

# Algorithms - CheatSheet

IN BA4 - Ola Nils Anders Svensson

Notes by Ali EL AZDI

*This is a cheat sheet for the Algorithms midterm exam. For suggestions, contact me on Telegram (`elazdi.al`) or via EPFL email (`ali.elazdi@epfl.ch`).*

<b>Master Theorem</b> If $T(n) = \Theta\left(\frac{n}{b}\right) + f(n)$ , $a \geq 1$ , $b > 1$ , and $f(n)$ asymptotically positive. <b>Case 1:</b> If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$ , then $T(n) = \Theta(n^{\log_b a})$ . <b>Case 2:</b> If $f(n) = \Theta(n^{\log_b a})$ , then $T(n) = \Theta(n^{\log_b a} \log n)$ . <b>Case 3:</b> If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ , and if $a\left(\frac{n}{b}\right) \leq c f(n)$ for some $c < 1$ and all sufficiently large $n$ , then $T(n) = \Theta(f(n))$ . <b>Common case:</b> If $f(n) = \Theta(n^d)$ for some exponent $d$ : - If $\frac{a}{b} < 1$ (or $> \log_b a$ ), then $T(n) = \Theta(n^d)$ . - If $\frac{a}{b^d} = 1$ (or $d = \log_b a$ ), then $T(n) = \Theta(n^d \log n)$ . - If $\frac{a}{b^d} > 1$ (or $d < \log_b a$ ), then $T(n) = \Theta(n^{\log_b a})$ .	<b>Akra-Bazzi, Ali Najib Variation:</b> For recurrence $T(n) = \alpha f(n) + \beta T(bn) + \Theta(n^d)$ , where $\alpha, \beta > 0$ , $a \in (0, 1)$ , $d \geq 0$ , and unique $p$ with $\alpha a^p + \beta b^p = 1$ Then: $p > d \Rightarrow T(n) = \Theta(n^p),$ $p = d \Rightarrow T(n) = \Theta(n^d \log n),$ $p < d \Rightarrow T(n) = \Theta(d)$ . <b>If <math>p</math> is not easily found, compare <math>\alpha a^d + \beta b^d</math> with 1:</b> $\alpha a^d + \beta b^d < 1 \Rightarrow T(n) = \Theta(n^d),$ $\alpha a^d + \beta b^d = 1 \Rightarrow T(n) = \Theta(n^d \log n),$ $\alpha a^d + \beta b^d > 1 \Rightarrow T(n) = \Theta(n^p),$ with $p$ determined by $\alpha a^p + \beta b^p = 1$ .	<b>Big-O</b> If $\exists c > 0$ and $\exists n_0 > 0$ , $0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0$ , $f(n) = O(g(n))$ . <b>Big-Omega</b> If $\exists c > 0$ and $\exists n_0 > 0$ , $0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0$ , $f(n) = \Omega(g(n))$ . <b>Big-Theta</b> If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ , $f(n) = \Theta(g(n))$ . <b>Little-o</b> If $\forall c > 0 \exists n_0 > 0$ , $0 \leq f(n) < c \cdot g(n) \forall n \geq n_0$ , $f(n) = o(g(n))$ . <b>Relations</b> $f(n) = o(g(n)) \implies f(n) = O(g(n))$ <b>Comparison of Common Functions (Ascending Order)</b> $O(1) \ll O((\log n)^c) \ll O(n^c)_{0 \leq c < 1} \ll O(n) \ll O(n \log n) \ll O(n^c)_{c > 1} \ll O(n^c) \ll O(n^{\log n}) \ll O(n!)$			
<b>Queue Operations</b> <b>Queue-Empty(Q): Time: <math>O(1)</math>, Auxiliary Space: <math>O(1)</math></b> 1. Returns TRUE if the queue is empty ( $Q.\text{head} = Q.\text{tail}$ ). 2. Returns FALSE otherwise. <b>Enqueue(Q, x): Time: <math>O(1)</math>, Auxiliary Space: <math>O(1)</math></b> 1. Adds element $x$ to the rear of queue $Q$ . 2. $Q.Q.\text{tail} = x$ 3. $Q.Q.\text{tail} = Q.Q.\text{tail} + 1$ (or wrap around if using circular array) <b>Dequeue(Q): Time: <math>O(1)</math>, Auxiliary Space: <math>O(1)</math></b> 1. If Queue-Empty( $Q$ ), return error "underflow". 2. Otherwise, remove and return the element at the front. 3. $x = Q.Q.\text{head}$ 4. $Q.Q.\text{head} = Q.Q.\text{head} + 1$ (or wrap around) 5. Return $x$ <b>Queue Implementation:</b> 1. $Q.\text{head}$ : Index of the front element 2. $Q.\text{tail}$ : Index where next element will be inserted 3. In a circular array, indices wrap around 4. Leave one slot empty to distinguish full/empty states <b>Overall Space Complexity:</b> $O(n)$ for a queue of capacity $n$	<b>Insertion Sort.</b> <b>1 - Select the key</b> Begin with the second element (at index 1) as the key. <b>INSERTION-SORT(A, n)</b> <pre>for i = 2 to n do     key = A[i] // Insert A[i] in the sorted seq.     