

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).

March 25th, 2025

Master Theorem If $T(n) = aT(\frac{n}{b}) + f(n)$, $a \geq 1$, $b > 1$, and $f(n)$ asymptotically positive Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. Case 2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$. Case 3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $a f(\frac{n}{b}) \leq c f(n)$ for some $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. Common case - If $f(n) = \Theta(n^d)$ for some exponent d: - If $\frac{d}{b} < 1$ (or $d < \log_b a$), then $T(n) = \Theta(n^d)$. - If $\frac{d}{b} = 1$ (or $d = \log_b a$), then $T(n) = \Theta(n^d \log n)$. - If $\frac{d}{b} > 1$ (or $d > \log_b a$), then $T(n) = \Theta(n^{\log_b a})$.	
Queue Operations Queue-Empty(Q): Time: $O(1)$, Auxiliary Space: $O(1)$ 1. Returns TRUE if the queue is empty ($Q.head = Q.tail$). 2. Returns FALSE otherwise. Enqueue(Q, x): Time: $O(1)$, Auxiliary Space: $O(1)$ 1. Adds element x to the rear of queue Q . Dequeue(Q): Time: $O(1)$, Auxiliary Space: $O(1)$ 1. If $Q.tail = x$ 2. $Q.tail = Q.tail + 1$ (or wrap around if using circular array) Dequeue(Q): Time: $O(1)$, Auxiliary Space: $O(1)$ 1. If Queue-Empty(Q), return error "underflow". 2. Otherwise, remove and return the element at the front. 3. $x = Q.head$ 4. $Q.head = Q.head + 1$ (or wrap around) 5. Return x Queue Implementation: 1. Q.head: Index of the front element 2. Q.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 Overall Space Complexity: $O(n)$ for a queue of capacity n	
Merge Sort 1. Divide: Split the array evenly into two smaller subarrays, and continue dividing recursively. 2. Sort (Recursively): Apply merge sort recursively on each subarray until each has only one element (base case). MERGE-SORT(A, p, q, r) $n_1 \leftarrow q - p + 1$ $n_2 \leftarrow r - q$ let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays for $i \leftarrow 1$ to n_1 do $L[i] \leftarrow A[p + i - 1]$ for $j \leftarrow 1$ to n_2 do $R[j] \leftarrow A[q + j]$ $L[n_1 + 1] \leftarrow \infty$ $R[n_2 + 1] \leftarrow \infty$ $i \leftarrow 1$ $j \leftarrow 1$ for $k \leftarrow p$ to r do if $L[i] \leq R[j]$ $A[k] \leftarrow L[i]$ $i \leftarrow i + 1$ else $A[k] \leftarrow R[j]$ $j \leftarrow j + 1$ 3. Merge: Combine the two sorted subarrays into a single sorted array. a) Initializing 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. Merge Cost Complexity: $O(n)$ per merge operation. Time Complexity: $O(n \log n)$ Space Complexity: $O(n)$	
Priority Queue Maintains a dynamic set of elements with associated priority values (keys). Maximum(S): Return element of S with highest priority (return $A[1]$, complexity $O(1)$) Insert(S, x): Insert element x into set S 1. Increment the heap size 2. Insert a new node in the last position in the heap, with key $-\infty$ 3. Increase the $-\infty$ value to key using Heap-Increase-Key Extract-Max(S): Remove and return element of S with highest priority 1. Make sure heap is not empty 2. Make a copy of the maximum element (the root) 3. Make the last node in the tree the new root 4. Re-heapify the heap, with one fewer node 5. Return the copy of the maximum element Increase-Key(S, k): Increase the value of element x 's key to the new value k 1. Make sure key $\geq A[i]$ 2. Update $A[i]$'s value to key 3. Traverse the tree upward comparing new key to the parent and swapping if necessary Time Complexity: Insert, Extract-Max, Increase-Key: $O(\log n)$ Maximum: $O(1)$ Space Complexity: $O(n)$	
Stack Operations Stack-Empty(S): Time: $O(1)$, Auxiliary Space: $O(1)$ 1. Returns TRUE if the stack is empty. 2. Returns FALSE otherwise. Push(S, x): Time: $O(1)$, Auxiliary Space: $O(1)$ 1. Adds element x to the top of stack S . 2. Increments the stack pointer. Pop(S): Time: $O(1)$, Auxiliary Space: $O(1)$ 1. If Stack-Empty(S), return error "underflow". 2. Otherwise, remove and return the top element. 3. Decrements the stack pointer. Stack Implementation: 1. Elements are stored in a simple array 2. S.top: Index of the topmost element 3. An empty stack has S.top = 0 or S.top = -1 (implementation dependent) Overall Space Complexity: $O(n)$ for a stack of size n	
Dynamic Programming Problem: Optimal solutions to overlapping subproblems Fibonacci Sequence: Top-Down (Memoization): 1. Create memo array $F[0..n]$ initialized to NIL. 2. Base cases: $F[0] = 0$, $F[1] = 1$ 3. Recursive with memo: - Return $F[n]$ if already computed - Otherwise compute $F[n] = F[n-1] + F[n-2]$ - Store result in $F[n]$ and return MEMOIZED-FIB(n) Let $r = [0..n]$ be a new array for $i = 0$ to n do $r[i] \leftarrow \infty$ return MEMOIZED-FIB-AUX(n, r) MEMOIZED-FIB-AUX(n, r) if $r[n] \geq 0$ return $r[n]$ if $n = 0$ or $n = 1$ $ans \leftarrow 1$ else $ans \leftarrow$ MEMOIZED-FIB-AUX($n-1, r$) + MEMOIZED-FIB-AUX($n-2, r$) $r[n] \leftarrow ans$ return $r[n]$ Bottom-Up (Tabulation): 1. Create array $F[0..n]$ with $r[0] = 0$ 2. For $j = 1$ to n : - Compute $r[j] = \max_{1 \leq i \leq j} (r[j] + r[j-i])$ 3. Return $r[n]$ EXTENDED-BOTTOM-UP-CUT-ROD(p, n) let $r[0..n]$ and $s[0..n]$ be new arrays $r[0] \leftarrow 0$ for $j \leftarrow 1$ to n do $q \leftarrow -\infty$ for $i \leftarrow 1$ to j do if $q < p[i] + r[j-i]$ $q = p[i] + r[j-i]$ $s[j] \leftarrow i$ $r[j] \leftarrow q$ return r, s Time Complexity: $O(n^2)$ Space Complexity: $O(n)$ (includes input and memo array)	

