CS Exit Ethiopia

Complete Study Notes

Computer Science Exit Examination

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 6

How to use these study notes

START01 A practical study cycle

Start with a small set of questions before reading.

Mark the exact thing that stopped you: an unknown term, a rule, a calculation, a trace or a choice between alternatives.

Use the question lookup to open the relevant section.

Read one section, close it, and explain the rule in your own words.

Solve the question again and say why each other choice fails.

Then change one assumption or input and solve the variation.

This checks understanding beyond recognition.

A useful 60-minute session: 10 minutes of cold questions, 20 minutes of focused reading, 20 minutes of closed-book solving and 10 minutes updating an error log.

Adjust the timing to your needs.

Revisit mistakes after a gap, for example the next day, several days later and the following week.

These intervals are a practical starting plan, not a fixed learning law.

Record the section ID, your mistaken rule, the corrected rule and one boundary case.

Keep the log short enough to review.

Mix related courses during revision: programming with data structures; databases with security; networking with administration.

Explain how the same word changes meaning across contexts.

START02 What to emphasize

Emphasize definitions, key differences, algorithm costs, SDLC models, databases, networking and security.

Use short oral questions to check understanding and correct common misconceptions.

Combine those habits with the tracing, calculation, design and diagnosis demanded by the Ministry learning outcomes.

Allocate revision by both exam share and your error log.

A course with fewer items can still deserve more time if your foundations are weak.

The blueprint is a planning baseline, not a prediction of exact questions.

Definitions: distinguish nearby concepts, not just their names.

Calculations: write units, formulas, assumptions and intermediate steps.

Code and SQL: check validity before tracing the text; state any changes before analyzing a corrected version.

Algorithms and protocols: know the input, decision rule, state change, termination and failure cases.

Design choices: state the criterion.

A choice can change when cost, reliability, consistency or latency matters more.

Course group	Items in the 100-item blueprint	Revision implication
Web Programming	9	Reserve extra practice for HTML, CSS, JavaScript, PHP, client/server behavior and safe data handling.
Data Structures and Algorithms	7	Trace operations and compare their assumptions and costs.
Each of the other fourteen courses	6 each	Study all sixteen; do not abandon a whole course based on a short note or an alleged repeated-question list.

START03 How to handle exam traps

Underline NOT, FALSE, BEST and EXCEPT before comparing choices.

Check whether the stem supplies enough facts to select one answer.

A long scenario is not automatically an analysis question, and a familiar answer letter is not evidence.

When code is invalid, a diagram is absent, or two choices express the same rule, identify the clarification needed before answering.

Learn the correct rule and the smallest clarification needed.

Do not memorize an arbitrary key as a universal fact.

Distinguish worst case from average case, and an upper bound from a tight bound.

Check language version, SQL dialect, addressing assumptions, graph weights and tie-breaking rules.

For subnetting and scheduling, calculate rather than guess from a familiar number.

Use timed mixed papers only after you can explain your errors.

Revisit the reading sections behind recurring mistakes.

Design and Analysis of Algorithms

Complete Study Notes

DAA01 | Specify the problem and prove correctness

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.

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.

Design and Analysis of Algorithms module

DAA02 | Cost model, experiments and input size

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.

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.

Design and Analysis of Algorithms module

DAA03 | Input cases and bound notation are independent

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.

O/Ω/Θ bound functions independently of case or resource.

A worst-case upper bound covers every size-n input, but O does not mean worst case.

Constant best case and quadratic worst case are not universal.

Design and Analysis of Algorithms module

DAA04 | Formal asymptotic bounds and constants

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.

Prove the inequality for the whole tail, not selected values.

Transitivity allows chained upper bounds; sums are dominated by the largest nonnegative growth term.

Example

Worked microexample

- $10n+5 \le 15n$ for $n \ge 1$, so $c=15, n_0=1$ works. Also $3n^2+4n+1 \le 8n^2$ for $n \ge 1$. The inequality $7n^2+3n+5 \le 8n^2$ is false at $n=1$; $c=15$ works for all $n \ge 1$.

Design and Analysis of Algorithms module

DAA05 | Growth ordering and loose bounds

Representative growth is $1, \log n, n, n \log n, n^2, n^3, \ldots, b^n, n!$ 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.

$O(n^2)$ also implies $O(n^3)$: upper bounds need not be tight.

Several asymptotic bounds can hold simultaneously for the same function.