A[i-1]..i-1     i = i - 1     while i &gt; 0 and A[i] &gt; key do         A[i+1] = A[i]         i = i - 1     A[i+1] = key</pre> <b>2 - Compare and Shift</b> Compare the key with elements in the sorted section (to its left). <b>3 - Shift Elements</b> If an element is greater than the key, shift that element one position to the right. <b>4 - Insert the Key</b> Once an element less than or equal to the key is found (or you reach the start), insert the key immediately after that element. <b>5 - Repeat</b> Move forward to the next element, treating it as the new key, and repeat until the array is sorted. <b>Time Complexity:</b> Worst-case $O(n^2)$ , Best-case $O(n)$ . <b>Space Complexity:</b> $O(1)$ .	<b>Heap</b> Root is A[1] Left(i) = 2i Right(i) = 2i + 1 Parent(i) = [i/2] <b>Max-Heapify(A, i, n)</b> <pre>l ← LEFT(i) r ← RIGHT(i) if l ≤ n ∧ A[l] &gt; A[i] then     largest ← l else     largest ← i     if l ≤ n ∧ A[r] &gt; A[largest] then         exchange A[i] with A[largest]         Max-Heapify(A, largest, n)     n ← n - 1 A[n] ← -∞ HEAP-INCREASE-KEY(A, n, key)</pre> <b>Max-Heap (Build a max-heap from an array)</b> 1. Start from the last non-leaf node at index $\frac{n}{2} - 1$ 2. Move upwards to the root (index 0) and: a. <b>Max-Heapify</b> the current node b. Ensure the subtree rooted here satisfies max-heap property 3. Repeat until the root node is processed 4. After completion, array A represents a valid max heap <b>Time Complexity:</b> $O(n \log n)$ <b>Space Complexity:</b> $O(n)$ including heap array, $O(1)$ auxiliary <b>Max-Heap-Increase-Key (Increase key at position i)</b> 1. Ensure new key is larger than current: if $key < A[i]$ then error 2. Set $A[i] = key$ 3. Compare with parent and swap if necessary: while $i > 1$ and $A[\text{Parent}(i)] < A[i]$ do 4. Exchange $A[i]$ with $A[\text{Parent}(i)]$ 5. Set $i = \text{Parent}(i)$ and continue upward <b>Time Complexity:</b> $O(\log n)$ <b>Space Complexity:</b> $O(n)$ including heap array, $O(1)$ auxiliary <b>Max-Heap-Increase-Key (Increase key at position i)</b> 1. Ensure new key is larger than current: if $key < A[i]$ then error 2. Set $A[i] = key$ 3. Compare with parent and swap if necessary: while $i > 1$ and $A[\text{Parent}(i)] < A[i]$ do 4. Exchange $A[i]$ with $A[\text{Parent}(i)]$ 5. Set $i = \text{Parent}(i)$ and continue upward <b>Time Complexity:</b> $O(\log n)$ <b>Space Complexity:</b> $O(n)$ including heap array, $O(1)$ auxiliary <b>Build-Max-Heap (build a max-heap from an array)</b> 1. Start from the last non-leaf node at index $\frac{n}{2} - 1$ 2. Move upwards to the root (index 0) and: a. <b>Max-Heapify</b> the current node b. Ensure the subtree rooted here satisfies max-heap property 3. Repeat until the root node is processed 4. After completion, array A represents a valid max heap <b>Time Complexity:</b> $O(n)$ <b>Space Complexity:</b> $O(n)$ including heap array, $O(1)$ auxiliary <b>Heap Sort</b> <b>BUILD-MAX-HEAP(A, n)</b> for i ← $[n/2]$ to 1 do MAX-HEAPIFY(A, i, n)			
<b>Merge Sort</b> 1. <b>Divide:</b> Split the array evenly into two smaller subarrays, and continue dividing recursively. 2. <b>Sort (Recursively):</b> Apply merge sort recursively on each subarray until each has only one element (base case). <b>MERGE-SORT(A, p, q, r)</b> If $p < r$ q = $\lfloor (p+r)/2 \rfloor$ MERGE-SORT(A, p, q) MERGE-SORT(A, q+1, r) MERGE(A, p, q, r) 3. <b>Merge:</b> Combine the two sorted subarrays into a single sorted array: a) Pointing pointers at the start of each subarray. b) Comparing the elements pointed to, and appending the smaller one into a new array. c) Advancing the pointer in the subarray from which the element was chosen. d) Repeating this process until all elements in both subarrays are merged into the sorted array. <b>Merge Cost Complexity:</b> $O(n)$ per merge operation. <b>Time Complexity:</b> $O(n \log n)$ <b>Space Complexity:</b> $O(n)$	<b>MERGE(A, p, q, r)</b> MERGE-SORT(A, p, q, r) if $p < r$ q = $\lfloor (p+r)/2 \rfloor$ MERGE-SORT(A, p, q) MERGE-SORT(A, q+1, r) MERGE(A, p, q, r) 3. Merge: Combine the two sorted subarrays into a single sorted array: a) Pointing pointers at the start of each subarray. b) Comparing the elements pointed to, and appending the smaller one into a new array. c) Advancing the pointer in the subarray from which the element was chosen. d) Repeating this process until all elements in both subarrays are merged into the sorted array. <b>Merge Cost Complexity:</b> $O(n)$ per merge operation. <b>Time Complexity:</b> $O(n \log n)$ <b>Space Complexity:</b> $O(n)$	<b>Strassen's Matrix Multiplication</b> <b>Divide:</b> Partition each of $A, B, C$ into four $\frac{n}{2} \times \frac{n}{2}$ submatrices: $(C_{11} \quad C_{12}) = (A_{11} \quad A_{12}) \cdot (B_{11} \quad B_{12}),$ $(C_{21} \quad C_{22}) = (A_{21} \quad A_{22}) \cdot (B_{21} \quad B_{22}).$ <b>Conquer:</b> Compute 7 products (recursively on $\frac{n}{2} \times \frac{n}{2}$ matrices): $M_1 := (A_{11} + A_{22})(B_{11} + B_{22}),$ $M_2 := (A_{21} + A_{22})(B_{11}),$ $M_3 := A_{11}(B_{12} - B_{22}),$ $M_4 := A_{22}(B_{21} - B_{11}),$ $M_5 := (A_{11} + A_{12})B_{22},$ $M_6 := (A_{21} - A_{11})(B_{11} + B_{12}),$ $M_7 := (A_{12} - A_{22})(B_{21} + B_{22}).$ <b>Combine:</b> Assemble the resulting submatrices to form $C$ : $C_{11} = M_1 + M_4 - M_5 + M_7,$ $C_{21} = M_2 + M_4,$ $C_{12} = M_3 + M_5,$ $C_{22} = M_1 + M_3 - M_2 + M_6.$ <b>Time Complexity:</b> $O(n^{\log 7}) \approx O(n^{2.81})$ <b>Space Complexity:</b> $O(n^2)$	<b>Binary Search Trees (BST)</b> <b>BST-Search</b> 1. Start at root 2. If NULL, return NULL 3. If key = root's key, return root 4. If key < root's key, search left 5. If key > root's key, search right <b>BST-Postorder</b> 1. Recursively traverse left 2. Recursively traverse right 3. Visit current node (Children first, then root) <b>POSTORDER-TREE-WALK(x)</b> if $x \neq \text{NIL}$ POSTORDER-TREE-WALK(x.left) POSTORDER-TREE-WALK(x.right) print key[x] <b>Time: <math>O(n)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(h)$ auxiliary <b>BST-Inorder</b> 1. Recursively traverse left 2. Visit current node 3. Recursively traverse right (Visits nodes in sorted order) <b>INORDER-TREE-WALK(x)</b> if $x \neq \text{NIL}$ INORDER-TREE-WALK(x.left) print key[x] INORDER-TREE-WALK(x.right) <b>Time: <math>O(n)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(h)$ auxiliary <b>BST-Minimum</b> 1. Start at root 2. If NULL, return NULL 3. Follow left pointers until no left child 4. Return leftmost node <b>Tree-MINIMUM(x)</b> while x.left ≠ NIL do x ← x.left <b>Time: <math>O(1)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(1)$ auxiliary <b>BST-Maximum</b> 1. Start at root 2. If NULL, return NULL 3. Follow right pointers until no right child 4. Return rightmost node <b>Tree-MAXIMUM(x)</b> while x.right ≠ NIL do x ← x.right <b>Time: <math>O(1)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(1)$ auxiliary <b>BST-Preorder</b> 1. Visit current node 2. Recursively traverse left 3. Recursively traverse right <b>PREORDER-TREE-WALK(x)</b> if $x \neq \text{NIL}$ PREORDER-TREE-WALK(x.left) print key[x] PREORDER-TREE-WALK(x.right) <b>Time: <math>O(n)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(h)$ auxiliary <b>BST-Transplant</b> Replace subtree at $u$ with $v$ : 1. If $u$ is root, set $v$ as root 2. If $u$ is left child, make $v$ left child of $u$ 's parent 3. If $u$ is right child, make $v$ right child of $u$ 's parent 4. Set $v$ 's parent to $u$ 's parent <b>TRANSPLANT(T, u, v)</b> if $u \neq \text{NIL}$ T.root ← v else if $v \neq \text{NIL}$ u.parent.left ← v else u.parent.right ← v <b>Time: <math>O(1)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(1)$ auxiliary <b>BST-Successor</b> 1. If right subtree exists: Return minimum in right subtree 2. Otherwise: Find first ancestor where node is in left subtree 3. Traverse the