Facts

$\sum_{k=0}^{n} ar^k = \frac{ar^{n+1} - a}{r - 1}$ for $r \notin \{0, 1\}$

Loop Invariant

A property that holds before and after each loop iteration. **Initialization**: Holds before the first iteration. **Maintenance**: If it holds before an iteration, it holds after. **Termination**: When the loop ends, the invariant helps prove correctness.

Divide and Conquer

An algorithmic paradigm with three steps:
1. **Divide**: Split the problem into smaller subproblems.
2. **Conquer**: Solve each subproblem recursively.
3. **Combine**: Merge the subproblem solutions into the final result. The recurrence relation is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ a\,T(n/b) + D(n) + C(n) & \text{otherwise,} \end{cases}$$

a: number of subproblems,
n/b: size of each subproblem,
$D(n)$: time to divide,
$C(n)$: time to combine.

Solving Recurrences

To solve a recurrence using the substitution method:
1. **Guess** the solution's form (e.g., $\Theta(n^2)$).
2. **Prove the upper bound** by induction using a constant and the guessed form.
3. **Prove the lower bound** similarly.
4. **Conclude** that the guess is correct. **Example:**

$T(n) = T(n-1) + cn \Rightarrow \Theta(n^2)$

Stack

A stack is a last-in/fist-out (LIFO) data-structure
Supported operations:
- **Push:** Insert an element at head.
- **Pop:** Retrieve head.

Master Theorem

Let $a \ge 1$, $b > 1$, and let $T(n)$ be defined by the recurrence:

$$T(n) = a\,T(n/b) + f(n)$$

Then $T(n)$ has the following asymptotic bounds:
1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \ge 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if $a\,f(n/b) \le c\,f(n)$ for some $c < 1$ and large n, then $T(n) = \Theta(f(n))$.

Insertion Sort

Steps:
1. Start with an empty (or trivially sorted) sublist.
2. Insert the next element in the correct position by comparing backwards.
3. Repeat for all elements.

Complexity:
Space: $\Theta(n)$, Time: $\Theta(n^2)$
Pseudocode:
Require: $A = \langle a_1, a_2, \ldots, a_n \rangle$

```
for i ← 2 to n do
    key ← A[i]
    j ← i − 1
    while j ≥ 1 and A[j] > key do
        A[j + 1] ← A[j]
        j ← j − 1
    end while
    A[j + 1] ← key
end for
```

Heap Sort

A heap is a **nearly complete binary tree** where each node satisfies the **max-heap property**: For every node i, its children have smaller or equal values.
The **height** of a heap is the length of the longest path from the root to a leaf.
Useful Index Rules (array-based heap):
- Root is at index $A[1]$
- Left child of node i: index $2i$
- Right child of node i: index $2i + 1$
- Parent of node i: index $\lfloor i/2 \rfloor$

Complexity:
Space: $\Theta(n)$, Time: $\Theta(n \log n)$
Pseudocode:

```
procedure Max-Heapify(A, i, n)
    l ← Left(i)
    r ← Right(i)
    largest ← i
    if l ≤ n and A[l] > A[largest] then
        largest ← l
    end if
    if r ≤ n and A[r] > A[largest] then
        largest ← r
    end if
    if largest ≠ i then
        Exchange A[i] ↔ A[largest]
        Max-Heapify(A, largest, n)
    end if
end procedure

procedure Build-Max-Heap(A[1, . . . , n])
    for i ← ⌊n/2⌋ downto 1 do
        Max-Heapify(A, i, n)
    end for
end procedure

procedure Heapsort(A[1, . . . , n])
    Build-Max-Heap(A)
    for i ← n downto 2 do
        exchange A[1] with A[i]
        Max-Heapify(A, 1, i − 1)
    end for
end procedure
```

Injective Functions

Let $f : \{1, 2, \ldots, q\} \to M$ be a function chosen uniformly at random, where $|M| = m$. If $q > 1.78\sqrt{m}$, then the probability that f is injective is at most $\frac{1}{2}$.

