CS Exit Ethiopia

Revision Outlines

Computer Science Exit Examination

1 selected course | 18 November 2026

Selected courses

01. Data Structure and 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.

Data Structure and Algorithms 3

Data Structure and Algorithms

Revision Outlines

Data types and representations | Exam focus

- ADT versus representation
  - ADT values and operations
  - Array-backed LIFO stack
  - Linked-node LIFO stack
  - Abstraction: relevant details
  - Patient urgency and arrival order
  - Hair colour irrelevant
- Structure classifications
  - Linear: arrays, lists, stacks, queues
  - Trees: hierarchies
  - Graphs: general connections
  - Heterogeneous records
  - Integer, character, floating fields
  - Ordinary array: one element type

Arrays: access, capacity, shifting | Exam focus

- Array address arithmetic
  - Equal-size elements
  - Zero-based indexing
  - address(A[i])=base+i×element_size
  - RAM positional access: O(1)
  - Row-major rectangular matrix
  - base+(row×columns+column)×element_size
- Array capacity and growth
  - Fixed capacity chosen beforehand
  - Capacity choice: beyond compile time
  - Resizing: larger block and copy
  - Middle insertion: linear shifts

- Spare-capacity append: O(1)
- Resizing append: O(n)
- Geometric growth: amortized O(1) append

• Worked example Insertion by rightward shifting
- Spare slot required
- Insert 9 at index 1
- Shift 6 before 4
- [2,4,6]→[2,9,4,6]

Operations, records, memory ownership

• Structure operations
- Traversal: every item
- Searching: target location
- Insertion/deletion: membership changes
- Updating: stored value changes
- Sorting: item reordering
- Merging: sequence combination
- Sorted merge: smaller front first
- [1,4]+[2,3]→[1,2,3,4]
- Destruction: owned storage release

• Records and memory ownership
- C struct: named fields
- Array and pointer fields
- Singly linked node: payload, same-type link
- C declaration: no field initializers
- Modern C++: default member initializers
- new↔delete
- new[]↔delete[]
- Repair links before freeing
- No dereferencing freed nodes

Algorithm properties

• Separate algorithm obligations

- Finiteness: every valid input terminates
- Definiteness: unambiguous instructions
- Effectiveness: executable steps
- Correctness: specified valid-input results
- Finite algorithms can be incorrect

• Problem and algorithm specification
- Inputs, outputs, assumptions
- Procedure language independence
- Implementation language dependence
- Efficiency resource model
- Omitted characteristic ≠ false characteristic

Resource cost models

• Resource cost models
- Time: chosen operations
- Space: memory and live recursive frames
- Distributed communication volume
- Assignment unit ≠ one second
- Sequential costs add
- Independent nested iteration counts multiply
- Conditional worst case: test+costlier branch

• Input-case comparisons
- Equal input sizes
- Best, average, worst cases
- Average requires distribution
- Expected total=m f(n)
- Independence unnecessary
- Exact coefficients depend on operation counting

Asymptotic bounds

• Asymptotic bound meanings
- Eventually nonnegative functions
- O: $f(n) \leq c\, g(n)$, $n \geq k$

- c>0
- Ω: reversed inequality
- Θ: upper and lower bounds
- o: f/g→0
- ω: f/g→∞
- Best/average/worst-case functions
- Space functions also bounded

• Polynomial bounds and witnesses
- $3n^2+4n+1 \leq 8n^2$, $n \geq 1$
- $O(n^2)$: c=8,k=1
- Ignore constants and lower-order terms
- Transitivity: f≤cg≤cdh
- Both thresholds required

• Worked example Increasing growth rates
- 1<log n<n<n log n
- n log n<n^2<n^3
- n^3<2^n<3^n
- 3^n<n!<n^n
- Log bases: constant-factor difference

• Exam trap Tightness versus valid upper bounds
- Linear functions also $O(n^2)$
- Tightest listed order usually intended
- Mixed O/Θ options may overlap