rightmost node <b>Tree-SUCCESSOR(x)</b> if x.right ≠ NIL return Tree-MINIMUM(x.right) else x ← x.parent while x.parent.left ≠ x do x ← x.parent return x.parent <b>Time: <math>O(1)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(1)$ auxiliary <b>BST-Delete</b> 1. Create new node $z$ with key 2. Start at root, track parent $y = \text{NIL}$ 3. Move down tree (left if key $< y$ , right if key $> y$ ) 4. Once NULL found, link $z$ as child of $y$ 5. If $y$ is NIL, $z$ becomes root 6. Otherwise, insert $z$ as left or right child based on key comparison <b>TREE-INSERT(T, z)</b> y ← x.p while x ≠ NIL do if x.key < z.key x = x.right else if x.key > z.key x = x.left else x.p.right ← z if x.p ≠ NIL x.p.right ← x x.p = z z.parent = x.p <b>Time: <math>O(n)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(1)$ auxiliary <b>BST-Delete</b> 1. If $z$ has no left: transplant right 2. If $z$ has no right: transplant left 3. With both children: a. Find successor $y$ b. Handle $y$ 's children c. Replace $z$ with $y$ <b>TREE-DELETE(T, z)</b> if $z.left = \text{NIL}$ TRANSPLANT(T, z, z.right) // z has no left child else if $z.right = \text{NIL}$ TRANSPLANT(T, z, z.left) // z has just a left child else // z has two children y = TREE-MINIMUM(z.right) // y is z's successor if y ≠ z TRANSPLANT(T, z, y) else if z.key = y.key y.left ← z.left y.left.parent = y else z.right ← y z.right.parent = y <b>Time: <math>O(h)</math></b> <b>Space: <math>O(n)</math></b> for tree, $O(1)$ auxiliary <b>Properties:</b> - Left subtree: all keys $<$ node's key - Right subtree: all keys $>$ node's key - Left and right subtrees are also BSTs - A Node has: key (value), left & right (child pointers), parent (optional) - Tree height h: length of longest path from root to leaf	<b>FIND-MAX-SUBARRAY(A, low, high)</b> if high == low return (low, high, A[low]) else mid = $\lfloor (low + high)/2 \rfloor$ (left-low, right-high, left-sum) = FIND-MAX-SUBARRAY(A, low, mid) (right-low, right-high, right-sum) = FIND-MAX-SUBARRAY(A, mid+1, high) low, mid, high if left-sum ≥ right-sum and left-sum ≥ cross-sum return (left-low, left-high, left-sum) else if right-sum ≥ left-sum and right-sum ≥ cross-sum return (right-low, right-high, right-sum) <b>FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)</b> mid = $\lfloor (low + high)/2 \rfloor$ (left-low, right-low, left-high, left-sum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid) (right-low, right-high, right-sum) = FIND-MAX-CROSSING-SUBARRAY(A, mid+1, high) <b>FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)</b> left-low ← $\infty$ , sum ← 0 for i = mid to low do sum ← sum + A[i] if sum > left-low left-low ← sum, max-left ← i right-low ← $\infty$ , sum ← 0 for j = mid + 1 to high do sum ← sum + A[j] if sum > right-low right-low ← sum, max-right ← j return (max-left, max-right, left-sum + right-sum)	<b>HEAPSORT(A, n)</b> <b>BUILD-MAX-HEAP(A, n)</b> for i = $[n/2]$ to 1 do MAX-HEAPIFY(A, i, n)
<b>Dynamic Programming</b> <b>Problem:</b> Optimal solutions to overlapping problems <b>Cut-Rod Problem:</b> Find optimal way to cut rod to maximize revenue	<b>Top-Down (Memoization):</b> 1. Create memo array $r[0..n]$ initialized to NIL 2. For uncalculated $r[j]$ , compute: $r[j] = \max_{1 \leq i \leq j} (p[i] + r[j-i])$ 3. Return $r[n]$ <b>Bottom-Up (Tabulation):</b> 1. Create array $r[0..n]$ with $r[0] = 0$ 2. For $j = 1$ to $n$ : - Compute $r[j] = \max_{1 \leq i \leq j} (p[i] + r[j-i])$ - Return $r[n]$ <b>EXTENDED-BOTTOM-UP-CUT-ROD(p, n)</b> let $r[0..n]$ and $s[0..n]$ be new arrays $r[0] = 0$ for $i = 1$ to $n$ do $r[i] \leftarrow r[i-1] + r[i-1]$ for $j = 2$ to $n$ do $r[i] \leftarrow r[i-1] + r[i-2]$ $r[i] \leftarrow r[i-1] + r[i-2]$ $r[i] \leftarrow q$ return $r[n]$	<b>Time Complexity:</b> $O(n)$ <b>Space Complexity:</b> $O(n)$ (includes input and memo array)			