Merge Sort

Uses the **divide and conquer** paradigm. **Complexity:**
Space: $\Theta(n)$, Time: $\Theta(n \log n)$
Pseudocode:

```
procedure sort(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        sort(A, p, q)
        sort(A, q + 1, r)
        merge(A, p, q, r)
    end if
end procedure

procedure merge(A, p, q, r)
    n1 ← q − p + 1, n2 ← r − q
    Let L[1 . . . n1 + 1], R[1 . . . n2 + 1] be new arrays
    for i ← 1 to n1 do L[i] ← A[p + i − 1]
    end for
    for j ← 1 to n2 do R[j] ← A[q + j]
    end for
    L[n1 + 1], R[n2 + 1] ← ∞
    i, j ← 1
    for k ← p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i]; i ← i + 1
        else
            A[k] ← R[j]; j ← j + 1
        end if
    end for
end procedure
```

Maximum Subarray Problem (Kadane's Algorithm)

Idea: Iterate from left to right, maintaining: - `endingHereMax`: best subarray ending at current index - `currentMax`: best seen so far
Observation: At index $j + 1$, the maximum subarray is either:
- the best subarray in $A[1..j]$, or
- a subarray ending at $j + 1$, i.e., $A[i..j + 1]$

Formula:
$\text{maxSub}(A[1..n]) = \max\left(\text{maxSub}(A[1..n-1]),\ \max_{i=1}^{n} \text{sum}(A[i..n])\right)$
Complexity: Time $\Theta(n)$,
Space $\Theta(1)$
Pseudocode:

```
procedure Linear-Max-Subarray(A[1..n])
    current_max ← −∞
    ending_here_max ← −∞
    for i ← 1 to n do
        ending_here_max ← max(A[i], ending_here_max + A[i])
        current_max ← max(current_max, ending_here_max)
    end for
    return current_max
end procedure
```

Queue

A queue is a first-in, first-out (FIFO) collection.
- **enqueue:** Insert an element at tail.
- **dequeue:** Retrieve head.

```
procedure Enqueue(Q, x)
    Q[Q.tail] ← x
    if Q.tail = Q.length then
        Q.tail ← 1
    else
        Q.tail ← Q.tail + 1
    end if
end procedure

procedure Dequeue(Q)
    x ← Q[Q.head]
    if Q.head = Q.length then
        Q.head ← 1
    else
        Q.head ← Q.head + 1
    end if
    return x
end procedure
```

Dynamic Programming

Two key approaches: **Top-down** and **Bottom-up**.
- **Top-down:** Starts from the problem n and solves subproblems recursively, storing results (memoization).
- **Bottom-up:** Starts from base cases (e.g., 0) and iteratively builds up to the final solution.

The core idea is to **remember previous computations** to avoid redundant work and save time.

Hash Functions and Tables

Tables are a special kind of collection that associate keys to values, allowing the following operations:
1. Insert a new key-value pair.
2. Delete a key-value pair.
3. Search for the value associated with a given key.

Direct-Address Tables. We define a function $f : K \to \{1, \ldots, |K|\}$ and create an array of size $|K|$ where each position corresponds directly to a key, allowing constant-time access.
Hash Tables. Hash tables use space proportional to the number of stored keys $|K'|$, i.e., $\Theta(|K'|)$, and support the above operations in expected time $\mathcal{O}(1)$ in the average case. To achieve this, we define a hash function $h : K \to \{1, \ldots, M\}$ and use an array of size M where each entry contains a linked list of key-value pairs (k, v).

Strassen's Algorithm for Matrix Multiplication

Instead of performing 8 recursive multiplications as in the naive divide-and-conquer matrix multiplication, Strassen's algorithm reduces it to 7, which improves the time complexity.
Definitions:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Resulting matrix:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Complexity:
Time: $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$,
Space: $\Theta(n^2)$

Priority Queue

A priority queue maintains a dynamic set S of elements, each with an associated key that defines its priority. At each operation, we can access the element with the highest key. **Supported operations:**
- **Insertion:** Insert an element x into S.
- **Maximum:** Return the element in S with the largest key.
- **Extract-Max:** Remove and return the element with the largest key.
- **Increase-Key:** Increase the key of an element x to a new value k (assuming $k \ge$ current key).

Pseudocode:

```
procedure Heap-Maximum(S)
    return S[1]
end procedure
procedure Heap-Extract-Max(S, n)
    if n < 1 then
        error "heap underflow"
    end if
    max ← S[1]
    S[1] ← S[n]
    n ← n − 1
    Max-Heapify(S, 1, n)
    return max
end procedure
procedure Heap-Increase-Key(S, i, key)
    if key < S[i] then
        error "new key is smaller than current key"
    end if
    S[i] ← key
    while i > 1 and S[Parent(i)] < S[i] do
        exchange S[i] with S[Parent(i)]
        i ← Parent(i)
    end while
end procedure
procedure Max-Heap-Insert(S, key, n)
    n ← n + 1
    S[n] ← −∞
    Heap-Increase-Key(S, n, key)
end procedure
```