Loops and hidden access costs

• Loop and stack counting
- i*=2: about $\log_2 n$ iterations
- Independent nested n-loops: $\Theta(n^2)$
- Constant work per iteration
- n recursive levels: Θ(n) stack
- Constant live storage per level

• Hidden indexed-access costs
- List implementation matters

- Java LinkedList.get(i): nearer-end traversal
- get(0),get(2),...: $\Theta(N^2)$ total
- Many middle indices: linear traversal each
- Iterator scan: $\Theta(N)$
- Damaged return: runtime uncertainty
- Missing cost convention: no exact runtime

Singly linked chains | Exam focus

- Singly linked chain representation
  - Node: data and next
  - Head: first node
  - Noncircular terminator: null
  - Empty: head==null
  - Dynamic node allocation
  - Pointer storage overhead
  - No direct index arithmetic
- Linked-chain traversal
  - Follow next from head
  - Stop at null
  - Display/count: process every node
  - Counting: $\Theta(n)$
  - Maintained size avoids recounting
  - Array-linked lists still follow links
  - Separate positional index exception

Safe linked-list updates | Exam focus

- Safe insertion and head deletion
  - Prepend: new.next=head; head=new
  - Insert-after: new.next=p.next before p.next=new
  - Head-only append: tail traversal
  - Maintained tail avoids append traversal
  - Delete-head: save old=head
  - Advance head=head.next

- Release old in C++
- Deletion boundary cases
  - Singly linked tail needs predecessor
  - Known tail insufficient
  - Empty and single-node cases
  - Only-node removal: head=tail=null
  - Successor deletion: bypass before deallocation
  - Clear owned chain: visit/release every node
- Three-pointer chain reversal
  - Initially prev=null, cur=head
  - Save next=cur.next
  - Redirect cur.next=prev
  - Advance prev=cur; cur=next
  - Finish head=prev
  - Linear time
  - Constant extra pointers
- Worked example Insertion link-order trace
  - 3→7→9; insert 5 after 3
  - First 5→7, then 3→5
  - Reversed assignments lose 7→9

List variants and practical choices | Exam focus

- Doubly linked and circular lists
  - Doubly linked nodes: prev
  - Head/tail: constant-time end operations
  - Singly linked tail deletion: predecessor needed
  - Circular tail.next=head
  - Traversal stops on return to start
  - Handle empty circular list separately
- Practical linked-list selection
  - Changing sequences with known insertion positions
  - Position search: linear
  - Polynomial terms: coefficient, exponent

    - Directory and block links
    - Nonbinary-tree child lists
    - Hash chains
    - Browser history: doubly linked cursor
    - Playlist: circular option
    - Airline seats: specify record model
- Index access comparison
    - Array: O(1)
    - Singly linked list: O(n) worst
- Front insertion comparison
    - Array: O(n) shifts
    - Singly linked list: O(1)
- Append comparison
    - Array spare capacity: O(1)
    - Linked list with tail: O(1)
    - Linked list head only: O(n)
- Tail deletion comparison
    - Array: O(1)
    - Singly linked list: O(n)
    - Predecessor required

Stack implementation conventions | Exam focus

- Stack convention and operations
    - LIFO=FILO
    - Push: insert
    - Pop: remove/return top
    - Peek: observe top
    - Zero-based occupied-slot top
    - Empty: top=-1
    - Full: top=C-1
    - Check before A[++top]=x
    - Check before returning A[top--]
- Alternative stack implementations

- Count-based empty: size=0
- Count-based push: A[size++]=x
- Keep empty/full conventions consistent
- Fixed-array push/pop: O(1)
- Linked stack: head insertion/deletion O(1)
- Singly linked tail pop: predecessor traversal

• Exam trap Stack failure conditions

- Overflow: insertion into full storage
- Underflow: removal from empty storage
- isEmpty check ≠ underflow

Stack traces and applications | Exam focus

• Stack operation tracing

- Record bottom-to-top after each instruction
- Record popped values separately
- Push 1,2; pop→2; remaining [1]
- Complete output: 2,2,1,1,2

