analysis?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Validation checks required behavior; analysis measures resource costs.

DAA01 specifies legal inputs, required outputs, and correctness proof through invariants or recursive induction. DAA02 counts operations and storage as input size grows. A procedure can be fast yet incorrect, or correct yet expensive. Testing can reveal failures but finite tests do not prove correctness for all inputs, so resource analysis does not replace validation.

Primary note proof

Study-note section DAA01

> State inputs, output, constraints and empty-solution policy. Algorithms are finite, definite executable methods independent of source language; feasible satisfies constraints, optimal is best feasible. Conditional branches are legitimate control flow.

Study-note section DAA02

> Analysis counts operations and storage as functions of input size: array length n, graph vertices V/edges E, or the number of bits in integers. A RAM model commonly gives constant cost to bounded-word arithmetic/access. Arbitrarily large integer arithmetic and printing long outputs need their own costs. Elapsed measurements also depend on hardware, compiler, cache, I/O and competing load.

Question sources

Algorithm Design & Analysis review questions · Question 10

Question 96 | NS0478

What is the general recurrence equation for a divide-and-conquer algorithm?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: For equal-size splits, T(n)=aT(n/b)+f(n), with an explicit base case.

DAA10 uses a recursive subcalls of size n/b and f(n) nonrecursive divide/combine work, with a≥1 and b>1. Include a base such as T(1)=Θ(1). Unequal or decrementing subproblems require a more general sum of subproblem costs; the equal-split form is a common model, not every possible divide-and-conquer recurrence.

Primary note proof

Study-note section DAA10

> For T(n)=aT(n/b)+f(n), a≥1, b>1, let p=log_b a. Level i has a^i nodes of size n/b^i. Compare f with n^p: a polynomial gap below gives Θ(n^p); f=Θ(n^p) gives Θ(n^p log n); a polynomial gap above gives Θ(f(n)) if a f(n/b)≤c f(n) for fixed c<1. The third case needs this regularity condition.

Question sources

Algorithm Design & Analysis review questions · Question 11

Question 97 | NS0479

What is the time complexity of Merge Sort and how is it derived?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Ordinary array merge sort is Θ(n log n).

DAA14 gives $T(n)=2T(n/2)+\Theta(n)$: two half-size sorts and linear merging. There are logarithmically many levels, each totaling linear merge work, with linear total leaf work. Therefore the tight result is n log n across input orders, assuming the ordinary algorithm rather than a variant that skips already ordered merges.

Primary note proof

Study-note section DAA14

> Ordinary array merge sort has $T(n)=2T(n/2)+\Theta(n)=\Theta(n \log n)$ across input orders, $\Theta(n)$ auxiliary buffer and $\Theta(\log n)$ recursion depth. A written $+O(n)$ recurrence alone establishes an upper bound; the tight result also uses linear merge work. Empty/singleton subarrays are base cases.

Question sources

Algorithm Design & Analysis review questions · Question 12

Question 98 | NS0480

Explain the three steps of Merge Sort in order.

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Split into halves, recursively sort each half, then merge the sorted halves.

DAA14 stops at empty or singleton ranges. During combination it repeatedly takes the smaller remaining front item, then appends the unexhausted suffix. Left-first equal-key selection preserves stability. These steps separate division, recursive solution, and combination; the merge assumes each half is already sorted rather than that the original full array was sorted.

Primary note proof

Study-note section DAA14

> Split by indices, recursively sort each half, then repeatedly copy the smaller remaining front element into a temporary result. The merge invariant says the output prefix is sorted and contains the smallest elements already consumed. Append the unexhausted suffix. Equal keys must be taken from the left half first to preserve stability.

Question sources

Algorithm Design & Analysis review questions · Question 13

Question 99 | NS0481

What property makes Merge Sort a 'stable' sorting algorithm?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Take equal keys from the left half first while merging.

DAA14 explicitly requires left-first equal-key handling for stability. Recursive halves retain their internal equal-key order, and choosing a left equal before a right equal preserves their original cross-half order. Inductively the entire output retains the relative order of equal-key records. Choosing the right equal first can break that property.

Primary note proof

Study-note section DAA14

> Split by indices, recursively sort each half, then repeatedly copy the smaller remaining front element into a temporary result. The merge invariant says the output prefix is sorted and contains the smallest elements already consumed. Append the unexhausted suffix. Equal keys must be taken from the left half first to preserve stability.

Question sources

Algorithm Design & Analysis review questions · Question 14

Question 100 | NS0482

What is the average and worst-case time complexity of Quick Sort, and when does the worst case occur?

Hint

Match the defining mechanism and its assumptions to the required claim; keep resource bounds separate from correctness.

Answer: Expected average $\Theta(n \log n)$; worst $\Theta(n^2)$ from repeated highly unbalanced partitions.

DAA15 gives logarithmic partition depth when splits are balanced and expected n log n with random pivots. Repeated extreme pivots leave sizes $n-1, n-2, \ldots$, causing $n+(n-1)+\ldots$ work. Sorted input can trigger this with certain fixed endpoint pivots, but input order alone is not a universal worst-case characterization across partition rules.

Primary note proof

Study-note section DAA15

> Balanced splits give $\Theta(n \log n)$; repeatedly highly unbalanced splits give $\Theta(n^2)$. Random pivots give expected $\Theta(n \log n)$, not a deterministic guarantee. Median-of-three does not eliminate adversarial cases; three-way partitioning helps repeated keys. Standard quicksort is unstable, uses expected $\Theta(\log n)$ stack but worst $\Theta(n)$; recursing on the smaller side and iterating on the larger bounds stack by $O(\log n)$.

Question sources

Algorithm Design & Analysis review questions · Question 15

Question 101 | NS0483

Describe the partitioning step of Quick Sort using indices i and j.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Advance i over values below the pivot and j over values above; swap while i≤j, then advance both.

DAA15's two-pointer partition bounds its scans and ensures progress even on equal values. After pointers cross, recurse on the resulting smaller ranges. Hoare-style boundaries need not put the pivot in its final position, whereas Lomuto commonly does. The chosen return convention and recursion ranges must match; mixing them can miss elements or fail to terminate.