<b>Bottom-Up (Tabulation):</b> 1. Create array $r[0..n]$ 2. For $j = 1$ to $n$ : - Compute $r[j] = \max_{1 \leq i \leq j} (p[i] + r[j-i])$ - Return $r[n]$ <b>EXTENDED-BOTTOM-UP-CUT-ROD(p, n)</b> let $r[0..n]$ and $s[0..n]$ be new arrays $r[0] = 0$ for $i = 1$ to $n$ do $r[i] \leftarrow r[i-1] + r[i-1]$ for $j = 2$ to $n$ do $r[i] \leftarrow r[i-1] + r[i-2]$ $r[i] \leftarrow r[i-1] + r[i-2]$ $r[i] \leftarrow q$ return $r[n]$	<b>Time Complexity:</b> $O(n^2)$ <b>Space Complexity:</b> $O(n)$ (includes input prices and array)				
<b>Bottom-Up (Tabulation):</b> 1. Create array $F[0..n]$ 2. Base cases: $F[0] = 0, F[1] = 1$ 3. For $i = 2$ to $n$ : - Compute $F[i] = F[i-1] + F[i-2]$ - Return $F[n]$ <b>BOTTOM-UP-FIB(n)</b> Let $r$ be a new array of size $n+1$ $r[0] \leftarrow 0$ $r[1] \leftarrow 1$ for $i = 2$ to $n$ do $r[i] \leftarrow r[i-1] + r[i-2]$ return $r[n]$	<b>Time Complexity:</b> $O(n)$ <b>Space Complexity:</b> $O(n)$				
<b>Time Complexity:</b> $O(n)$ <b>Space Complexity:</b> $O(n)$	<b>Time Complexity:</b> $O(n^2)$ <b>Space Complexity:</b> $O(n)$				
<b>Data Structure Operations Summary</b> <b>Priority Queue Operations</b>	<b>Operation</b> Construction Maximum(S) Extract-Max(S) Insert(S, x) Increase-Key(S, x, k) <b>Stack Operations</b> Operation Construction Stack-Empty(S) Push(S, x) Pop(S) <b>Heap Operations</b> Operation Construction Max-Heapify(A, i) Build-Max-Heap(A, n) Build-Max-Heap(A, n, key) Heap-Sort(A, n) Max-Heap-Insert(A, k) Heap-Extract-Max(A) Heap-Increase-Key(A, k, k) <b>BST Operations</b> Operation Construction BST-Search(T, k) BST-Minimum(T) BST-Maximum(T) BST-Predecessor(x, k) BST-Postorder(T) <b>Queue Operations</b> Operation Construction Queue-Empty(Q) Enqueue(Q, x) Dequeue(Q) <b>Linked List Operations</b> Operation Construction List-Search(L, k) List-Insert(L, x) List-Delete(L, x)	<b>Description</b> Create priority queue from array of $n$ elements Returns element with highest priority Removes and returns element with highest priority Inserts element $x$ into set $S$ Increases priority of element $x$ to $k$ <b>Description</b> Create stack from array of $n$ elements Returns TRUE if stack is empty, FALSE otherwise Adds element $x$ to top of stack $S$ Removes and returns top element from stack $S$ <b>Description</b> Create heap from array of $n$ elements Maintains max-heap property at node $i$ Converts array $A$ of $n$ elements into max heap Sorts array $A$ of $n$ elements using heap structure Inserts key $k$ into heap $A$ Returns and removes largest element from heap $A$ Increases key at index $i$ to new value $k$ <b>Description</b> Create BST from array of $n$ elements Best: $O(n \log n)$ , Worst: $O(n^2)$ Finds node with key $k$ in tree $T$ Returns node with smallest key in $T$ Returns node with largest key in $T$ Returns node with smallest key greater than $x$ 's key Inserts node $z$ into BST $T$ Removes node $z$ from BST $T$ Visits all nodes in sorted order Visits root before its children Visits children before root <b>Description</b> Create queue from array of $n$ elements Returns TRUE if queue is empty, FALSE otherwise Adds element $x$ to rear of queue $Q$ Removes and returns front element from queue $Q$ <b>Description</b> Create linked list from array of $n$ elements Returns pointer to first node with key $k$ , NULL if not found Inserts node $x$ at beginning of list $L$ Removes node $x$ from list $L$ <b>Description</b> MAX-HEAPIFY(A, i, n) if $i \leq n$ and $A[i] > A[i]$ then largest ← $i$ if $i \leq n$ and $A[i] > A[largest]$ then exchange A[i] with A[largest] MAX-HEAPIFY(A, largest, n) n ← n - 1 A[n] ← -∞ HEAP-INCREASE-KEY(A, n, key) <b>HEAP-INCREASE-KEY(A, i, key)</b> if $key < A[i]$ error "new key is smaller than current key" A[i] ← key while $i > 1$ and $A[\text{Parent}(i)] < A[i]$ do exchange A[i] with A[Parent(i)] i ← Parent(i) <b>BUILD-MAX-HEAP(A, n)</b> for i ← $[n/2]$ to 1 do MAX-HEAPIFY(A, i, n)			