Neither representative growth ordering nor Big-O labels alone determine finite-input speed.

Example

Worked microexample

- 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 $\Theta(g)$.

Design and Analysis of Algorithms module

DAA06 | Loops, exact counts and additive costs

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 $\Theta(n^2)$.

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.

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.

Example

Worked microexample

- 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).

Design and Analysis of Algorithms module

DAA07 | Analyze conditional nested loops and code validity

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∑m=1.. i−1 $m=i^2(i-1)/2=\Theta(i^3)$.

Summing over i gives $\Theta(n^4)$; condition checks add only $\Theta(n^3)$.

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.

Design and Analysis of Algorithms module

DAA08 | Peak storage and recursion stacks

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.

Slicing/copying arrays can add storage omitted by an index-based pseudocode analysis.

Tail-call elimination is language/runtime-dependent and cannot be assumed for ordinary C++/Java recursion.

Resource tradeoffs include caching subproblem results to save time while consuming memory.

Design and Analysis of Algorithms module

DAA09 | Recurrences: substitution and expansion

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 $\Theta(n \log n)$.

Substitution guesses a bound and proves it inductively with constants/base cases; a lower-order correction may close the induction.

Unequal/decrementing recurrences need separate analysis.

Design and Analysis of Algorithms module

DAA10 | Recursion trees and the Master theorem

For $T(n)=aT(n/b)+f(n)$, $a\geq1$, $b>1$, 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 $\Theta(n^p)$; $f=\Theta(n^p)$ gives $\Theta(n^p \log n)$; a polynomial gap above gives $\Theta(f(n))$ if a $f(n/b)\leq c\, f(n)$ for fixed $c<1$.

The third case needs this regularity condition.

A log-factor boundary may need an extended theorem or explicit summation.

State base work and include leaves.

This version does not apply to $T(n)=T(n-1)+n$ or unequal-split quicksort.

Example

Worked microexample

- $4T(n/2)+n$ gives $\Theta(n^2)$;$2T(n/2)+n$ gives $\Theta(n \log n)$; $T(n/2)+n^2$ gives $\Theta(n^2)$. Each follows a different case. Also $4T(n/2)+n^2$ gives $\Theta(n^2 \log n)$, case 2.

Design and Analysis of Algorithms module

MIT6.006 Recitation3

DAA11 | Comparison sorting lower bounds

A comparison decision tree needs at least n! leaves for distinct-key orderings.

Height h allows at most 2^h leaves, so $h\geq\log 2(n!)=\Omega(n \log n)$; merge sort matches this worst-case bound.

Array indexing is not forbidden.

Counting/radix sorting exploits structured keys outside the comparison-only model.

Recurrences analyze particular algorithms; this decision tree bounds the entire comparison class.

Assess a lower-bound argument by its computational model and reasoning rather than by a technique label alone.

Design and Analysis of Algorithms module

Virginia Tech OpenDSA, Sorting Lower Bound

DAA12 | Choose a design from structural conditions

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.

“Breaks into subproblems,” “sequence of choices,” “incremental,” or “optimization” alone describes several strategies.

Exponential runtime is an outcome, not enough information to select a method.

Specify overlap, local-choice safety, revisability, exactness and usable bounds.

Ordinary selection sort repeatedly selects one minimum; it is not a divide-and-conquer algorithm.

Design and Analysis of Algorithms module

DAA13 | Binary search and simultaneous minimum/maximum

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 $\Theta(\log n)$.

Finding both extrema by two unconditional comparisons per element costs $2n-2$ comparisons, regardless of order.

Pair elements first, compare within each pair, then compare the smaller with minimum and larger with maximum: even n uses $3n/2-2$ comparisons.

A balanced recursive version has $T(n)=2T(n/2)+\Theta(1)=\Theta(n)$.

An unconditional two-if scan does not have an $n-1$-comparison best case.

Design and Analysis of Algorithms module

DAA14 | Merge sort: invariant, trace and costs

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.

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.

Example

Worked microexample

- [4,1,3,2] splits into[4,1] and[3,2], which sort to[1,4] and[2,3]. Merge selects 1,2,3 and appends 4. No assumption that the original array is already sorted is needed.

Design and Analysis of Algorithms module

DAA15 | Quicksort: partition variants and worst cases

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 $i \le j$, 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.

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 $O(\log n)$.

Design and Analysis of Algorithms module

DAA16 | Sorting choices and comparison costs

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.