• Stack applications

- Nested calls: latest returns first
- Bracket matching: openings pushed
- Closing bracket matches top
- Reversal: push input then pop
- Undo: latest action→redo
- Redo: action→undo
- New edit after undo: clear redo
- DFS/backtracking: latest pending choice

Expression notation and evaluation

• Expression notation and conversion

- Infix: operator between operands
- Precedence and parentheses
- Prefix: operator first
- Postfix: operator last

- Fixed arity determines grouping
- Infix→postfix: output operands immediately
- Stack pending operators
- Pop higher precedence before pushing
- Left-associative: pop equal precedence
- Parentheses override operator order

• Postfix evaluation and expression trees

- Push operands
- Binary operator: pop right then left
- Push left op right
- Prefix tree traversal: preorder
- Postfix tree traversal: postorder
- Stack useful, other representations possible

• Worked example Expression conversion traces

- A+B*C-D/E→ABC*+DE/-
- 8 3 -: pop 3 then 8
- 8−3=5
- (A+B)*C→AB+C*
- Prefix: *+ABC

FIFO queue operations | Exam focus

• Ordinary queue policy

- FIFO=first-come, first-served
- Enqueue at rear
- Dequeue/peek at front
- Linked head/tail operations: O(1)
- Append tail, remove head
- First insertion: set both pointers
- Last removal: clear both pointers
- Head-only insertion: tail traversal

• Queue applications and modeling

- Equal-priority printer jobs
- Shared-resource waiting

- Asynchronous buffers
- Level-order exploration
- Play sequence
- Airline seats versus waiting requests
- Application name insufficient for representation

• Worked example Conditional queue trace
- Enqueue 4,8; peek=4
- peek>5 false: no removal
- Enqueue 1,6; dequeue
- Result: [8,1,6]

Circular buffer conventions | Exam focus

• Circular buffer count convention
- Noncircular rear exhaustion: false overflow
- Released front slots reusable
- Modulo wraparound
- Front: next removal
- Size: item count
- Insertion slot=(front+size)%C
- Empty: size==0
- Full: size==C

• Circular dequeue and alternate convention
- Read A[front]
- front=(front+1)%C
- Decrease size
- Alternative: reserve one unused slot
- Full: next(rear)==front
- Different empty test and capacity
- State convention before tracing

• Worked example Physical versus logical queue order
- Capacity six, initially full
- Remove 11; enqueue 9 at slot 0
- Physical: [9,13,15,17,19,21]

- Logical: [13,15,17,19,21,9]
- Noncircular version may overflow

Deques and queue policies | Exam focus

- Deque versus dequeue
  - Deque: insertion/removal at both ends
  - No guaranteed middle access
  - Linked endpoints: O(1)
  - Linked middle indexing: O(n)
  - Circular-array deque: possible indexed access
  - Dequeue: removal operation
- Queue combination and capacity policies
  - Merge: concatenate or alternate fronts
  - Predicate split: matching output queue
  - Append preserves subgroup order
  - Full buffer: reject, block, overwrite
  - Capacity alone specifies no policy

Priority queues and tie rules | Exam focus

- Priority queue service
  - Minimum or maximum priority convention
  - Urgency before arrival
  - Interleaved insertions and extractions
  - Binary heap peek: O(1)
  - Binary heap insert/extract: O(log n)
- Priority queue implementations and ties
  - Unordered sequence: cheap insertion
  - Unordered sequence: linear extremum removal
  - Ordered sequence: reversed tradeoff
  - Equal priority ≠ inherent FIFO
  - Stable service: (priority,arrival_counter)
  - Specify priority direction and tie rule
- Exam trap FIFO tie-policy trap

- Guide assumes undefined stability
- FIFO ties: additional policy
- No universal priority-queue stability

Tree vocabulary and binary shapes

- Rooted-tree vocabulary
  - Root: no parent
  - Other nodes: one parent
  - Connected and acyclic
  - n nodes: n−1 edges
  - Unique path between each pair
  - Depth: root-to-node edges
  - Height: maximum depth
  - Single-node height: zero
  - Leaves: no children
- Binary-tree shape distinctions
  - Binary: at most two children
  - Full: zero or two children
  - Complete: earlier levels filled
  - Complete: last level fills leftward
  - Perfect: leaves at same level
  - Perfect: two children per internal node
  - Balanced: logarithmic height under stated rule
  - Balanced ≠ necessarily perfect