Primary note proof

Study-note section DAA15

> Choose a pivot, partition, then sort the resulting smaller ranges. In a two-pointer scheme, advance i while values are below pivot and j while above; when i≤j, swap and advance both. Bound the scans and ensure progress on equal values. Recurse on the resulting ranges. Hoare-style partition boundaries need not put the pivot at its final position; Lomuto-style partition commonly does. Do not mix their return conventions.

Question sources

Algorithm Design & Analysis review questions · Question 16

Question 102 | NS0484

Compare Merge Sort and Quick Sort on: stability, worst-case complexity, and space usage.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Merge: stable, Θ(n log n) worst, Θ(n) buffer. Quick: unstable, Θ(n^2) worst, expected Θ(log n) and worst Θ(n) stack.

DAA14 provides merge's linear buffer and logarithmic recursion depth, with left-first ties for stability. DAA15 gives quicksort's in-array partitioning but nonconstant recursive storage and potentially linear depth. Recursing only on the smaller side can bound that stack logarithmically. These describe ordinary implementations; ignoring recursive frames would misstate total auxiliary space.

Primary note proof

Study-note section DAA14

> Ordinary array merge sort has T(n)=2T(n/2)+Θ(n)=Θ(n log n) across input orders, Θ(n) auxiliary buffer and Θ(log n) recursion depth. A written +O(n) recurrence alone establishes an upper bound; the tight result also uses linear merge work. Empty/singleton subarrays are base cases.

Study-note section DAA15

> Balanced splits give Θ(n log n); repeatedly highly unbalanced splits give Θ(n^2). Random pivots give expected Θ(n log n), not a deterministic guarantee. Median-of-three does not eliminate adversarial cases; three-way partitioning helps repeated keys. Standard quicksort is unstable, uses expected Θ(log n) stack but worst Θ(n); recursing on the smaller side and iterating on the larger bounds stack by O(log n).

Study-note section DAA14

> Equal keys must be taken from the left half first to preserve stability.

Question sources

Algorithm Design & Analysis review questions · Question 17

Question 103 | NS0485

Why is Binary Search classified as a Divide and Conquer algorithm?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Each midpoint comparison reduces the ordered-search instance to one half.

DAA13 maintains sorted low/high bounds and excludes the half that cannot contain the target. The recurrence $T(n)=T(n/2)+\Theta(1)$ gives logarithmic worst work. Unlike sorting two halves, binary search solves only one surviving subinstance and requires no substantial combination step; this is the common divide-and-conquer or decrease-by-factor classification.

Primary note proof

Study-note section DAA13

> Binary search requires sorted random-access data. Maintain inclusive low/high bounds, compute mid=low+(high−low)//2, and replace high with mid−1 or low with mid+1 according to the comparison. Stop when found or low>high. $T(n)=T(n/2)+\Theta(1)$: worst $\Theta(\log n)$, best $\Theta(1)$; iterative auxiliary space $\Theta(1)$, recursive $\Theta(\log n)$.

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results.

Question sources

Algorithm Design & Analysis review questions · Question 18

Question 104 | NS0486

What is the core principle of the Greedy algorithmic paradigm?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Repeatedly make a locally attractive feasible choice and keep it.

DAA19 defines greedy decisions as irrevocable rather than exploring and undoing alternatives. To obtain a global optimum, the local choice needs a problem-specific safety argument, often an exchange proof. Optimal substructure alone is insufficient. Thus local attractiveness is the mechanism, while global optimality is a separate correctness obligation rather than an automatic consequence.

Primary note proof

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Question sources

Algorithm Design & Analysis review questions · Question 19

Question 105 | NS0487

What is the difference between a feasible solution and an optimal solution in the context of Greedy algorithms?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Feasible satisfies constraints; optimal is best among feasible solutions.

DAA01 separates constraint satisfaction from objective quality. For example, a legal set of knapsack items may fit capacity but have less value than another legal set. DAA19 warns that greedy choices require a safety proof to reach an optimum; constructing a feasible result alone does not prove global optimality.

Primary note proof

Study-note section DAA01

> State inputs, output, constraints and empty-solution policy. Algorithms are finite, definite executable methods independent of source language; feasible satisfies constraints, optimal is best feasible. Conditional branches are legitimate control flow.

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Question sources

Algorithm Design & Analysis review questions · Question 20

Question 106 | NS0488

Name two advantages and two disadvantages of Greedy algorithms.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Advantages: simple choices and low overhead. Disadvantages: irrevocable mistakes and problem-specific applicability.

DAA19 identifies these benefits and limitations. Greedy can avoid maintaining a large state table, but a locally attractive choice can prevent an optimum and cannot later be undone within the paradigm. An exchange or other safety proof is therefore essential. A greedy heuristic has no approximation guarantee solely because it uses local choices.

Primary note proof

Study-note section DAA19

> Advantages are often simple decisions and low overhead; limitations are irrevocable errors and problem-specific applicability. A heuristic has no approximation guarantee merely because it is greedy. For coins{1,3,4} and target 6, largest-first gives 4+1+1 while 3+3 uses fewer coins. Activity selection instead has a safe earliest-finish rule for maximizing the number of nonoverlapping unweighted intervals; replacing the first chosen activity in an optimum with an earlier-finishing one leaves at least as much room.

Question sources

Algorithm Design & Analysis review questions · Question 21

Question 107 | NS0489

In the Job Sequencing with Deadlines algorithm, why are jobs sorted in decreasing order of profit first?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Profit-first gives high-value feasible jobs priority under the unit-duration scheduling assumptions.

DAA21 processes nonnegative-profit jobs in decreasing profit and places each in its latest free slot meeting its deadline. This preserves earlier slots for tighter deadlines, while an exchange argument supports profit-first selection for feasible unit-job sets. The rule needs one machine, initially available unit jobs, and on-time profit; variable durations can invalidate it.