Heap sort builds a max-heap in $\Theta(n)$, then extracts/restores in $O(\log n)$ per item: $O(n \log n)$ total, $\Theta(1)$ 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.

Design and Analysis of Algorithms module

DAA17 | Strassen multiplication and divide/combine tradeoffs

Ordinary n×n multiplication computes n products/sums per output entry: $\Theta(n^3)$.

Eight half-size block products still give $8T(n/2)+\Theta(n^2)=\Theta(n^3)$.

Strassen uses seven products plus block sums/differences:$7T(n/2)+\Theta(n^2)=\Theta(n^{\log_2 7})\approx\Theta(n^{2.807})$.

Zero-padding preserves the required product.

The smaller exponent trades extra additions/temporary storage and possible numerical issues against multiplication work.

Use an ordinary base case for small matrices and specify exact versus floating-point arithmetic.

Design and Analysis of Algorithms module

DAA18 | Maximum contiguous subarray: two valid strategies

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 $2T(n/2)+\Theta(n)=\Theta(n \log n)$.

Kadane's DP keeps the best sum ending at i: $end[i]=\max(A[i], end[i-1]+A[i])$; keep the largest end seen.

This is $\Theta(n)$ time and $\Theta(1)$ extra space.

For a required nonempty subarray initialize from A[0], not zero; all-negative input then returns its largest element.

Empty-allowed variants may return zero.

Both D&C and DP can solve the maximum-subarray problem; specify the algorithm before assigning its complexity.

Example

Worked microexample

- For[−2,3,−1,4,−5], ending sums are−2,3,2,6,1; optimum 6 comes from[3,−1,4]. For[−4,−2], a nonempty answer is−2.

University of Alberta CMPUT498/501 Lecture2

DAA19 | Greedy correctness, limits and counterexamples

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.

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.

Design and Analysis of Algorithms module

DAA20 | Fractional knapsack versus indivisible items

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.

The exchange argument replaces lower-density selected weight by available higher-density weight; value cannot decrease.

Sorting costs O(n log n), followed by a linear scan.

Handle zero-weight beneficial items separately rather than divide by zero.

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.

Example

Worked microexample

- Weights 15,10,18 and values 24,15,25 with capacity 20: take the weight 15 item and half the weight 10 item. Value 24+7.5=31.5; initial remaining capacity must be 20, not the number of items.

Design and Analysis of Algorithms module

DAA21 | Unit-time job sequencing with deadlines

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.

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.

Example

Worked microexample

- 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.

Design and Analysis of Algorithms module

DAA22 | Optimal merge patterns and Huffman coding

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.

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.

Example

Worked microexample

- For weights 2,3,5,6: merge 2+3=5, then 5+5=10, then 10+6=16. Total weighted merge cost 31; the same tree construction minimizes weighted prefix-code length.

Design and Analysis of Algorithms module

DAA23 | MSTs: Prim, Kruskal and the cut proof

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.

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.

Example

Worked microexample

- Triangle AB=1,BC=2,AC=4: Kruskal accepts AB and BC for cost 3. If every edge is 1, any two edges give an MST, so uniqueness fails.

Design and Analysis of Algorithms module

DAA24 | Dijkstra: relaxation and the nonnegative condition

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.

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.

Design and Analysis of Algorithms module

DAA25 | DP states, optimal substructure and proof

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.

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.

Design and Analysis of Algorithms module

DAA26 | Memoization, tabulation and Fibonacci

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.

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.

Design and Analysis of Algorithms module

DAA27 | Multistage/DAG paths and Bellman-Ford

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.

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.

Example

Worked microexample

- s→a=2,s→b=5,a→t=4,b→t=−1 gives F(a)=4,F(b)=−1 and F(s)=min(6,4)=4 via b.

Design and Analysis of Algorithms module

DAA28 | Floyd-Warshall, reachability and negative cycles

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.

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.

Example

Worked microexample

- If D[a,b]=4,D[b,c]=2,D[a,c]=11, phase b reduces D[a,c] to 6. Later phases must retain that improvement rather than restoring 11.

Design and Analysis of Algorithms module

DAA29 | 0/1 knapsack DP and dominance

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.

Time O(nW) is pseudo-polynomial because W takes only O(log W) binary input bits.

Storage can be O(W); update capacities downward for 0/1 selection.

Upward iteration allows reuse and solves an unbounded variant.