**Optimal Binary Search Tree Problem:** Construct a BST with minimum expected search cost given access probabilities

**Optimal-BST Algorithm:** - Input: Keys  $K_1, \dots, K_n$ , probabilities  $p_1, \dots, p_n$  for successful searches

- Optional: Probabilities  $q_0, \dots, q_n$  for unsuccessful searches
- Create tables  $e[1..n+1, 0..n]$  for expected costs
- Create  $w[i..j]$  for sum of probabilities from  $i$  to  $j$
- Create  $root[i..j]$  to record optimal roots
- Fill tables bottom-up by increasing subproblem size
- For each subproblem, try all possible roots and pick the minimum cost
- Formula:  $e[i..j] = \min_{l \leq r \leq j} \{e[i..r-1] + e[r+1..j] + w[i..j]\}$

- The root of the overall optimal tree is in  $root[1..n]$

OPTIMAL-BST( $p, q, n$ )

```

let  $e[1..n+1, 0..n], w[1..n+1, 0..n]$ , and  $root[1..n, 1..n]$  be new tables
for  $i = 1$  to  $n+1$  do
   $e[i..i-1] \leftarrow 0$ 
   $w[i..i-1] \leftarrow 0$ 
for  $i = 1$  to  $n$  do
  for  $j = i+1$  to  $n+1$  do
     $f \leftarrow i..j-1$ 
     $e[i..j] \leftarrow \infty$ 
     $w[i..j] \leftarrow w[i..j-1] + p_j$ 
    for  $r = i$  to  $j$  do
       $l \leftarrow e[i..r-1] + e[r+1..j] + w[i..j]$ 
      if  $r < e[i..j]$ 
         $e[i..j] \leftarrow l$ 
         $w[i..j] \leftarrow l$ 
         $root[i..j] \leftarrow r$ 
      return  $e, root$ 

```

**Complexity:**

- Time:  $O(n^3)$
- Space:  $O(n^2)$  for tables  $e$ ,  $w$ , and  $root$
- The algorithm computes optimal costs for all possible subtrees

### Matrix Chain Multiplication:

Find optimal parenthesization to minimize multiplications

**Bottom-Up (Tabulation):**

1. Create table  $m[1..n, 1..n]$  with  $m[i..j] = 0$
- $m[i..j]$  stores minimal cost of multiplying matrices  $i$  through  $j$
2. For  $i = 1$  to  $n$  (chain length):
3. For  $i = 1$  to  $n - l + 1$ :
- Set  $j = i + l - 1$
- Compute  $m[i..j] = \min_{l \leq k \leq j} \{m[i..k] + m[k+1..j] + p_{i-1} p_k p_j\}$
- Store  $k$  in  $s[i..j]$  that achieved minimum cost
4. Return  $m[1..n]$

MATRIX-CHAIN-ORDER( $p$ )

```

 $n = p.length - 1$ 
let  $m[1..n, 1..n]$  and  $s[1..n, 1..n]$  be new tables
for  $i = 1$  to  $n$  do
   $m[i..i] = 0$ 
for  $\ell = 2$  to  $n$  do //  $\ell$  is the chain length
  for  $i = 1$  to  $n - \ell + 1$  do
     $j \leftarrow i + \ell - 1$ 
     $m[i..j] \leftarrow \infty$ 
    for  $k = i + 1$  to  $j - 1$  do
       $q = m[i..k] + m[k+1..j] + p_{i-1} p_k p_j$ 
      if  $q < m[i..j]$ 
         $m[i..j] \leftarrow q$ 
         $s[i..j] \leftarrow k$ 
    return  $m, s$ 

```

**Complexity:**

- Time:  $O(n^3)$
- Space:  $O(n^2)$  (includes matrix dimensions and tables)
- Table  $s[i..j]$  stores index of last matrix in first parenthesized group

### Linked List

Linear data structure where each node contains:

1. key/data: The value stored in the node
  2. next: A pointer to the next node in the sequence
  3. prev: A pointer to the previous node (in doubly linked lists)
- List-Insert (insert a new node at the List-Delete beginning)**
- (remove a node from the list)
  1. Create a new node with the given  $i$ . Find the node to be deleted (may require traversal).
  2. Set the next pointer of the new node 2. If the node is the head, update the head pointer to the next node.
  3. If implementing a doubly linked list, 3. Otherwise, update the next pointer set the prev pointer of the current head of the previous node to skip the node to the new node.
  4. Update the head pointer to point to 4. For doubly linked lists, also update the new node.
  5. If the list was empty, update the tail 5. Free the memory allocated for the pointer as well.

```

LIST-INSERT(x, L)
x.next \leftarrow L.head
if L.head \neq nil
  L.head.prev \leftarrow x
L.head \leftarrow x
x.prev \leftarrow NIL

```

**Time Complexity:**  $O(1)$

**Space Complexity:**  $O(1)$

### List-Search (find a node with a given key)

1. Start from the head of the linked list.
2. Traverse the list by following the next pointers.
3. Compare each node's key with the target key.
4. Return the node if the key is found.
5. Return NULL if the end of the list is reached without finding the key

**Time Complexity:**  $O(n)$

**Space Complexity:**  $O(1)$

### Disjoint Set.

A disjoint set is a collection of sets where each set is disjoint from the others.

#### Disjoint Set Operations.

- **Make-Set( $x$ ):** Create a new set with a single element  $x$ .
- **Find( $x$ ):** Return the representative of the set containing  $x$ .
- **Union( $x, y$ ):** Merge the sets containing  $x$  and  $y$  into a single set.

if  $x \in S_x, y \in S_y$ , then  $S = S_x - S_y \cup S_x \cup S_y$

#### Linked List Representation.

- Each set is a single linked list represented by a set object that has
- a pointer to the head of the list (assumed to be the representative)
  - a pointer to the tail of the list
  - Each object in the list has attributes for the set member, pointer to the set object and next

#### Operations:

- **Make-Set( $x$ ):** Create a singleton list in time  $\Theta(1)$
- **Find( $x$ ):** follow the pointer back to the list object, and then follow the head pointer to the representative (time  $\Theta(1)$ )
- **Union( $x, y$ ):**

**Method 1 - Append  $y$ 's list onto the end of  $x$ 's list.** Use  $x$ 's tail pointer to find the end.

**Cons:**

- a. Need to update the pointer back to the set object for every node on  $y$ 's list.
- b. If appending a large list onto a small list, it can take a while

or

#### Method 2 - Weighted-union heuristic.

Always append the smaller list to the larger list (break ties arbitrarily, for m operations on n elements, time  $\Theta(m + n \log n)$ )

#### Disjoint-Set Forest (Union-Find)

A disjoint-set forest represents each set as a rooted tree. Each node stores a parent pointer; the root of the tree is the *representative*.