BST ordering and search | Exam focus

- Global BST ordering invariant
  - Distinct keys assumption
  - Entire left subtree smaller
  - Entire right subtree larger
  - Immediate-child checks insufficient
  - 10→left 5→right 12: invalid
  - Duplicates: count or placement policy

- BST operations and search costs
  - Compare; follow one child
  - Search ends at equality or null
  - Insertion: same path, new leaf
  - Operation cost: O(height)
  - Balanced: logarithmic; chain: linear
  - Root hit: $\Theta(1)$ best case
  - Optimal BST: access probabilities
  - Minimize expected search cost
  - Construction distinct from searching
- Optimal-BST dynamic programming
  - Every key interval
  - Try each possible root
  - $O(n^2)$ intervals × O(n) roots
  - Construction: $O(n^3)$
  - Probability-weighted search depth
  - Objective ≠ minimum worst-case height
- Worked example BST insertion shape trace
  - Insert 40,20,50,10,30,60
  - 40 children: 20,50
  - 20 children: 10,30
  - 50 right child: 60
  - Height: two edges
  - Sorted 1,2,3,4,5: right chain

Tree traversal rules

- Tree traversal rules
  - Preorder: node,left,right
  - Inorder: left,node,right
  - Postorder: left,right,node
  - Recursive application; null does nothing
  - Sorted inorder requires BST
  - Preorder: copying

- Postorder: children-before-parent deletion
- Level order: FIFO queue

• Traversal worked tree

- Root 4; children 2,6
- Leaves 1,3,5,7
- Preorder: 4,2,1,3,6,5,7
- Inorder: 1,2,3,4,5,6,7
- Postorder: 1,3,2,5,7,6,4
- Level order: 4,2,6,1,3,5,7
- Randomized visitation possible
- Randomized ≠ standard named DFS order

BST deletion methods

• BST deletion cases

- Leaf: clear parent link
- One child: parent bypass
- Two children: copy inorder replacement
- Predecessor: left-subtree maximum
- Successor: right-subtree minimum
- Delete replacement at original position
- Replacement deletion: at most one child

• BST deletion by merging

- Promote right subtree
- Attach old left at right minimum
- Symmetric left promotion
- Root deletion: update root
- Copying versus reconnecting: different valid shapes
- Name deletion method before tracing

Balanced search trees

• AVL balancing and rotations

- Balance factor=height(left)−height(right)
- Allowed factors: −1,0,1

- Insertion/deletion: update heights upward
- LL: right rotation
- RR: left rotation
- LR: left-child left, node right
- RL: symmetric double rotation
- Rotations preserve inorder keys

• Balanced search-tree guarantees
- 30,20,10: LL imbalance
- Repair root 20; children 10,30
- Red-black colouring discipline
- Logarithmic height
- Avoid linear worst-case search
- Ordered ranges versus ordinary hashing

Heap order and indexing

• Heap shape and order
- Binary heap: complete tree
- Max-heap: parent≥children
- Min-heap: reversed order
- No BST-style subtree ordering
- Max-heap root: maximum
- Distinct keys: second largest root child
- Duplicate maxima: distinct runner-up may deepen
- [9,9,9,8,7,6,5]: runner-up 8, index 3
- Minimum may occupy any leaf
- Minimum search: linear worst case

• Heap array indexing
- Zero-based root: 0
- Zero-based children: 2i+1,2i+2
- Zero-based parent: floor((i−1)/2), i>0
- One-based children: 2i,2i+1
- One-based parent: floor(i/2)
- Child index < current heap size

- Exam trap Heap indexing convention trap
  - Right child 2i+1: one-based
  - Zero-based heap indexing
  - Explicit convention change required