Heap Sort

Complexity:
Space: $\Theta(n)$, Time: $\Theta(n \log n)$

```
procedure Max-Heapify(A, i, n)
    l ← Left(i)
    r ← Right(i)
    if l ≤ n and A[l] > A[i] then
        largest ← l
    else
        largest ← i
    end if
    if r ≤ n and A[r] > A[largest] then
        largest ← r
    end if
    if largest ≠ i then
        exchange A[i] with A[largest]
        Max-Heapify(A, largest, n)
    end if
end procedure
procedure Build-Max-Heap(A, n)
    for i ← ⌊n/2⌋ downto 1 do
        Max-Heapify(A, i, n)
    end for
end procedure
procedure LargestK(A, B, k)
    Create an empty heap H
    B[k] ← A[1]
    Insert A[2] and A[3] into H
    for i ← k − 1, k − 2, . . . , 1 do
        tmp ← Extract-Max(H)
        B[i] ← tmp
        Insert tmp's children in A into H
    end for
end procedure
```

Linked List

A linked list is a linear data structure where each element (node) points to the next. Unlike arrays, it is not index-based and allows efficient insertions and deletions.
Operations:
- **Search:** Find an element with a specific key — $\Theta(n)$
- **Insert:** Insert an element at the head — $\Theta(1)$
- **Delete:** Remove an element — $\Theta(1)$

Pseudocode:

```
                        ▷ Searches for the first element with key k
procedure List-Search(L, k)
    x ← L.head
    while x ≠ NIL and x.key ≠ k do
        x ← x.next
    end while
    return x
end procedure
                        ▷ Inserts a new node x at the head of the list
procedure List-Insert(L, x)
    x.next ← L.head
    if L.head ≠ NIL then
        L.head.prev ← x
    end if
    L.head ← x
    x.prev ← NIL
end procedure
                        ▷ Deletes node x from the list
procedure List-Delete(L, x)
    if x.prev ≠ NIL then
        x.prev.next ← x.next
    else
        L.head ← x.next
    end if
    if x.next ≠ NIL then
        x.next.prev ← x.prev
    end if
end procedure
```

Binary Search Trees

A binary search tree (BST) is a binary tree where each node has a key and satisfies the following properties:
- For any node x, all keys in its left subtree are less than x.key.
- All keys in its right subtree are greater than or equal to x.key.

Pseudocode:

```
    ▷ Searches for a node with key k starting from node x
    ▷ Runs in O(h) time, where h is the height of the tree
procedure Tree-Search(x, k)
    if x = NIL or k = x.key then
        return x
    else if k < x.key then
        return Tree-Search(x.left, k)
    else
        return Tree-Search(x.right, k)
    end if
end procedure
    ▷ Finds the minimum key node in the subtree rooted at x
procedure Tree-Minimum(x)
    while x.left ≠ NIL do
        x ← x.left
    end while
    return x
end procedure
    ▷ Finds the maximum key node in the subtree rooted at x
procedure Tree-Maximum(x)
    while x.right ≠ NIL do
        x ← x.right
    end while
    return x
end procedure
```

Modify a Binary Tree

```
procedure Tree-Insert(T, z)
    y ← NIL                         ▷ Parent of x
    x ← T.root
    while x ≠ NIL do
        y ← x
        if z.key < x.key then
            x ← x.left
        else
            x ← x.right
        end if
    end while
    z.p ← y
    if y = NIL then
        T.root ← z
    else if z.key < y.key then
        y.left ← z
    else
        y.right ← z
    end if
end procedure
procedure Transplant(T, u, v)
    if u.p = NIL then
        T.root ← v
    else if u = u.p.left then
        u.p.left ← v
    else
        u.p.right ← v
    end if
    if v ≠ NIL then
        v.p ← u.p
    end if
end procedure
procedure Tree-Delete(T, z)
    if z.left = NIL then
        Transplant(T, z, z.right)
    else if z.right = NIL then
        Transplant(T, z, z.left)
    else
        y ← Tree-Minimum(z.right)
        if y.p ≠ z then
            Transplant(T, y, y.right)
            y.right ← z.right
            y.right.p ← y
        end if
        Transplant(T, z, y)
        y.left ← z.left
        y.left.p ← y
    end if
end procedure
```