- **Make-Set( $x$ ):** create a singleton node.
- **Find-Set( $x$ ):** follow parent pointers to the root. *Path compression:* after the search, point every node on the path directly to the root.
- **Union( $x, y$ ):** make one root a child of another. *Union-by-rank:* always hang the shallower tree under the deeper one (break ties by increasing the new root's rank).

*Complexity with both heuristics:* a sequence of  $m$  operations on  $n$  elements runs in  $O(m \alpha(n))$  time. For all practical sizes  $\alpha(n) \leq 5$ .

MAKE-SET( $x$ )

```

x.p \leftarrow x
x.rank \leftarrow 0

```

FIND-SET( $x$ )

```

if x \neq x.p
  x.p \leftarrow FIND-SET(x.p)
return x.p

```

UNION( $x, y$ )

```

LINK(FIND-SET(x), FIND-SET(y))

```

LINK( $x, y$ )

```

if x.rank > y.rank
  y.p \leftarrow x
else
  x.p \leftarrow y
  if x.rank == y.rank
    y.rank \leftarrow y.rank + 1

```

**Flow Network.** A directed graph  $G = (V, E)$  with a source  $s$  and sink  $t$ . Each edge  $(u, v)$  has a capacity  $c(u, v) \geq 0$ .

**Flow Function.** A function  $f: V \times V \rightarrow \mathbb{R}$  satisfying:

- Capacity constraint:  $0 \leq f(u, v) \leq c(u, v)$

- Flow conservation:  $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$  for all  $u \in V \setminus \{s, t\}$ . (flow in = flow out)

**Value of a Flow.** Total flow out of the source:  $|f| = \sum_{v \in V} f(s, v)$ .

**Max-Flow Problem (What are we maximizing?)** Choose a flow  $f$  that maximizes its value  $|f|$  (the total amount of flow that leaves  $s$  and reaches  $t$ ) subject to the capacity and conservation constraints above.

**Algorithm (Ford-Fulkerson).** Start with 0-flow

- while there is an augmenting path from  $s$  to  $t$  in residual network do
- 1. Find augmenting path
- 2. Compute bottleneck = min capacity on path
- 3. Increase flow on the path by the bottleneck
- When finished, resulting flow is maximal

**Cut.**

A cut is a partition  $(S, T)$  of  $V$  such that  $s \in S$  and  $t \in T$ . Capacity of a cut:  $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$  (sum of capacities of all edges that cross from  $S$  to  $T$ ).

**Min-Cut Problem (What are we minimizing?)** Find the cut  $(S, T)$  that minimizes  $c(S, T)$  — the smallest total capacity that disconnects  $s$  from  $t$ .

**Algorithm.** If no augmenting path exists in residual network,

1. Find set of nodes  $S$  reachable from  $s$  in residual network
2. Set  $T = V \setminus S$
- $S$  and  $T$  define a minimum cut

### Longest Common Subsequence

**Problem:** Find the longest subsequence common to two sequences

**LCS-Length:**

- Build tables for length  $c[0..m, 0..n]$  and direction  $b[1..m, 1..n]$
- Initialize first row and column to zeros
- For each cell  $(i, j)$  in the table:

If characters match, take diagonal value + 1

Otherwise, take maximum from above or left

- Table  $c[m..n]$  contains the LCS length

- Table  $b$  records decisions for reconstruction

LCS-LENGTH( $X, Y, m, n$ )

```

let b[1..m, 1..n] = 0
let c[1..m, 1..n] = 0
for i = 1 to m do
  c[i..i] = 0
for j = 1 to n do
  c[1..j] = 0
c[0..0] = 0
for i = 1 to m do
  for j = 1 to n do
    if X[i] == Y[j]
      b[i..j] = c[i..j-1] + 1
    else
      c[i..j] = max(c[i..j-1], c[i-1..j])
      if b[i..j-1] >= c[i..j-1]
        b[i..j] = b[i..j-1]
      else
        b[i..j] = c[i..j-1]
      c[i..j] = c[i..j-1]
    print b
  
```

return c, b

**Time Complexity:**  $O(mn)$  for two sequences of lengths  $m$  and  $n$

**Space Complexity:**  $O(mn)$  (includes input sequences and tables)

Space can be optimized to  $O(m+n)$  if only length is needed

Print-LCS takes  $O(m+n)$  time to reconstruct the solution

**Print-LCS:**

```

PRINT-LCS(b, X, i, j)
if i == j == 0 return
if b[i..j] == \infty
  PRINT-LCS(b, X, i-1, j-1)
  print xi
else if b[i..j] == \infty
  PRINT-LCS(b, X, i-1, j)
else
  PRINT-LCS(b, X, i, j-1)

```

**Example Recording of Solution:**

i	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	1
2	0	0	1	1	1	1	1
3	0	0	1	1	1	2	2
4	0	1	1	1	1	2	2
5	0	1	1	1	1	2	3
6	0	1	2	2	2	2	3

### Graph.

A graph  $G = (V, E)$  consists of a vertex set  $V$  and an edge set  $E$  that contain (ordered) pairs of vertices.