Heap restoration and construction | Exam focus

- Heap restoration and construction
  - Insert at next complete-tree position
  - Swim while parent order violated
  - Root removal: final element→root
  - Reduce heap size
  - Sink with more dominant child
  - Stop when heap order restored
  - Swim/sink: logarithmic paths
  - Bottom-up internal-node sinking: O(n)
  - Repeated insertion construction: O(n log n)
- Worked example Max-heap removal trace
  - Start [10,8,6,2,1,4,5]
  - Remove 10; move 5; sink
  - Result [8,5,6,2,1,4]
  - Remove 8; move 4; exchange 6
  - Result [6,5,4,2,1]

Addressing and hash functions

- Direct addressing versus hashing
  - Direct addressing: every possible key
  - Finite universe U: $\Theta(|U|)$ slots
  - Hash address $h(key) \in 0..m-1$
  - Actual entries plus bucket capacity
  - $h(k)=k \bmod 6$
  - 36 home address: 0
  - Collision may change final storage slot
- Hash-function properties

- Determinism: equal keys, same chosen-function address
- Uniformity: evenly spread expected keys
- Universal hashing: random suitable-family function
- One function chosen per table
- Choice independent of fixed key set
- No unrelated fresh function per lookup
- Collision probability controlled
- Zero collisions not guaranteed

Collisions, probing, load factor | Exam focus

• Collision handling and load
  - Separate chaining: bucket lists
  - Load α=n/m
  - α: average chain length
  - Expected O(1+α): suitable distribution
  - Worst case: O(n)
  - Open addressing: entries inside array
  - Probe until key or available slot
• Probing schemes and deletion
  - Linear probing: (h(k)+j)%m
  - m=7; keys 10,17,24
  - Home 3; final positions 3,4,5
  - Quadratic probing: changing step
  - Double hashing: (h1(k)+j h2(k)) mod m
  - Step relatively prime to m
  - Relatively prime step: all-slot coverage
  - Collisions remain possible
  - Deletion: tombstones or reinsertion
  - Avoid premature search termination
• Linear-probing wraparound
  - Seven slots; home address 3
  - Probe order: 3,4,5,6,0,1,2
  - Home slot ≠ guaranteed final slot

Maps, sets, dictionaries, caches | Exam focus

- Maps, sets, lists: contracts
  - Map: unique keys→values
  - Different keys may share values
  - Map entries traversable
  - Set: no duplicate elements
  - Null policy: implementation-dependent
  - Iteration order: implementation-dependent
  - Lists preserve positions
  - Lists generally allow duplicates
  - Java LinkedList permits null
- Dictionaries, symbol tables, caches
  - Hashing: expected O(1) lookup
  - Balanced tree: ordered O(log n) lookup
  - Bounded cache: eviction policy required
  - FIFO eviction: queue
  - LRU: hash+doubly linked recency list
  - Hit: move to recent end
  - Full: evict least-recent end
  - Best structure depends on operations/policies

Linear search | Exam focus

- Linear search cost cases
  - Successive target comparisons
  - Stop at equality or exhaustion
  - Sorted or unsorted data
  - First match: Θ(1)
  - Absent/last match: n comparisons
  - Worst time: Θ(n)
  - Uniform successful positions: (n+1)/2 average
  - Average remains linear
- When scanning suffices
  - Single unsorted query: sorting rarely justified

- Small lists: setup costs matter
- Mean: examine all values
- Variance: examine all values
- Arbitrary parity counts: examine all values
- Sorting avoids none of these scans

Binary search | Exam focus

- Binary search interval invariant
  - Ascending random-access sequence
  - lo=0; hi=n−1
  - mid=lo+(hi−lo)//2
  - Equality: finish
  - target<A[mid]: hi=mid−1
  - Otherwise lo=mid+1
  - Failure: lo>hi
  - Descending order: reverse comparisons
  - Comparable keys; finite valid bounds
  - Fixed preallocation unnecessary
- Binary search costs and variants
  - Worst probes=$\lfloor \log_2 n \rfloor + 1$, $n > 0$
  - 100 keys: seven probes
  - Million keys: about twenty probes
  - Iteration: constant space
  - Recursion: logarithmic stack
  - Linked midpoint traversal loses logarithmic total
  - Boundary variants: retain candidate at equality
- Worked example Binary search worked interval
  - Target 19 in [11,13,15,17,19,21]
  - mid=2: 15
  - Keep indices 3..5
  - mid=4: 19
  - Binary debugging: ordered good/bad boundary
  - Arbitrary fault locations insufficient