Primary note proof

Study-note section DAA21

> Assume one machine, initially available jobs, unit durations, integer deadlines and profit only for on-time completion. Process nonnegative-profit jobs by decreasing profit; place each in the latest free slot finishing by its deadline, or reject it. Latest placement preserves earlier slots for tighter deadlines; feasible unit-job sets support profit-first exchange.

Question sources

Algorithm Design & Analysis review questions · Question 22

Question 108 | NS0490

Given jobs J4(deadline=2, profit=300), J1(deadline=5, profit=200), J3(deadline=3, profit=190), J2(deadline=3, profit=180), J5(deadline=4, profit=120), J6(deadline=2, profit=100) - what is the optimal schedule and maximum profit?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Slots 1-5: J2,J4,J3,J5,J1; maximum profit 990.

DAA21's greedy placement gives J4 at slot2, J1 at5, J3 at3, J2 at1, and J5 at4; J6 cannot fit before deadline2. Every accepted deadline holds. Their profits sum to 180+300+190+120+200=990. Since these are the five largest profits and only five slots exist, 990 also matches an upper bound.

Primary note proof

Study-note section DAA21

> Assume one machine, initially available jobs, unit durations, integer deadlines and profit only for on-time completion. Process nonnegative-profit jobs by decreasing profit; place each in the latest free slot finishing by its deadline, or reject it. Latest placement preserves earlier slots for tighter deadlines; feasible unit-job sets support profit-first exchange.

Study-note section DAA21

> Slots finishing 1…5 contain J 2,J 4,J 3,J 5,J 1. Profits 180+300+190+120+200=990; deadlines hold. J 6 cannot finish by 2. The five largest profits sum to 990, giving a matching upper bound.

Question sources

Algorithm Design & Analysis review questions · Question 23

Question 109 | NS0491

What is the Optimal Merge Pattern and what greedy strategy does it apply?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Repeatedly merge the two smallest files or weights, reinsert their sum, and minimize total merge cost.

DAA22 charges p+q for combining file sizes p and q, adds that charge to total cost, and returns the merged size to a min-heap. Combining small weights early avoids repeatedly charging large weights at deep levels. The min-heap implementation costs O(n log n); actual comparison counts are a distinct cost model.

Primary note proof

Study-note section DAA22

> Combine the two smallest weights, add their sum to total cost, and reinsert it. A min-heap gives O(n log n) time/O(n) space. Cost p+q counts moved records; worst-case comparisons are p+q−1 for nonempty lists, and actual comparisons may be fewer.

Question sources

Algorithm Design & Analysis review questions · Question 24

Question 110 | NS0492

What is a Minimum Spanning Tree (MST) and what problem does it solve?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: An MST connects all vertices without cycles using minimum total edge weight.

DAA23 assumes a connected undirected weighted graph and gives an MST V−1 edges. It minimizes total connection cost, not every source-to-vertex route. Prim and Kruskal find such trees through safe light edges. Equal weights can allow several optima, and disconnected input instead yields a minimum spanning forest rather than one spanning tree.

Primary note proof

Study-note section DAA23

> For a connected weighted undirected graph, an MST spans every vertex without cycles and minimizes total edge weight; it has V−1 edges. Negative edges are allowed. Equal weights may yield multiple MSTs; distinct weights suffice for uniqueness. A tree is itself a graph.

Question sources

Algorithm Design & Analysis review questions · Question 25

Question 111 | NS0493

Compare Prim's Algorithm and Kruskal's Algorithm: approach, edge selection, and best use case.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Prim grows one tree across a cut; Kruskal accepts globally light edges joining distinct components.

DAA23 gives matrix Prim $O(V^2)$, heap/list Prim $O(E \log V)$, and edge-sorting Kruskal $O(E \log E)$. Matrix Prim can favor dense graphs because it avoids sorting quadratically many edges; Kruskal naturally uses an edge list and union-find. Best use depends on representation and implementation, not a universal measured-speed ranking.

Primary note proof

Study-note section DAA23

> Prim repeatedly adds the cheapest edge crossing from its current tree to an outside vertex. Kruskal sorts edges globally, accepting one only when union-find says its endpoints are in different components. The cut proof exchanges a light crossing edge into some optimum without raising its weight. Matrix Prim costs $O(V^2)$; binary-heap Prim $O(E \log V)$, and Kruskal $O(E \log E)$. Dense-graph comparisons depend on these implementations. Disconnected input gives a minimum spanning forest.

Question sources

Algorithm Design & Analysis review questions · Question 26

Question 112 | NS0494

How does Kruskal's Algorithm detect and avoid cycles when building the MST?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Use union-find: reject an edge if its endpoints already belong to the same component.

DAA23 processes edges globally in increasing weight. If the endpoint representatives differ, the edge joins components without creating a cycle, so accept it and union the sets. If they match, a path already connects the endpoints and adding the edge would close a cycle. The light-edge exchange property supports minimum total weight.

Primary note proof

Study-note section DAA23

> Prim repeatedly adds the cheapest edge crossing from its current tree to an outside vertex. Kruskal sorts edges globally, accepting one only when union-find says its endpoints are in different components. The cut proof exchanges a light crossing edge into some optimum without raising its weight. Matrix Prim costs $O(V^2)$; binary-heap Prim O(E log V), and Kruskal O(E log E). Dense-graph comparisons depend on these implementations. Disconnected input gives a minimum spanning forest.

Question sources

Algorithm Design & Analysis review questions · Question 27

Question 113 | NS0495

What is Dijkstra's Algorithm used for and what is its key limitation?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Single-source shortest paths with nonnegative edge weights; negative edges break its general guarantee.

DAA24 finalizes the least unsettled tentative distance and relaxes outgoing edges. Nonnegative weights ensure a later route through unsettled vertices cannot improve a finalized distance. A negative edge can violate that proof even without a negative cycle. This differs from MST construction, and unreachable vertices retain infinite distance.