Building a Binary Search Tree

Given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct sorted keys and, for every k_i, a probability p_i, find a binary search tree T that minimizes:
$E[\text{search cost in } T] = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$
This is solved via dynamic programming.
Time complexity: $\mathcal{O}(n^3)$

```
procedure 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] ← 0
        w[i][i − 1] ← 0
    end for
    for l ← 1 to n do               ▷ length of subproblem
        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] then
                    e[i][j] ← t
                    root[i][j] ← r
                end if
            end for
        end for
    end for
end procedure
```

Rod Cutting

Given a rod of length n and a table of prices p_i for rods of length $i = 1, \ldots, n$, determine the optimal way to cut the rod to maximize profit.
The optimal revenue function $r(n)$ is defined as:

$$r(n) = \begin{cases} 0 & \text{if } n = 0, \\ \max_{1 \le i \le n} \{p_i + r(n-i)\} & \text{if } n \ge 1. \end{cases}$$

```
procedure Extended-Bottom-Up-Cut-Rod(p, n)
    let r[0 . . . n] and s[0 . . . n] be new arrays
    r[0] ← 0
    s[0] ← 0        ▷ Usually s[0] isn't explicitly used for solution reconstruction, but included as per your pseudocode
    for j ← 1 to n do
        q ← −∞
        for i ← 1 to j do
            if q < p[i] + r[j − i] then
                q ← p[i] + r[j − i]
                s[j] ← i
            end if
        end for
        r[j] ← q
    end for
    return r and s
end procedure
```

Counting Sort

Counting Sort assumes the input consists of n integers in the range 0 to k and sorts them in $\mathcal{O}(n + k)$ time. It is stable and non-comparative.

```
procedure Counting-Sort(A, B, n, k)
    let C[0 . . . k] be a new array
    for i ← 0 to k do
        C[i] ← 0
    end for
    for j ← 1 to n do
        C[A[j]] ← C[A[j]] + 1
    end for
    for i ← 1 to k do
        C[i] ← C[i] + C[i − 1]
    end for
    for j ← n downto 1 do
        B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
    end for
end procedure
```

Matrix-Chain Multiplication

Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimensions $p_{i-1} \times p_i$, find the most efficient way to fully parenthesize the product $A_1 A_2 \cdots A_n$ so as to minimize the total number of scalar multiplications.
The optimal substructure is defined by the recurrence:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

```
procedure 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
    end for
    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] then
                    m[i][j] ← q
                    s[i][j] ← k      ▷ s stores the optimal split point
                end if
            end for
        end for
    end for
end procedure
```

Time complexity: $\mathcal{O}(n^3)$
Space complexity: $\mathcal{O}(n^2)$

Longest Common Subsequence

Given as input two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, we want to find the longest common subsequence (not necessarily contiguous, but in order).

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } x_i = y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{otherwise.} \end{cases}$$

```
procedure 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
    end for
    for j ← 0 to n do
        c[0][j] ← 0
    end for
    for i ← 1 to m do
        for j ← 1 to n do
            if X[i] = Y[j] then
                c[i][j] ← c[i−1][j−1] + 1
                b[i][j] ← "↖"                ▷ North-west arrow
            else
                if c[i−1][j] ≥ c[i][j−1] then
                    c[i][j] ← c[i−1][j]
                    b[i][j] ← "↑"            ▷ Up arrow
                else
                    c[i][j] ← c[i][j−1]
                    b[i][j] ← "←"            ▷ Left arrow
                end if
            end if
        end for
    end for
    return c and b
end procedure
```

Time complexity: $\mathcal{O}(mn)$ **Space complexity:** $\mathcal{O}(mn)$

Graph

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

Feature	Adjacency List	Adjacency Matrix
Space	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert^2)$
Time to list all vertices adjacent to u	$\Theta(\deg(u))$	$\Theta(\lvert V\rvert)$
Time to determine whether $(u,v) \in E$	$O(\deg(u))$	$\Theta(1)$

Connectivity: A graph is said to be connected if every pair of vertices in the graph is connected.
Connected Component: A connected component is a maximal connected subgraph of an undirected graph.
Complete Graph: A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.
Vertex Cut: A vertex cut or separating set of a connected graph G is a set of vertices whose removal renders G disconnected.

Breadth-First Search

Given as input a graph $G = (V, E)$, either directed or undirected, and a source vertex $s \in V$, we want to find $v.d$, the smallest number of edges (distance) from s to v, for all $v \in V$.