Search variants and assumptions | Exam focus

- Interpolation search assumptions
  - Numeric index estimation
  - lo+(hi−lo)×(key−A[lo])/(A[hi]−A[lo])
  - Round to valid index
  - Guard equal endpoints
  - Guard out-of-range targets
  - Sorted, suitably uniform data
  - Expected probes: O(log log n)
  - Pathological distribution: linear
  - Binary search: logarithmic worst case
  - Binary search: no uniformity requirement
- Other sorted-search variants
  - Ternary: compare two split points
  - Retain relevant third
  - Ternary search: logarithmic
  - Jump search: block-boundary probes
  - Then scan selected block
  - Regular intervals: unit steps or jumps
  - Interpolation also requires sortedness
  - Prerequisite MCQ options may overlap

Sorting properties and cost comparison | Exam focus

- Sorting properties and comparison assumptions
  - Stability: preserve equal-key input order
  - In-place: fixed extra element storage
  - Total auxiliary space includes recursive stack
  - Stable/in-place alone identify no unique algorithm
  - Specify implementation, input case, resource
- Bubble sort with early stopping
  - Best: n
  - Average/worst: n^2
  - Constant extra storage

- Stable

- Selection sort profile
  - Best/average/worst: n^2
  - Constant extra storage
  - Unstable
- Insertion sort profile
  - Best: n
  - Average/worst: n^2
  - Constant extra storage
  - Stable
- Quicksort profile
  - Best/average: n log n
  - Worst: n^2
  - Average stack: log n
  - Worst stack: n
  - Unstable
- Ordinary array mergesort profile
  - All cases: n log n
  - Linear extra storage
  - Stable with left-first ties
- Heapsort profile
  - All cases: O(n log n) bound
  - Iterative sink: constant storage
  - Unstable
- Exam trap Overlapping bounds and logarithm placement
  - $n \log(n^2)=2n \log n$
  - Same growth as n log n
  - Different from $n(\log n)^2$
  - Big-O options can overlap

Bubble sort | Exam focus

- Bubble-sort adjacent passes
  - Ascending: swap only left>right

- Left-to-right pass: remaining maximum→end
- Shrink unsorted boundary each pass
- Up to n(n−1)/2 pair comparisons
- Average/worst: quadratic

• Bubble optimizations and stability
- Early stop: no-swap pass finishes
- Sorted input with early stop: linear
- Fixed pass schedule: quadratic
- Strict swaps preserve ties
- Cocktail sort: bidirectional bubble passes
- Cocktail stability: strict swaps

• Worked example Bubble first-pass trace
- [4,1,3,2]→[1,4,3,2]
- Then [1,3,4,2]
- Then [1,3,2,4]
- Maximum 4 reaches final position

Selection sort | Exam focus

• Selection-sort comparison and exchange counts
- Scan suffix minimum
- Exchange minimum with A[i]
- n−1 meaningful placements
- n(n−1)/2 comparisons regardless of order
- At most n−1 distinct-position exchanges
- Final self-swap: n swap calls possible

• Selection-sort suitability and instability
- Expensive record moves; cheap comparisons
- Large dataset alone insufficient
- [2a,2b,1]→[1,2b,2a]
- Equal-key order reversed
- Constant element storage: nonunique property

Insertion sort | Exam focus

- Insertion-sort sorted-prefix invariant
  - First element initially sorted
  - Save next value
  - Shift larger predecessors right
  - Insert into gap
  - n−1 insertion passes
  - Sorted input: one check, no shifts
  - Sorted time: linear
  - Reverse shifts: 1+2+…+(n−1)
  - Reverse time: quadratic
- Insertion-sort adaptiveness and stability
  - Few inversions: few shifts
  - Nearly sorted or small arrays
  - Strictly larger shifts preserve ties
  - Linked implementation avoids array shifts
  - Linked insertion position still requires search
- Worked example Insertion into sorted prefix
  - Insert 2 into [1,4,7]
  - Save 2
  - Shift 7 before 4
  - Write after 1
  - Result: [1,2,4,7]