Akra-Bazzi, Ali Najib Variation: For recurrence $T(n) = \alpha T(an) + \beta T(bn) + \Theta(n^d)$, where $\alpha, \beta > 0, a, b \in (0, 1), d \geq 0$, and unique p with $\alpha a^p + \beta b^p = 1$. Then: $p > d \Rightarrow T(n) = \Theta(n^p)$ $p = d \Rightarrow T(n) = \Theta(n^d \log n)$ $p < d \Rightarrow T(n) = \Theta(n^d)$ If p is not easily found, compare $\alpha a^d + \beta b^d$ with 1: $\alpha a^d + \beta b^d < 1 \Rightarrow T(n) = \Theta(n^d)$, $\alpha a^d + \beta b^d = 1 \Rightarrow T(n) = \Theta(n^d \log n)$, $\alpha a^d + \beta b^d > 1 \Rightarrow T(n) = \Theta(n^p)$, with p determined by $\alpha a^p + \beta b^p = 1$.	
Insertion Sort. 1 - Select the key Begin with the second element (at index 1) as the key. 2 - Compare and Shift Compare the key with elements in the sorted section (to its left). 3 - Shift Elements If an element is greater than the key, shift that element one position to the right. 4 - Insert the Key Once an element less than or equal to the key is found (or you reach the start), insert the key immediately after that element. 5 - Repeat Move forward to the next element, treating it as the new key, and repeat until the array is sorted. Time Complexity: Worst-case $O(n^2)$, Best-case $O(n)$. Space Complexity: $O(1)$.	
Strassen's Matrix Multiplication Divide: Partition each of A, B, C into four $\frac{n}{2} \times \frac{n}{2}$ submatrices: $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ Conquer: Compute 7 products (recursively on $\frac{n}{2} \times \frac{n}{2}$ matrices): $M_1 := (A_{11} + A_{22})(B_{11} + B_{22})$ $M_2 := (A_{21} + A_{22})B_{11}$ $M_3 := A_{11}(B_{12} - B_{22})$ $M_4 := A_{22}(B_{21} - B_{11})$ $M_5 := (A_{11} + A_{12})B_{22}$ $M_6 := (A_{21} - A_{11})(B_{11} + B_{12})$ $M_7 := (A_{12} - A_{22})(B_{21} + B_{22})$ Combine: Assemble the resulting submatrices to form C : $C_{11} = M_1 + M_4 - M_5 + M_7$ $C_{12} = M_2 + M_3$ $C_{21} = M_4 + M_5 - M_2 + M_6$ $C_{22} = M_1 + M_5 - M_2 + M_6$ Time Complexity: $O(n^{\log_2 7}) \approx O(n^{2.81})$ Space Complexity: $O(n^2)$	
Binary Search Trees (BST) BST-Search 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 TREE-SEARCH(x, k) if $x = \text{NIL}$ or $k = \text{key}[x]$ return x else if $k < x.\text{key}$ return TREE-SEARCH($x.\text{left}, k$) else return TREE-SEARCH($x.\text{right}, k$) Time: $O(\log n)$ avg, $O(h)$ worst Space: $O(n)$ for tree, $O(h)$ auxiliary BST-Minimum 1. Start at root 2. If NULL, return NULL 3. Follow left pointers until no left child 4. Return leftmost node TREE-MINIMUM(x) while $x.\text{left} \neq \text{NIL}$ do $x \leftarrow x.\text{left}$ return x Time: $O(h)$ Space: $O(n)$ for tree, $O(1)$ auxiliary BST-Maximum 1. Start at root 2. If NULL, return NULL 3. Follow right pointers until no right child 4. Return rightmost node TREE-MAXIMUM(x) while $x.\text{right} \neq \text{NIL}$ do $x \leftarrow x.\text{right}$ return x Time: $O(h)$ Space: $O(n)$ for tree, $O(1)$ auxiliary BST-Successor 1. If right subtree exists: Return minimum in right subtree 2. Otherwise: Find first ancestor where node is in left subtree TREE-SUCCESSOR(x) if $x.