Primary note proof

Study-note section DAA24

> Nonnegative edges ensure a later path through unsettled vertices cannot improve a finalized distance. With $s \to a = 2$, $s \to b = 5$, $b \to a = -4$, finalizing a at 2 misses the later distance 1. Binary-heap/list implementation costs $O((V+E)\log V)$, matrix scanning $O(V^2)$. An MST minimizes total connection cost; Dijkstra minimizes each source-path distance, a different objective.

Question sources

Algorithm Design & Analysis review questions · Question 28

Question 114 | NS0496

List the steps of Dijkstra's Algorithm in order.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Initialize distances; repeatedly finalize the smallest unsettled distance and relax its outgoing edges.

DAA24 starts source at zero, all others at infinity, with predecessors unset. Extract the minimum unsettled vertex; update dist[v] and its predecessor when dist[u]+w(u,v) improves it. Stop when none remain or the minimum is infinite. For an early destination stop, finalize it first rather than stopping merely on discovery.

Primary note proof

Study-note section DAA24

> Initialize dist[source]=0, others∞ and predecessors unset. Repeatedly extract the unsettled vertex u of minimum distance; relax each edge by dist[v]=min(dist[v], dist[u]+w(u, v)), updating predecessors when improved. Once the minimum is∞, remaining vertices are unreachable. A destination may be finalized before stopping, not merely discovered.

Question sources

Algorithm Design & Analysis review questions · Question 29

Question 115 | NS0497

What is the difference between MST (Minimum Spanning Tree) and SSSP (Single-Source Shortest Path)?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: MST minimizes total connection weight; SSSP minimizes each source-to-vertex path distance.

DAA23's spanning tree connects every vertex with minimum sum of selected edge weights. DAA24's source shortest paths optimize a different objective for a designated source and can choose a different tree. An MST need not contain shortest source routes. Standard MST permits negative weights, while Dijkstra's particular SSSP algorithm needs nonnegative weights.

Primary note proof

Study-note section DAA23

> For a connected weighted undirected graph, an MST spans every vertex without cycles and minimizes total edge weight; it has V−1 edges. Negative edges are allowed. Equal weights may yield multiple MSTs; distinct weights suffice for uniqueness. A tree is itself a graph.

Study-note section DAA24

> Nonnegative edges ensure a later path through unsettled vertices cannot improve a finalized distance. With s→a=2, s→b=5, b→a=−4, finalizing a at 2 misses the later distance 1. Binary-heap/list implementation costs $O((V+E)\log V)$, matrix scanning $O(V^2)$. An MST minimizes total connection cost; Dijkstra minimizes each source-path distance, a different objective.

Question sources

Algorithm Design & Analysis review questions · Question 30

Question 116 | NS0498

Which two textbook structural properties justify and motivate the usual dynamic-programming approach to optimization? Explain their different roles rather than treating them as universal requirements for every use of DP.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Optimal substructure and overlapping subproblems: the first justifies composing optimal subsolutions, while the second motivates reusing stored state results.

Optimal substructure means an optimal solution can be built using appropriate optimal subsolutions. Overlap means the same state problem occurs repeatedly, making saved results useful. These properties justify and motivate the familiar optimization pattern, provided the state captures all relevant information and the recurrence is correct. DP can also count or decide, so the two properties are not stated here as universal necessities for every DP computation.

Primary note proof

Study-note section DAA25

> Overlapping states motivate reuse; optimal substructure supports optimization. These are textbook signatures, not requirements that every DP problem optimize or every state have multiple users. DP also counts/decides. Cost=states×work per state and can remain exponential.

Question sources

Algorithm Design & Analysis review questions · Question 31

Question 117 | NS0499

How does Dynamic Programming differ from Divide and Conquer?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Divide and conquer solves largely independent instances; DP reuses dependent or overlapping states.

DAA12 distinguishes independent recursive decomposition and combination from a defined set of reused states. DP requires state meaning, transitions, bases, and dependency order so each needed state is computed without redundant recomputation. Divide and conquer usually does not need a shared-state cache. Merely splitting a problem into smaller parts is common to both and does not distinguish them.

Primary note proof

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results. DP reuses a defined set of subproblem states. Greedy commits to a feasible local choice only when a safety argument supports it. Backtracking grows/retracts candidates when constraints fail; branch and bound also rejects candidates whose best possible objective cannot beat an incumbent.

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Question sources

Algorithm Design & Analysis review questions · Question 32

Question 118 | NS0500

How does Dynamic Programming differ from the Greedy approach?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Greedy commits to a proved-safe local choice; DP compares stored subproblem alternatives.

DAA19 makes greedy choices irrevocable and requires a safety argument. DAA25 defines states and recurrence choices, then computes optimal state values in dependency order. DP can retain alternatives that a greedy rule would discard. Optimal substructure does not itself make greedy safe, and neither approach is universally preferable without the problem's structural conditions.

Primary note proof

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Question sources

Algorithm Design & Analysis review questions · Question 33

Question 119 | NS0501

What are the four development steps of a Dynamic Programming algorithm?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Define and justify optimal states; derive recurrence and bases; compute dependencies once; reconstruct retained choices.

DAA25 first requires a state meaning and legal choices, then a correctness argument for the recurrence and feasible candidates. Evaluate states in dependency order, accounting for states times work per state. Store decisions so the chosen solution can be reconstructed afterward. These provide the requested four development stages without assuming DP is automatically polynomial.

Primary note proof

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Question sources

Algorithm Design & Analysis review questions · Question 34

Question 120 | NS0502

Explain memoization and tabulation - the two DP implementation strategies.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Memoization solves requested states recursively with a cache; tabulation fills states in explicit dependency order.

DAA26 checks the cache before recursively solving a memoized state and stores its result. Tabulation schedules prerequisite states first without that demand-driven recursion. Keys must include all relevant parameters, and chosen domains may avoid irrelevant states. Both reuse results, but sparsity, recursion overhead, and locality can affect which implementation is preferable.

Primary note proof

Study-note section DAA26

> Memoization recursively checks a cache before solving a state and stores its result. Tabulation orders dependencies explicitly; chosen domains may exclude irrelevant states. Cache keys include all relevant parameters; recursion/locality/sparsity affect implementation cost.