Shell sort | Exam focus

- Shell-sort gap passes
  - Positive gaps ending at 1
  - Gap h: insertion-sort h-spaced subsequences
  - Earlier long moves reduce disorder
  - Final h=1 finishes sorting
  - h-sorted: $A[j] \le A[j+h]$
  - Both indices must exist
- Shell-sort complexity and stability

- Bounds depend on gap sequence
- Knuth increments: 1,4,13,...
- Knuth worst order: O(n^(3/2))
- Distant moves can reorder ties
- Generally unstable
- Constant extra element storage

• Worked example Shell-sort gap trace
- Start [5,4,3,2,1]
- Gap 3: positions (0,3),(1,4)
- After gap 3: [2,1,3,5,4]
- After gap 1: [1,2,3,4,5]

Quicksort partition rules | Exam focus

• Quicksort partition balance
- Choose pivot, partition, recurse
- Balanced partitions: n log n
- Repeated extremes: n+(n−1)+...
- Extreme-pivot time: quadratic
- Expected n log n: input/random-pivot model
- Random versus median-of-three tradeoffs
- No universally best pivot policy

• Partition invariants versus exact arrays
- Ten distinct values; final pivot 5
- Five smaller before; four larger after
- Internal order depends on partition algorithm
- Multiple arrays may satisfy invariant
- Name algorithm for exact trace
- Same-array rearrangement
- Worst recursive storage: linear

Mergesort and external sorting | Exam focus

• Merge-sort recurrence and stability
- Split halves, sort, merge

- Advance smaller current front
- Left-first ties preserve equal-key order
- Ordinary array buffer: linear
- Logarithmic levels × linear merge work
- Sorted input: n log n
- Merge-skipping optimization exception

• Merge variants and external sorting

- Top-down: recursive
- Bottom-up: iterative widths 1,2,4,…
- In-place merging variants: different tradeoffs
- Linear merge: operation, ambiguous sort label
- Internal sorting: dataset fits memory
- External sorting: sorted disk runs
- Buffered stream merges reduce I/O
- Reverse input ≠ maximum merge comparisons

Heapsort | Exam focus

• Ascending in-place heapsort trace

- Build max-heap
- Exchange root with final active element
- Shrink active heap
- Sink new root
- Removed maximum joins sorted suffix
- Repeat until one active item

• Heapsort costs and distinguishing mechanism

- Bottom-up build: linear
- Repeated extraction: O(n log n) worst
- Iterative sink: constant auxiliary storage
- Long root-to-end exchanges reorder ties
- Generally unstable
- Any comparison sort can compare priorities
- Heap extraction: distinctive mechanism

Radix sorting | Exam focus

- LSD radix sort
  - Fixed-width digit keys
  - Least-significant digit first
  - Move left with stable digit passes
  - Stability preserves earlier digit ordering
  - d digits; radix r
  - Counting passes: O(d(n+r))
  - Auxiliary storage: O(n+r)
- MSD radix and key assumptions
  - Leading digits first
  - Subdivide equal-prefix groups
  - Strings: lexicographic comparison
  - Accessible digits required
  - Specify length and signed-value conventions
  - Arbitrary unstructured keys unsuitable
  - Large radix/long keys/small n: comparison alternative

Graph models and representations | Exam focus

- Graph relationship models
  - Vertices V; edges E
  - Directed: u→v distinct from v→u
  - Following accounts: directed
  - Mutual friendship: undirected
  - Weighted edges: costs
  - Unweighted shortest paths: edge counts
  - DAG: directed acyclic graph
  - Trees: connected acyclic graphs
  - Graph structures may contain cycles
- Graph representation costs
  - Matrix space: $\Theta(V^2)$
  - Matrix edge lookup: O(1)
  - Adjacency-list space: $\Theta(V+E)$