\text{right} \neq \text{NIL}$ return TREE-MINIMUM($x.\text{right}$) $y \leftarrow x.p$ while $y \neq \text{NIL}$ and $x = y.\text{right}$ do $x \leftarrow y$ $y \leftarrow y.p$ return y Time: $O(h)$ Space: $O(n)$ for tree, $O(1)$ auxiliary BST-Insert 1. Create new node z with key 2. Start at root, track parent $y = \text{NIL}$ 3. Move down tree (left if key y ; NIL key, right otherwise) 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 TREE-INSERT(T, z) $y \leftarrow \text{NIL}$ $x \leftarrow T.\text{root}$ while $x \neq \text{NIL}$ do $y \leftarrow x$ if $z.\text{key} < x.\text{key}$ $x \leftarrow x.\text{left}$ else $x \leftarrow x.\text{right}$ $z.p \leftarrow y$ if $y = \text{NIL}$ $T.\text{root} \leftarrow z$ // tree T was empty else if $z.\text{key} < y.\text{key}$ $y.\text{left} \leftarrow z$ else $y.\text{right} \leftarrow z$ Time: $O(h)$ Space: $O(n)$ for tree, $O(1)$ auxiliary	BST-Postorder 1. Recursively traverse left 2. Recursively traverse right 3. Visit current node (Children first, then root) POSTORDER-TREE-WALK(x) if $x \neq \text{NIL}$ POSTORDER-TREE-WALK($x.\text{left}$) POSTORDER-TREE-WALK($x.\text{right}$) print $x.\text{key}$ Time: $O(n)$ Space: $O(n)$ for tree, $O(h)$ auxiliary BST-Inorder 1. Recursively traverse left 2. Visit current node 3. Recursively traverse right (Visits nodes in sorted order) INORDER-TREE-WALK(x) if $x \neq \text{NIL}$ INORDER-TREE-WALK($x.\text{left}$) print $x.\text{key}$ INORDER-TREE-WALK($x.\text{right}$) Time: $O(n)$ Space: $O(n)$ for tree, $O(h)$ auxiliary BST-Preorder 1. Visit current node 2. Recursively traverse left 3. Recursively traverse right (Root first, then children) PREORDER-TREE-WALK(x) if $x \neq \text{NIL}$ print $x.\text{key}$ PREORDER-TREE-WALK($x.\text{left}$) PREORDER-TREE-WALK($x.\text{right}$) Time: $O(n)$ Space: $O(n)$ for tree, $O(h)$ auxiliary BST-Transplant 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. Else make v right child of u 's parent 4. Set v 's parent to u 's parent TRANSPLANT(T, u, v) if $u.p = \text{NIL}$ $T.\text{root} \leftarrow v$ else if $u = u.p.\text{left}$ $u.p.\text{left} \leftarrow v$ else $u.p.\text{right} \leftarrow v$ if $v.p \neq \text{NIL}$ $v.p.p \leftarrow u.p$ Time: $O(1)$ Space: $O(n)$ for tree, $O(1)$ auxiliary BST-Delete 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 TREE-DELETE(T, z) if $z.\text{left} = \text{NIL}$ TRANSPLANT($T, z, z.\text{right}$) // z has no left child else if $z.\text{right} = \text{NIL}$ TRANSPLANT($T, z, z.\text{left}$) // z has just a left child else // z has two children $y \leftarrow \text{TREE-MINIMUM}(T.\text{right})$ // y is z 's successor if $y.p \neq \text{NIL}$ TRANSPLANT($T, y, y.\text{right}$) $y.\text{right} \leftarrow z.\text{right}$ $y.\text{right}.p \leftarrow y$ TRANSPLANT(T, z, y) $y.\text{left} \leftarrow z.\text{left}$ $y.\text{left}.p \leftarrow y$ Time: $O(h)$ Space: $O(n)$ for tree, $O(1)$ auxiliary
Properties: - 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	