Question sources

Algorithm Design & Analysis review questions · Question 35

Question 121 | NS0503

In the Fibonacci sequence, why is naive recursion inefficient and how does DP improve it?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Naive recursion repeats Fibonacci states; DP computes each required state once.

DAA26 shows that F(n−2) reappears across recursion branches, causing Θ(φ^n) calls, with O(2^n) a valid loose bound. Memoization or tabulation uses the two preceding values and only Θ(n) unit-cost additions. Two rolling values reduce working machine words to constant, though arbitrary-precision values require growing bit costs.

Primary note proof

Study-note section DAA26

> Naive Fibonacci recomputes F(n−2) across branches: call count is Θ(φ^n), with O(2^n) a valid loose upper bound. F(0)=0, F(1)=1 and F(i)=F(i−1)+F(i−2) yield Θ(n) unit-cost additions with memoization/tabulation. Keeping only two recent values uses Θ(1) machine words; arbitrary-precision output needs growing bit storage. This naive overlapping recursion is not the independent-subproblem ideal of D&C.

Question sources

Algorithm Design & Analysis review questions · Question 36

Question 122 | NS0504

What is the 0/1 knapsack optimization problem with positive integer weights, nonnegative values and integer capacity W, and why does its item-and-capacity state formulation suit dynamic programming? Each item may be chosen at most once.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Choose a subset with total weight at most W and maximum total value. DP compares excluding each item with including it once and reuses item/capacity state results.

Define K[i,w] as the best value using the first i items within capacity w. An item that fits can be excluded or included with its value added to K[i-1,w-weight_i]; taking the larger value gives the recurrence. The previous-row reference enforces at-most-once use, and repeated states are reused. The O(nW) running time is pseudo-polynomial in binary-encoded capacity, unlike the separate fractional-knapsack greedy method.

Primary note proof

Study-note section DAA29

> For integer capacity W, K[i, w] is maximum value using the first i items within capacity w. Initialize K[0, w]=0. If weight_i>w, use K[i−1, w]; otherwise take max(K[i−1, w], value_i+K[i−1, w−weight_i]). The previous-row reference ensures each item is used at most once. Backtrack changed values to recover chosen items.

Study-note section DAA20

> This proof depends on divisibility. With capacity 10 and items(6,12),(5,8),(5,8), ratio-greedy chooses value 12; the two weight 5 items give 16 under 0/1 rules. Specify whether the knapsack problem is fractional,0/1 or unbounded. Feasibility alone does not guarantee optimality.

Question sources

Algorithm Design & Analysis review questions · Question 37

Question 123 | NS0505

What is the All Pairs Shortest Path (APSP) problem and which algorithm solves it?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: APSP finds shortest distances for every ordered vertex pair; Floyd-Warshall is one DP algorithm.

DAA28 initializes direct-edge costs and progressively allows each vertex as an intermediate. Its min-plus recurrence gives $\Theta(V^3)$ time and $\Theta(V^2)$ storage. Negative cycles make affected pair distances lack a finite minimum; Floyd detects them through negative diagonals. Its cost is a taught solution rather than a universal best-achievable APSP bound.

Primary note proof

Study-note section DAA28

> For all-pairs distances initialize D[i, i]=0, D[i, j] to the minimum direct edge weight, and missing edges to∞. Preserve a negative self-loop by taking min(0, its weight). For k outermost, update D[i, j]=min(D[i, j], D[i, k]+D[k, j]); before phase k, intermediates are restricted to earlier vertices. Guard unreachable sums and numeric overflow.

Study-note section DAA28

> Three loops give $\Theta(V^3)$ time and $\Theta(V^2)$ storage. A negative diagonal afterward identifies a negative cycle; pairs able to traverse it have no finite minimum walk distance. Warshall reachability replaces min/+ by OR/AND. Floyd's cost is not a universal best-achievable bound for all APSP settings; sparse nonnegative graphs permit repeated Dijkstra and other models permit other methods.

Question sources

Algorithm Design & Analysis review questions · Question 38

Question 124 | NS0506

Describe the APSP initialization step. What values fill the matrix initially?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Initialize D[i,i]=0, direct pairs to minimum edge weights, and missing edges to infinity.

DAA28 uses zero for the empty self-path, but preserves a negative self-loop by taking min(0, its weight). Parallel direct edges use their minimum. Infinity denotes no known route, so later arithmetic must guard unreachable sums and overflow. This is Floyd's initialization before intermediate vertices are considered, not an adjacency-only final answer.

Primary note proof

Study-note section DAA28

> For all-pairs distances initialize D[i, i]=0, D[i, j] to the minimum direct edge weight, and missing edges to∞. Preserve a negative self-loop by taking min(0, its weight). For k outermost, update D[i, j]=min(D[i, j], D[i, k]+D[k, j]); before phase k, intermediates are restricted to earlier vertices. Guard unreachable sums and numeric overflow.

Question sources

Algorithm Design & Analysis review questions · Question 39

Question 125 | NS0507

What is the DP update rule used in the APSP algorithm for each intermediate vertex k?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: D[i,j] ← min(D[i,j], D[i,k]+D[k,j]), with k outermost.

DAA28's phase invariant permits only earlier vertices as intermediates before phase k. Each pair then compares its current route with a route through k. Guard infinite operands and overflow rather than adding unreachable values blindly. Keeping k as the outer phase index is essential to the stated dependency proof.

Primary note proof

Study-note section DAA28

> For all-pairs distances initialize D[i, i]=0, D[i, j] to the minimum direct edge weight, and missing edges to∞. Preserve a negative self-loop by taking min(0, its weight). For k outermost, update D[i, j]=min(D[i, j], D[i, k]+D[k, j]); before phase k, intermediates are restricted to earlier vertices. Guard unreachable sums and numeric overflow.