- Lists enumerate neighbours
- Dense simple graphs: near-quadratic edges
- Every-edge work increases with density
- Vertex-only work needn't inspect every edge

Breadth-first search | Exam focus

- BFS discovery discipline
  - Mark source on enqueue
  - Dequeue; examine neighbours
  - Unvisited neighbour: mark and enqueue
  - Store predecessor[v] on discovery
  - Predecessors reconstruct paths
  - FIFO: distance d before later layers
- BFS guarantees and costs
  - Unweighted minimum-edge paths
  - No arbitrary minimum-weight guarantee
  - Adjacency lists: O(V+E)
  - Matrix scan: $O(V^2)$
  - Disconnected graph: restart at unvisited vertices
  - Neighbour order changes tied traversal
  - Minimum distances unchanged
- Worked example BFS alphabetical-neighbour trace
  - Edges A-B,A-C,B-D,C-D
  - Queue [A]→[B,C]→[C,D]→[D]→[]
  - Discovery: A,B,C,D
  - D distance: two

Depth-first search | Exam focus

- DFS latest-pending-work discipline
  - Mark vertex
  - Recursively finish each unmarked neighbour
  - Then continue later neighbours
  - Explicit stack alternative

- Queue changes discipline
- Marking prevents cyclic infinite repetition

• DFS guarantees, applications, costs
- Alphabetical recursive discovery: A,B,D,C
- Reachability witness, not necessarily shortest
- Adjacency lists: O(V+E)
- Visited state: O(V)
- Possible stack depth: O(V)
- Cycle and connectivity analysis
- Directed cycle test: active versus completed

Shortest paths and spanning trees | Exam focus

• Dijkstra shortest-path conditions
- Nonnegative weights
- Source distance 0; others ∞
- Finalize smallest unsettled distance
- Relax outgoing edges
- dist[v]=min(dist[v],dist[u]+w(u,v))
- Min-priority queue: next vertex
- Equal edge costs: BFS sufficient
- Negative edges: justified alternative
- Bellman-Ford alternative

• Minimum spanning tree objective
- Connected undirected weighted graph
- Minimum total connection cost
- Prim: cheapest current-tree crossing edge
- Kruskal: increasing edge weights
- Reject already-connected endpoints
- MST ≠ shortest source routes
- Equal weights: multiple MSTs possible

• Worked example Triangle MST and shortest paths
- AB=1; BC=2; AC=10
- Prim/Kruskal: AB,BC

- MST total: 3
- Dijkstra from A: B=1
- C: 10→3 through B

Kadane and KMP

- Kadane maximum contiguous subarray
  - Required nonempty answer
  - Initialize ending=best=A[0]
  - ending=max(x,ending+x)
  - best=max(best,ending)
  - Linear time
  - Constant working storage
  - best=0 silently allows empty answer
  - All-negative nonempty case fails
- Knuth-Morris-Pratt pattern matching
  - Pattern inside text
  - Preprocess longest proper prefix/suffix lengths
  - Mismatch: prefix-length fallback
  - Avoid restarting comparisons from beginning
  - Text pointer needn't move backward
  - Preprocessing+scan: O(pattern length+text length)
- Worked example Kadane and KMP worked values
  - Input [-2,3,-1,4,-5]
  - Ending values: −2,3,2,6,1
  - Best=6 from [3,−1,4]
  - ABAB prefix lengths: [0,0,1,2]

Missing data and misleading labels

- Trace completeness and answer-key limits
  - Complete operation sequence
  - Initial state
  - Capacity
  - Output orientation

- State assumptions before calculation
- Highlighted key cannot remove valid overlaps

• Implementation versus best-choice questions

- Can implement versus best under costs
- ADT permits multiple representations
- Serial-access memory ≠ inherently LIFO
- Heap file: unordered database records
- Heap file unrelated to binary heap
- Insertion: available space
- Lookup: scanning possible
- Ordered output: sorting or index
- Beginning insertion unnecessary