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 $e[1 \dots n+1, 0 \dots n]$ 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: $e[i, j] = \min_{i \leq r \leq j} \{ e[i, r-1] + e[r+1, j] + w[i, j] \}$
- The root of the overall optimal tree is in $root[1, n]$

```
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] ← 0
  w[i, i-1] ← 0
for l = 1 to n do
  for i = 1 to n-l+1 do
    j ← i+l-1
    e[i, j] ← ∞
    w[i, j] ← w[i, j-1] + p_j
    for r = i to j do
      t ← e[i, r-1] + e[r+1, j] + w[i, j]
      if t < e[i, j]
        e[i, j] ← t
        root[i, j] ← 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 $m[i, i] = 0$
- $m[i, j]$ stores minimal cost of multiplying matrices i through j
2. For $l = 2$ to n (chain length):
3. For $i = 1$ to $n-l+1$:
- Set $j = i+l-1$
- Compute $m[i, j] = \min_{i \leq k < j} \{ m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \}$
- 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 l = 2 to n do // l is the chain length
    for i = 1 to n-l+1 do
      j ← i+l-1
      m[i, j] ← ∞
      for k = i 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] ← q
          s[i, j] ← 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 1. 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. being deleted.
4. Update the head pointer to point to 4. For doubly linked lists, also update the prev pointer of the next node.
5. If the list was empty, update the tail 5. Free the memory allocated for the node as well.

Complexity:

- Time: $O(n^2)$
- Space: $O(1)$

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
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 $S = S_x - S_y \cup S_y \cup S_x \cup S_y$

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 ← x
x.rank ← 0
```