Question sources

Algorithm Design & Analysis review questions · Question 40

Question 126 | NS0508

Which design strategy explores ALL possible solutions by incrementally building candidates and abandoning invalid ones?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Backtracking

DAA36 assigns the next choice, checks whether a feasible extension remains possible, recurses, and undoes the choice on return. A complete search reports every required candidate while pruning only impossible branches. Stop after one solution only if the task asks for existence or a single solution; otherwise early stopping would fail the ALL requirement.

Primary note proof

Study-note section DAA36

> Represent a partial assignment x[1.. k]. For each allowed next choice: assign it, test whether an extension remains possible, recurse if promising, and undo the assignment on return. Report complete feasible candidates; stop after one only if the task requests existence/a single answer. Explicit constraints restrict each variable’s domain; implicit constraints relate choices.

Question sources

Algorithm Design & Analysis review questions · Question 41

Question 127 | NS0509

Explain why Greedy algorithms are not guaranteed to give a globally optimal solution, with an example.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: A locally attractive choice can block the optimum; coins {1,3,4} for target 6 are a counterexample.

DAA19's largest-first choice produces 4+1+1, using three coins, while 3+3 uses two. Once the greedy method keeps 4, it cannot repair the earlier choice within its irrevocable policy. Therefore local feasibility and attractiveness do not prove global optimality; a problem-specific exchange or safety proof is needed.

Primary note proof

Study-note section DAA19

> Advantages are often simple decisions and low overhead; limitations are irrevocable errors and problem-specific applicability. A heuristic has no approximation guarantee merely because it is greedy. For coins{1,3,4} and target 6, largest-first gives 4+1+1 while 3+3 uses fewer coins. Activity selection instead has a safe earliest-finish rule for maximizing the number of nonoverlapping unweighted intervals; replacing the first chosen activity in an optimum with an earlier-finishing one leaves at least as much room.

Question sources

Algorithm Design & Analysis review questions · Question 42

Question 128 | NS0510

For ordinary array quicksort, what partition pattern causes quadratic worst-case time, and how can randomized pivot selection or three-way partitioning mitigate relevant workloads? Distinguish expected-time improvements from worst-case and stack-space guarantees.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Repeated very unbalanced partitions cause quadratic work. Random pivots give expected Theta(n log n) time; three-way partitioning helps repeated keys, and smaller-side recursion can limit stack space.

Extreme pivots can leave subproblems of sizes n-1,n-2 and so on, giving quadratic total partition work. Random pivots improve expected time to Theta(n log n) but do not remove quadratic possible executions. Three-way partitioning helps repeated keys, while median-of-three still admits adversarial inputs. Recursing only on the smaller side and iterating on the larger separately bounds stack depth by O(log n). No universal rarity claim is needed.

Primary note proof

Study-note section DAA15

> Balanced splits give $\Theta(n \log n)$; repeatedly highly unbalanced splits give $\Theta(n^2)$. Random pivots give expected $\Theta(n \log n)$, not a deterministic guarantee. Median-of-three does not eliminate adversarial cases; three-way partitioning helps repeated keys. Standard quicksort is unstable, uses expected $\Theta(\log n)$ stack but worst $\Theta(n)$; recursing on the smaller side and iterating on the larger bounds stack by $O(\log n)$.

Question sources

Algorithm Design & Analysis review questions · Question 43

Question 129 | NS0511

For the Job Sequencing algorithm, if two jobs have the same profit, how should ties be broken?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Equal-profit jobs may be processed in any order under the stated unit-job assumptions.

DAA21 explicitly permits arbitrary equal-profit ordering; earlier deadline is an optional tie convention rather than a correctness requirement. Each accepted job still goes into its latest free slot finishing by its deadline. The freedom relies on the one-machine, initially available unit-duration model and does not automatically extend to variable durations or release times.

Primary note proof

Study-note section DAA21

> Sorting plus slot scans costs O(n log n+nD), with D capped at n. Equal-profit order is arbitrary; earlier deadline is optional. Variable durations/release times invalidate this rule.

Question sources

Algorithm Design & Analysis review questions · Question 44

Question 130 | NS0512

Using the note's worst-case bounds, compare matrix-based Prim with Kruskal that comparison-sorts all edges on a dense simple undirected graph, where E=Theta(V^2). Why can this implementation pairing favor Prim asymptotically?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Matrix Prim has O(V^2) time; edge-sorting Kruskal has O(E log E)=O(V^2 log V). The matrix implementation avoids the all-edge sorting factor.

The matrix-based Prim implementation scans vertex connections in O(V^2) time. Kruskal's stated bound includes sorting all E edges, giving O(E log E). For dense simple graphs E has order V^2, so that expression becomes O(V^2 log V), whereas matrix Prim retains its quadratic bound. This explains the advantage of the specified implementations and does not claim that every version of Prim is always faster in measured runtime.

Primary note proof

Study-note section DAA23

> Prim repeatedly adds the cheapest edge crossing from its current tree to an outside vertex. Kruskal sorts edges globally, accepting one only when union-find says its endpoints are in different components. The cut proof exchanges a light crossing edge into some optimum without raising its weight. Matrix Prim costs O(V^2); binary-heap Prim O(E log V), and Kruskal O(E log E). Dense-graph comparisons depend on these implementations. Disconnected input gives a minimum spanning forest.

Question sources

Algorithm Design & Analysis review questions · Question 45

Question 131 | NS0513

Can Dijkstra's Algorithm be applied to a graph with negative edge weights? Why or why not?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Standard Dijkstra is not generally correct with negative edges.

DAA24's example $s \to a=2$, $s \to b=5$, $b \to a=-4$ finalizes a at 2 but a later route improves it to 1. That violates the nonnegative-edge finalization proof even without a negative cycle. Some instances may happen to work, but no general guarantee remains. Bellman-Ford or DAG DP supplies an appropriate alternative under its assumptions.

Primary note proof

Study-note section DAA24

> Nonnegative edges ensure a later path through unsettled vertices cannot improve a finalized distance. With $s \to a=2$, $s \to b=5$, $b \to a=-4$, finalizing a at 2 misses the later distance 1. Binary-heap/list implementation costs $O((V+E)\log V)$, matrix scanning $O(V^2)$. An MST minimizes total connection cost; Dijkstra minimizes each source-path distance, a different objective.