**In-degree:** Number of edges coming into vertex  $v$ .

**Out-degree:** Number of edges going out from vertex  $v$ .

**Breadth-First Search (BFS):**

**Goal:**

Find all vertices reachable from a starting vertex  $s$ , visiting neighbors level by level (like ripples in water).

**How it works:**

- Initialize all vertices as WHITE (unvisited).

- For each WHITE vertex, start DFS-Visit:

- Mark current vertex as GRAY (being processed), record discovery time.

- Recursively visit each WHITE neighbor (ignore GRAY/BLACK).

- When done with all neighbors, mark vertex as BLACK (finished), record finish time.

**Key Data:** Discovery and finish times are kept for each node in  $DFS$ . These times help determine ancestor-descendant relationships and can be used for topological sorting.

**DFS-Visit Time Comp.  $O(E)$**

**DFS-Visit Space Comp.  $O(V)$**

```

DFS(G)
for all u \in G do
  u.color \leftarrow WHITE
  u.time \leftarrow -1
  u.d \leftarrow \infty
for all u \in G do
  if u.color == WHITE
    DFS-Visit(G, u)
DFS-Visit(G, u)
  if u.color == WHITE
    u.time \leftarrow time + 1
    u.d \leftarrow time
    u.color \leftarrow GRAY
    DFS-Visit(G, u)
    u.time \leftarrow time + 1
    u.d \leftarrow time
    u.f \leftarrow time // finish u
  
```

**Definition - Adjacency list.**

Array of linked lists storing neighbors for each vertex

**Space Complexity:**  $\Theta(V + E)$

**Time Complexity:** to list all vertices adjacent to  $u$ :  $\Theta(\deg(u))$

**Time Complexity:** to determine whether  $(u, v) \in E$ :  $\Theta(1)$

**Definition - Adjacency matrix.**

2D matrix where entry  $(i, j)$  indicates edge from vertex  $i$  to  $j$

**Space Complexity:**  $\Theta(V^2)$

**Time Complexity:** to list all vertices adjacent to  $u$ :  $\Theta(V)$

**Time Complexity:** to determine whether  $(u, v) \in E$ :  $\Theta(1)$

**Directed Acyclic Graph (DAG)**

A directed graph with no cycles. A directed graph  $G$  is a DAG if and only if  $G$  of  $DFS$  yields no back edges.

**Algorithm - Topological Sort.**

Linear ordering of vertices where for each edge  $(u, v)$ ,  $u$  comes before  $v$ . Possible only on Directed Acyclic Graphs (DAGs).

**Topological-Sort( $G$ ):**

1. Call DFS( $G$ ) to compute finishing times  $v.f$  for all  $v \in V$ .

2. Compute  $G^T$

3. Call  $DFS(G^T)$  but in the main loop, consider vertices in order of decreasing  $u.f$  (as computed in first DFS).

4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

**Time Complexity:**  $O(V+E)$

**Space Complexity:**  $O(V)$

**Condition**

$(u, v)$  is the tree-edge that first discovered  $v$

$d[u] < d[v] < f[v] < f[u]$

$d[v] < d[u] < f[u] < f[v]$

$d[v] < f[u] < d[u] < f[u]$

**Edge type**

Tree part of the DFS forest

Forward non-tree edge to a descendant edge to an ancestor (cycle)

Back edge between two different subtrees

**Residual Network ( $G_f$ ).**

basically, for each two edges, draw edges representing the max changes in each direction.

The network of residual capacities  $c_f(u, v) = c(u, v) - f(u, v)$ , indicating how much more flow can be pushed. Edges are created for any pair of vertices  $(u, v)$  where  $c_f(u, v) > 0$ . This includes reverse edges for backward flow.

**Augmenting Path.**

A simple path from  $s$  to  $t$  in the residual network  $G_f$ . The path's capacity is the minimum residual capacity of its edges.

**Ford-Fulkerson Algorithm.**

To find the maximum flow in a network. It does this by repeatedly finding an augmenting path in the residual network and increasing the flow along that path until no augmenting paths exist.

**Ford-Fulkerson-Method( $G, s, t$ ):**

1. Initialize flow  $f$  to 0 for all edges

2. **while** there exist an augmenting path  $p$  from  $s$  to  $t$  in the residual network  $G_f$

3. Augment flow  $f$  along path  $p$

4. **return**  $f$

**Max-flow min-cut theorem.**

Let  $G = (V, E)$  be a flow network with source  $s$  and sink  $t$  and capacities  $c$  and a flow  $f$ .

The following are equivalent:

1.  $f$  is a maximum flow

2.  $f$  has no augmenting path

3.  $|f| = c(S, T)$  for a minimum cut  $(S, T)$  (max flow = min cut)