In sparse(weight, profit) states, discard an option dominated by another with no greater weight and no smaller profit, provided both have the same remaining items.

Example

Worked microexample

- Weights 2,3,4, values 1,2,5 and W=6 yield value 6 by choosing weights 2+4. A state(weight 5,profit 3) is dominated by(weight 4,profit 5).

Design and Analysis of Algorithms module

DAA30 | LCS: prefix states and reconstruction

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.

Fill rows/columns in increasing order.

Reconstruct by taking matching characters diagonally or following a neighbor with the same optimal value; ties may produce different LCSs.

Time Θ(mn), table storage Θ(mn); two rows reduce value-only storage to O(min(m, n)), while ordinary reconstruction needs retained decisions or another technique.

Example

Worked microexample

- X=ABC,Y=AC: LCS is AC, length 2. Contiguous-substring matching would require a different recurrence, resetting a mismatch to zero.

MIT6.006 Lecture16

DAA31 | Interval DP: matrix chains and optimal BSTs

For matrix A_i of size p[i−1]×p[i], M[i, i]=0 and M[i, j]=min over i≤k<j of(M[i, k]+M[k+1, j]+p[i−1]p[k]p[j]).

Fill by increasing interval length and store the split.

Three matrices 10×30,30×5,5×60 cost 4500 under(AB)C versus 27000 under A(BC).

Time O(n^3), space O(n^2); multiplication order stays fixed while parentheses change.

For an OBST use sorted-key success weights p_i and gap failure weights q_i.

Let W(i, j)=∑p_i.. p_j+∑q_(i−1).. q_j.

Under the comparison-count convention, C(i, i−1)=0 and C(i, j)=W(i, j)+min_r(C(i, r−1)+C(r+1, j)); store r. CLRS-style dummy-node depths instead initialize empty cost to q_(i−1); final values differ by the fixed total gap weight.

Basic time O(n^3), space O(n^2).

Design and Analysis of Algorithms module

Stony Brook University, Dynamic Programming

DAA32 | Reliability design under a budget

Assume independent device failures.

With m_i parallel copies of stage i, each reliability r_i, stage reliability is φ_i(m_i)=1−(1−r_i)^m_i.

Series stages multiply: total∏φ_i.

Choose integers m_i≥1 with∑m_i c_i≤B; extra copies improve reliability but consume budget.

Correlated failures invalidate the simple independence formula.

Let R[i, b] be best reliability for the first i stages within budget b. Base R[0, b]=1; maximize R[i−1, b−m c_i]φ_i(m) over feasible m≥1.

Empty candidate sets give 0.

Time is O(stages×budget×maximum copies), with O(budget) value storage; reconstruct retained copy counts.

Example

Worked microexample

- For costs 30,15,20, reliabilities 0.9,0.8,0.5 and budget 105, copy counts(1,2,2) cost 100 and yield 0.9×0.96×0.75=0.648. Three copies of the weak last stage with the others single cost 105 but yield only 0.63.

Design and Analysis of Algorithms module

DAA33 | Exact TSP with subset DP

TSP asks for a minimum-cost tour visiting every vertex once and returning to its start; a shortest path need not visit all vertices or return.

Fix start s. Let F(S, j) be minimum cost from s through exactly S, ending at j∈S. Base F({s}, s)=0.

Transition F(S, j)=min_i∈S−{j}[F(S−{j}, i)+w(i, j)].

Finish with min_j≠s[F(V, j)+w(j, s)].

Missing edges cost∞.

Store predecessor choices to reconstruct a tour.

Held-Karp uses O($V^2$2^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.

Specify the problem variant and algorithm before choosing a solution method.

For Held-Karp, S contains the start s. Initialize F({s},s)=0 and impossible states to infinity; for larger sets, evaluate endpoints j other than s. These boundary cases prevent using a nonexistent predecessor state.

Design and Analysis of Algorithms module

DAA34 | BFS, DFS and graph representation

BFS marks a vertex when enqueued, explores neighbors with a FIFO queue and records predecessor/level.

Levels give shortest paths by edge count in unweighted/equal-weight graphs.

DFS marks on entry, recursively or iteratively explores an unvisited neighbor to completion, then resumes the parent.

Run from each unvisited vertex for a full disconnected traversal.

Adjacency lists give O(V+E) traversal time and O(V) auxiliary visited/queue/stack space; adjacency matrices give O(V^2) scanning.

State neighbor order before requesting one exact traversal sequence.