Study-note section DAA27

> For a general graph, Bellman-Ford relaxes all edges for up to $V-1$ passes and tests an additional pass for a reachable negative cycle: $O(VE)$. Thus "single-source shortest path" does not uniquely imply greedy; graph assumptions determine the algorithm.

Question sources

Algorithm Design & Analysis review questions · Question 46

Question 132 | NS0514

What is the time complexity of APSP using the Floyd-Warshall approach and why?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: $\Theta(V^3)$ time and $\Theta(V^2)$ distance storage.

DAA28 runs one intermediate-vertex phase for every k and examines every ordered pair i,j during each phase. V phases times V^2 constant-cost relaxations gives cubic time. The matrix has V^2 entries. These bounds assume bounded-cost arithmetic and do not claim that Floyd is universally optimal among all APSP algorithms or models.

Primary note proof

Study-note section DAA28

> For all-pairs distances initialize D[i, i]=0, D[i, j] to the minimum direct edge weight, and missing edges to∞. Preserve a negative self-loop by taking min(0, its weight). For k outermost, update D[i, j]=min(D[i, j], D[i, k]+D[k, j]); before phase k, intermediates are restricted to earlier vertices. Guard unreachable sums and numeric overflow.

Question sources

Algorithm Design & Analysis review questions · Question 47

Question 133 | NS0515

Explain the difference between 'Divide and Conquer' and 'Dynamic Programming' using the Fibonacci problem.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Naive Fibonacci has overlapping recursive calls; DP reuses them instead of treating them as independent.

DAA26 explicitly says naive Fibonacci is not the independent-subproblem ideal of divide and conquer. Both branches recompute states such as F(n−2), causing exponential calls. A cache or increasing-index table computes each state once, giving linear unit-cost additions. Recursion alone therefore does not establish a well-structured independent divide-and-conquer algorithm.

Primary note proof

Study-note section DAA26

> Naive Fibonacci recomputes F(n−2) across branches: call count is Θ(φ^n), with O(2^n) a valid loose upper bound. F(0)=0, F(1)=1 and F(i)=F(i−1)+F(i−2) yield Θ(n) unit-cost additions with memoization/tabulation. Keeping only two recent values uses Θ(1) machine words; arbitrary-precision output needs growing bit storage. This naive overlapping recursion is not the independent-subproblem ideal of D&C.

Study-note section DAA12

> Divide and conquer solves smaller largely independent instances and combines results.

Question sources

Algorithm Design & Analysis review questions · Question 48

Question 134 | NS0516

What does the recurrence T(n) = 2T(n/2) + O(n) represent and what is its solution?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Two half-size recursive calls plus at most linear combine work; O(n log n), tight Θ(n log n) if combine work is Θ(n).

DAA14 warns that a written +O(n) term alone gives an upper bound rather than a matching lower bound. Expanding logarithmically many levels with at most linear aggregate work gives O(n log n). Ordinary merging actually costs Θ(n), supplying the tight result, but a cheaper unspecified combine function need not do so.

Primary note proof

Study-note section DAA14

> Ordinary array merge sort has T(n)=2T(n/2)+Θ(n)=Θ(n log n) across input orders, Θ(n) auxiliary buffer and Θ(log n) recursion depth. A written +O(n) recurrence alone establishes an upper bound; the tight result also uses linear merge work. Empty/singleton subarrays are base cases.

Question sources

Algorithm Design & Analysis review questions · Question 49

Question 135 | NS0517

How does the Optimal Merge Pattern relate to Huffman Coding?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Both repeatedly combine the two smallest weights; Huffman interprets the resulting tree as a prefix code.

DAA22 uses file sizes as merge weights and frequencies as Huffman weights. Reinsert each combined weight so large values avoid unnecessary depth. In Huffman the weighted path length is ∑frequency×depth, the same merge-tree objective. Equal-weight choices can change codewords without changing the optimal total cost.

Primary note proof

Study-note section DAA22

> Combine the two smallest weights, add their sum to total cost, and reinsert it. A min-heap gives O(n log n) time/O(n) space. Cost p+q counts moved records; worst-case comparisons are p+q−1 for nonempty lists, and actual comparisons may be fewer.

Study-note section DAA22

> Huffman uses frequencies as weights and 0/1 branches to leaf symbols, making a prefix-free code. Cost is∑frequency×depth. An exchange makes the two least frequencies deepest siblings; contract them and induct. Ties can change codes without changing optimal cost. Header overhead can outweigh tiny-file savings.

Question sources

Algorithm Design & Analysis review questions · Question 50

Question 136 | NS0518

In Dynamic Programming, what is 'optimal substructure' and give one example.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: An optimal solution contains suitable optimal subsolutions; a DAG shortest path has an optimal suffix.

DAA25 justifies replacing a subsolution with a better one without breaking feasibility. For DAA27's DAG cost-to-go, if a chosen u→v route used a nonoptimal v-to-target suffix, replacing it with the cheaper suffix would improve the whole route, contradicting optimality. The state must retain enough information for this replacement argument to be valid.

Primary note proof

Study-note section DAA25

> Define state meaning, choices, recurrence, bases and dependency order. Prove replacing a subsolution by a better one improves the full solution; otherwise state information may be missing. Show the recurrence covers all legal final choices and generates feasible candidates, then induct in dependency order. Store choices for reconstruction.

Study-note section DAA27

> For a DAG or multistage graph, compute cost-to-go in reverse topological order: F(t)=0 and F(u)=min over outgoing(u, v) of[w(u, v)+F(v)]. Missing routes have∞. Record the minimizing next vertex. Equivalently propagate source distances forward. Complexity is Θ(V+E) with adjacency lists; negative edges are harmless without cycles. Follow the recurrence's dependencies when choosing the evaluation direction.

Question sources

Algorithm Design & Analysis review questions · Question 51

Question 137 | NS0622

What is the difference between Big O, Big Omega, and Big Theta notation? Provide examples.