FIND-SET(x)

```
if x ≠ x.p
  x.p ← 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 ← x
else
  x.p ← y
if x.rank = y.rank
  y.rank ← y.rank + 1
```

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] and c[0...m, 0...n] be new tables
for i = 1 to m do
  c[i, 0] ← 0
  for j = 0 to n do
    c[0, j] ← 0
  for i = 1 to m do
    for j = 1 to n do
      if x_i = y_j
        c[i, j] ← c[i-1, j-1] + 1
      b[i, j] ← ↖
      else if c[i-1, j] ≥ c[i, j-1]
        c[i, j] ← c[i-1, j]
        b[i, j] ← ↑
      else
        c[i, j] ← c[i, j-1]
        b[i, j] ← ←
    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:

- Recursively trace back through direction table b
- Follow diagonal arrows and print characters
- Skip cells with up or left arrows
- Stops when reaching first row or column

```
PRINT-LCS(b, X, i, j)
if i = 0 ∨ j = 0 return
if b[i, j] == ↖
  PRINT-LCS(b, X, i-1, j-1)
  print x_i
else if b[i, j] == ↑
  PRINT-LCS(b, X, i, j-1)
else
  PRINT-LCS(b, X, i, j-1)
```

Example Recording of Solution:

	1	2	3	4	5	6	7
i	0	0	0	0	0	0	0
1	0	0	↗	0	↑	↖	↖
2	0	0	↑	↖	↖	↑	↑
3	0	0	↑	↑	↑	↖	↖
4	0	↑	↑	↑	↑	↑	↑
5	0	↑	↑	↑	↑	↑	↖
6	0	↑	↑	↑	↑	↑	↖

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:

- 1 - Start at vertex s and mark it as distance 0.
- 2 - Visit all immediate neighbors first, marking them as distance 1.
- 3 - Then visit their unvisited neighbors, marking as distance 2.
- 4 - Continue this layer-by-layer exploration, incrementing distance at each level.

Key Data: Distance of each node from source is stored and gives shortest path in unweighted graphs.

Time Complexity: $O(V+E)$

Space Complexity: $O(V)$

BFS(V, E, s)

```
for all u ∈ V \ {s} do
  u.d ← ∞
s.d ← 0, Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅ do
  u ← DEQUEUE(Q)
  for all v ∈ G.Adj[u] do
    if u.d + 1 < v.d then
      v.d ← u.d + 1 and ENQUEUE(Q, v)
```

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 : $\Theta(\text{degree}(u))$

Time Complexity: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$

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 a DFS of G 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 $v \in G.V$
2. Output vertices in order of decreasing finishing times

Time Complexity: $O(V+E)$

Space Complexity: $O(V)$

Condition	Edge type	Notes
(u, v) is the tree-edge that first discovered v	Tree	part of the DFS forest
$d[u] < d[v] < f[v] < f[u]$	Forward	non-tree edge to a descendant
$d[v] < d[u] < f[u] < f[v]$	Back	edge to an ancestor (cycle)
$d[v] < f[v] < d[u] < f[u]$	Cross	between two different subtrees

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: $c(S, T) = \sum_{s \in S, v \in T} c(u, v)$ (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, then

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

Depth-First Search (DFS)

Goal:

Explore a graph by going as deep as possible along each path before backtracking (like exploring a maze).

How it works:

- 1 - Initialize all vertices as WHITE (unvisited).
- 2 - For each WHITE vertex, start DFS-Visit:
- 3 - Mark current vertex as GRAY (being processed), record discovery time.
- 4 - Recursively visit each WHITE neighbor (ignore GRAY/BLACK).
- 5 - 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 Space Comp. $O(E)$

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

Time Comp. $O(V+E)$

Space Comp.: $O(V)$

Tree edge. Edge from parent to child in DFS tree.

Back edge. Edge from descendant to ancestor (indicates cycle).

Forward edge. Edge from ancestor to descendant (not in DFS tree).

Cross edge. Edge between vertices with no ancestor-descendant relationship.

Theorem - White Path. v is descendant of u iff when u is discovered, there exists a white path (only undiscovered vertices) from u to v .

Theorem - Parenthesis. For vertices u, v : intervals $[u.d, u.f]$ and $[v.d, v.f]$ are either disjoint or one contains the other.

G transpore. The graph G with all edges reversed.

Algorithm - Strongly Connected Component.

A maximal set of vertices in a directed graph where for every pair u, v there is a path from u to v and v to u .

SCC(G):

1. Call DFS(G) to compute finishing times $u.f$ for all u .
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: $O(V)$

Algorithm - Strongly Connected Component.

A maximal set of vertices in a directed graph where for every pair u, v there is a path from u to v and v to u .

SCC(G):

1. Call DFS(G) to compute finishing times $u.f$ for all u .
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	Edge type	Notes
(u, v) is the tree-edge that first discovered v	Tree	part of the DFS forest
$d[u] < d[v] < f[v] < f[u]$	Forward	non-tree edge to a descendant
$d[v] < d[u] < f[u] < f[v]$	Back	edge to an ancestor (cycle)
$d[v] < f[v] < d[u] < f[u]$	Cross	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.

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.

Goal: 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 more augmenting paths exist.

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

1. Initialize flow f to 0 for all edges
2. while there exists 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. G_f has no augmenting path
3. $|f| = c(S, T)$ for a minimum cut (S, T) (max flow = min cut)

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. G_f has no augmenting path
3. $|f| = c(S, T)$ for a minimum cut (S, T) (max flow = min cut)