Reverse DFS finish order gives a topological order only for DAGs.

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

Design and Analysis of Algorithms module

DAA35 | Articulation points and biconnected components

An articulation vertex increases the number of connected components when removed.

In an undirected DFS, disc[u] is discovery time and low[u] is the smallest discovery number reachable from u's subtree using downward tree edges and at most one back edge.

Update low from children and nonparent back edges.

A root is articulation if it has at least two DFS children; a nonroot u is articulation if some child v has low[v]≥disc[u].

Such a child cannot reach an ancestor of u without u. A bridge has the stricter condition low[v]>disc[u].

An edge stack can output a vertex-biconnected block when low[v]≥disc[u], popping through edge(u, v).

With adjacency lists these checks cost O(V+E).

Distinguish vertex-biconnectivity from strong connectivity in directed graphs.

For a simple undirected graph initialize disc[u]=low[u]=++time on entry.

For an unvisited neighbor v, push tree edge(u,v), DFS(v), then low[u]=min(low[u],low[v]).

For a visited nonparent ancestor v, push the back edge once when disc[v]<disc[u] and use low[u]=min(low[u],disc[v]).

After a child returns with low[v]>=disc[u], pop through(u,v) to output one block.

Keep the separate root-child articulation rule and bridge test low[v]>disc[u].

Restart DFS at each unvisited vertex.

For multigraphs skip the parent edge by identity rather than skipping every edge to the parent.

Example

A small procedure check

- Two triangles A-B-C-A and C-D-E-C, DFS order A,B,C,D,E: disc=(1,2,3,4,5), low=(1,1,1,3,3). Only C is articulation; no edge is a bridge; stack yields blocks{A-B,B-C,C-A} and{C-D,D-E,E-C}. low[D]=disc[C] shows why articulation uses>= but bridge uses>.

Design and Analysis of Algorithms module

IARCS, Articulation Points

CMU15-451: Undirected DFS and Biconnected Components

DAA36 | Backtracking template and safe pruning

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.

Correctness requires exhaustive consideration of every feasible extension and pruning only impossible branches.

Merely disliking a partial objective is not enough.

DFS recursion uses space proportional to search depth, while total search can remain exponential.

Memoizing equivalent states or stronger constraints can reduce work, but branch count depends on actual choices/tests.

Design and Analysis of Algorithms module

DAA37 | N-queens: row, column and diagonal tests

Use x[r]=column for one queen per row.

Reject against earlier row i when x[r]=x[i] or $|x[r]-x[i]|=|r-i|$; distinct rows are built into the representation.

Recurse after legal placement and remove it on return.

Column and r−column/r+column sets permit constant-time conflict tests.

Distinct columns leave at most n! complete permutations plus partial-node work; pruning does not guarantee polynomial time.

Placement/conflict storage is O(n).

Boards 2 and 3 have no solutions; state empty-board conventions.

Example

Worked microexample

- For n=4, column vector(2,4,1,3) is valid. Rows 1 and 2 with columns 2 and 3 would share a diagonal because both differences are 1, so that partial branch is rejected.

Design and Analysis of Algorithms module

DAA38 | Subset sum: include/exclude and admissible bounds

Include/exclude each item while tracking current sum s and remaining sum R. For nonnegative weights prune s>target or s+R<target.

On equality report the subset, considering zero-weight extensions if all subsets are needed.

Sorting positive weights can strengthen tests but cannot force inclusion of the next item.

With negative values, s>target may recover later; use valid attainable-range bounds.

There are up to 2^n subsets and O(n) recursion depth.

Define whether duplicate-valued items count by index.

Example

Worked microexample

- For{3,4,5,6}, target 13, subset{3,4,6} succeeds. At s=9 with remaining total 3, s+R=12<13 proves failure. At s=9 with remaining 4, equality can still be reached.

Design and Analysis of Algorithms module

DAA39 | Graph coloring: feasibility versus minimum colors

For a fixed m, assign colors 1…m to vertices, rejecting a color used by an already colored neighbor.

Backtrack when all colors fail.

At most m^V complete assignments exist, with additional checking cost; a simple O(V)-per-test implementation gives an exponential bound.

A self-loop makes proper coloring impossible.

The decision question asks whether m colors suffice; optimization asks for the chromatic number.

Trying increasing m with a complete decision search finds the minimum.

Greedy coloring chooses a color by vertex order and may use too many; it is not a general optimum proof.