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: O is an eventual upper bound, Ω a lower bound, and Θ both.

DAA04 defines the relations by fixed constant-factor inequalities beyond a threshold. For $f=3n^2+4n+1$ and $n\geq 1$, $3n^2\leq f\leq 8n^2$, so $f\in\Theta(n^2)$, and consequently both $O(n^2)$ and $\Omega(n^2)$. It is also $O(n^3)$, showing that upper bounds need not be tight or name an input case.

Primary note proof

Study-note section DAA04

> For eventually nonnegative f, g, $f\in O(g)$ means $f(n)\leq c\cdot g(n)$ for all $n\geq n_0$, with fixed $c>0$. Ω reverses that inequality; Θ requires both bounds. Little-o means $f/g\to 0$, and little-ω means $f/g\to\infty$ where the ratio exists. Θ specifies growth up to constant factors, not exact seconds or identical best/worst cases.

Study-note section DAA04

> Also $3n^2+4n+1\leq 8n^2$ for $n\geq 1$.

Study-note section DAA05

> $O(n^2)$ also implies $O(n^3)$: upper bounds need not be tight.

Question sources

Algorithm Design & Analysis review questions · Question 1

Question 138 | NS0623

Why is Merge Sort $\Theta(n \log n)$ in all cases while Quick Sort can be $O(n^2)$?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Merge has balanced halves with linear merging per level; quicksort can repeatedly keep an n−1 side.

DAA14 yields logarithmically many full linear-work merge levels across input orders. DAA15 allows extreme pivots creating the sum $n+(n-1)+\ldots$ and quadratic work. Random pivots improve expected quicksort performance without a worst-case guarantee. The comparison concerns ordinary merge sorting; optimizations that skip merges require their own best-case analysis.

Primary note proof

Study-note section DAA14

> Ordinary array merge sort has $T(n)=2T(n/2)+\Theta(n)=\Theta(n \log n)$ across input orders, $\Theta(n)$ auxiliary buffer and $\Theta(\log n)$ recursion depth. A written $+O(n)$ recurrence alone establishes an upper bound; the tight result also uses linear merge work. Empty/singleton subarrays are base cases.

Study-note section DAA15

> Balanced splits give $\Theta(n \log n)$; repeatedly highly unbalanced splits give $\Theta(n^2)$. Random pivots give expected $\Theta(n \log n)$, not a deterministic guarantee. Median-of-three does not eliminate adversarial cases; three-way partitioning helps repeated keys. Standard quicksort is unstable, uses expected $\Theta(\log n)$ stack but worst $\Theta(n)$; recursing on the smaller side and iterating on the larger bounds stack by $O(\log n)$.

Question sources

Algorithm Design & Analysis review questions · Question 2

Question 139 | NS0624

When would you choose a greedy algorithm over dynamic programming?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Choose greedy when a local-choice safety proof exists and its lower overhead meets the task.

DAA19 supports exchange arguments that insert the greedy choice into some optimum without worsening it. With that proof, irrevocable decisions can avoid a DP table and simplify implementation. If local safety is absent but a correct reusable-state recurrence exists, DP is a stronger candidate. Optimal substructure alone is not enough to justify greedy.

Primary note proof

Study-note section DAA19

> A greedy algorithm repeatedly selects a locally attractive feasible option without later undoing it. Optimal substructure alone does not prove this safe. An exchange proof inserts the greedy choice into an optimum without worsening cost; recurse. MST cut proofs use this exchange.

Study-note section DAA19

> Advantages are often simple decisions and low overhead; limitations are irrevocable errors and problem-specific applicability.

Question sources

Algorithm Design & Analysis review questions · Question 3

Question 140 | NS0625

Why is naive recursive Fibonacci $O(2^n)$ while DP Fibonacci is O(n)?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: Naive recursion repeats states exponentially; DP computes n states once.

DAA26 gives tight naive call growth Θ(φ^n), so O(2^n) is a valid loose upper bound rather than the exact rate. Memoization or tabulation needs Θ(n) unit-cost additions and can keep two rolling values for value-only output. Arbitrary-precision arithmetic grows with output bit length and is not truly constant cost per addition.

Primary note proof

Study-note section DAA26

> Naive Fibonacci recomputes F(n−2) across branches: call count is Θ(φ^n), with O(2^n) a valid loose upper bound. F(0)=0, F(1)=1 and F(i)=F(i−1)+F(i−2) yield Θ(n) unit-cost additions with memoization/tabulation. Keeping only two recent values uses Θ(1) machine words; arbitrary-precision output needs growing bit storage. This naive overlapping recursion is not the independent-subproblem ideal of D&C.

Question sources

Algorithm Design & Analysis review questions · Question 5

Question 141 | NS0626

Apply the Master Theorem to $T(n) = 4T(n/2) + n^2$. What is the solution?

Hint

State the model and invariant, then derive the result from the cited procedure or recurrence, checking base cases and limitations.

Answer: $\Theta(n^2 \log n)$

DAA10 has a=4, b=2, so n^(log_b a)=n^2. The additive f(n)=n^2 matches that power, selecting the balanced second case. Its logarithmically many levels each contribute $\Theta(n^2)$, including the appropriate base work, so the total is $\Theta(n^2 \log n)$. The polynomial-gap regularity condition belongs to the larger-f third case and is unnecessary here.

Primary note proof

Study-note section DAA10

> For T(n)=aT(n/b)+f(n), a≥1, b>1, let p=log_b a. Level i has a^i nodes of size n/b^i. Compare f with n^p: a polynomial gap below gives Θ(n^p); f=Θ(n^p) gives Θ(n^p log n); a polynomial gap above gives Θ(f(n)) if a f(n/b)≤c f(n) for fixed c<1. The third case needs this regularity condition.

Study-note section DAA10

> Also $4T(n/2)+n^2$ gives $\Theta(n^2 \log n)$, case 2.

Question sources

Algorithm Design & Analysis review questions · Question 6