An odd cycle needs 3 colors; an edge-containing bipartite graph needs 2, and an edgeless nonempty graph needs 1.

Design and Analysis of Algorithms module

DAA40 | Hamiltonian cycles and permutation search

Fix a start vertex to avoid rotational duplicates.

Extend a path with an unused vertex adjacent to its current end.

At V vertices require an edge back to the start; otherwise backtrack.

A Hamiltonian cycle visits vertices once, while an Euler cycle concerns visiting edges.

Fixing the start leaves up to(V−1)! candidate permutations, with polynomial checking overhead.

For a simple undirected graph with at least 3 vertices, a degree 1 vertex or articulation point rules out a Hamiltonian cycle, but their absence does not prove one exists.

TSP optimizes a weighted Hamiltonian tour; the decision existence search alone does not minimize its cost.

Reversed undirected tours may be equivalent if enumerating unique cycles.

Design and Analysis of Algorithms module

DAA41 | Branch and bound: node states and bound direction

A live node is generated and still eligible for expansion; the E-node is currently expanded; a dead node is finished or pruned.

FIFO selects live nodes by queue order, LIFO by stack order, and least-cost/best-bound uses a priority queue.

Selection changes work/storage; valid bounds preserve exactness.

For minimization, an incumbent feasible solution supplies an upper bound U; a partial node supplies a lower bound L on every completion.

Prune if L≥U when only one optimum/value is needed.

For maximization, feasible profit is a lower bound and an optimistic completion estimate is an upper bound; prune if upper≤incumbent.

Keep equality branches when all optimal solutions must be enumerated.

Design and Analysis of Algorithms module

DAA42 | Knapsack branch and bound

Sort positive-weight/value items by decreasing density and branch include/exclude.

Reject overweight nodes.

To bound a node’s possible profit, fill its remaining capacity fractionally from remaining items; this relaxation can only improve on 0/1 feasibility, so its value is an upper bound.

A whole-item feasible completion updates the incumbent; the fractional result itself is not a legal 0/1 answer.

With capacity 10 and items(6,12),(5,8),(5,8), the root fractional bound is 18.4: take the first and 4/5 of the second.

The first-item-in branch has whole-item value 12; excluding it permits the two other items, value 16.

Continue until every live upper bound is≤16 to certify optimality.

Avoid downward floating-point rounding that could invalidate the bound; worst-case search remains exponential.

Design and Analysis of Algorithms module

DAA43 | TSP branch and bound by matrix reduction

For directed TSP, set diagonal/forbidden entries to∞.

Subtract each finite row minimum, then each finite column minimum, accumulating reductions.

Every tour uses one entry per row/column, so their sum is a lower bound; the relaxed assignment may contain disjoint subtours.

When including edge i→j, remove its used row/column, prevent premature subtours, reduce again and add the selected reduced-edge cost and new reductions to the parent bound.

Compare bounds with the best complete tour and expand a promising live node.

Never treat a reduced matrix's zeros as proof of a complete valid tour.

Branch on an undecided edge i->j. The include child deletes row i/column j and adds its selected reduced cost plus new reductions to the parent bound.

The exclude child sets that entry to infinity and adds only new reductions.

Keep forced in/out-degree at most one.

Track each forced path from head h to tail t and forbid t->h until the path contains every vertex; this blocks premature cycles.

A required residual row/column with no finite choice is infeasible and pruned.

Accept only one complete cycle through every vertex, not a set of disjoint cycles.

Example

A small procedure check

- For four cities, cheap forced edges1->2,2->1 and3->4,4->3 satisfy one incoming/outgoing edge per city but are two disjoint2-cycles, not a tour. Once forced path1->2 is formed, forbid2->1 while fewer than4 vertices are in the path. Preserve the existing3-city reduction example: bound25 equals tour cost25.

Design and Analysis of Algorithms module

BYU CS312: TSP Branch and Bound

DAA44 | P, NP, reductions and encoding size

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.

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.

University of Waterloo CS365, P versus NP

UC Davis ECS120, NP-Completeness and Reductions

DAA45 | Check algorithm specifications and cost assumptions

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.

A precise degree for 3n?+7n+12 requires the missing exponent; fixed nonnegative integer exponents still give polynomials.

Use valid declarations rather than ini/inl or a redeclared n; exact counts also need a primitive-cost model.

Justify the result from the specification, valid code and cost assumptions.

Design and Analysis of Algorithms module