1. Send a wave out from s,
2. first hit all vertices at 1 edge from s,
3. then, from there, hit all vertices at 2 edges from s, and so on.

```
procedure BFS(V, E, s)
    for each u ∈ V \ {s} do
        u.d ← ∞
    end for
    s.d ← 0
    let Q be a new queue
    Enqueue(Q, s)
    while Q ≠ ∅ do
        u ← Dequeue(Q)
        for each v ∈ G.Adj[u] do
            if v.d = ∞ then
                v.d ← u.d + 1
                Enqueue(Q, v)
            end if
        end for
    end while
end procedure
```

Time complexity: $\mathcal{O}(|V| + |E|)$

Depth-First Search

Given, as input, a graph $G = (V, E)$, either directed or undirected, we want to output two timestamps on each vertex:

- $v.d$ — discovery time (when v is first encountered),
- $v.f$ — finishing time (when all vertices reachable from v have been fully explored).

Each vertex has a color state:

- **WHITE**: undiscovered,
- **GRAY**: discovered but not finished,
- **BLACK**: fully explored.

```
procedure DFS(G)
    for each u ∈ G.V do
        u.color ← WHITE
    end for
    time ← 0
    for each u ∈ G.V do
        if u.color = WHITE then
            DFS-Visit(G, u)
        end if
    end for
end procedure
procedure DFS-Visit(G, u)
    time ← time + 1
    u.d ← time                       ▷ Discovery time
    u.color ← GRAY
    for each v ∈ G.Adj[u] do
        if v.color = WHITE then
            DFS-Visit(G, v)
        end if
    end for
    u.color ← BLACK                  ▷ Finishing time
    time ← time + 1
    u.f ← time
end procedure
```

Time complexity: $\mathcal{O}(|V| + |E|)$

Topological Sort

Given a directed acyclic graph (DAG) $G = (V, E)$, the goal is to produce a linear ordering of its vertices such that for every edge $(u, v) \in E$, vertex u appears before v in the ordering.

Key Properties:

- A graph is a DAG if and only if a DFS yields no back edges.
- The topological sort is obtained by performing DFS and ordering vertices in decreasing order of their finishing times.

Algorithm:

1. Run DFS on G to compute finishing times $v.f$ for all $v \in V$.
2. Return the vertices sorted in descending order of $v.f$.

Running Time: $\Theta(|V| + |E|)$, same as DFS.

Strongly Connected Components

A **strongly connected component (SCC)** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair $u, v \in C$, there is a path from u to v and from v to u.

Transpose of a Graph: The transpose of G, denoted $G^T = (V, E^T)$, has all edges reversed:

$$E^T = \{(u, v) \mid (v, u) \in E\}$$

G and G^T share the same SCCs. Computing G^T takes $\Theta(|V| + |E|)$ time with adjacency lists.

Algorithm (Kosaraju's):

1. Run **DFS** on G to compute finishing times $u.f$ for all $u \in V$.
2. Compute the transpose G^T.
3. Run **DFS** on G^T, but visit vertices in order of decreasing $u.f$ (from step 1).
4. Each tree in the resulting DFS forest is one SCC.

Time Complexity: $\Theta(|V| + |E|)$

Flow Network

We model the movement of flow through a network of edges, where each edge has a **capacity**—the maximum flow allowed. Our goal is to maximize the total flow from a **source** vertex s to a **sink** vertex t.
Flow Function: A flow is a function $f : V \times V \to$ that satisfies:

1. **Capacity Constraint:** For all $u, v \in V$,
$$0 \leq f(u, v) \leq c(u, v)$$
where $c(u, v)$ is the capacity of edge (u, v).
2. **Flow Conservation:** For all $u \in V \setminus \{s, t\}$,
$$_{v \in V} f(v, u) = {}_{v \in V} f(u, v)$$
i.e., the total flow into u equals the total flow out of u (except for source and sink).

Value of the Flow:

$$|f| = {}_{v \in V} f(s, v) - {}_{v \in V} f(v, s)$$

This represents the total net flow out of the source s.

Ford-Fulkerson Method (1954)

The Ford-Fulkerson method finds the maximum flow from a source s to a sink t in a flow network $G = (V, E)$.
Algorithm:

1. Initialize flow $f(u, v) = 0$ for all $(u, v) \in E$.
2. While there exists an **augmenting path** p from s to t in the residual network G_f:
   - Compute the **bottleneck capacity** $c_f(p)$ (the minimum residual capacity along p).
   - Augment flow f along p by $c_f(p)$.

Residual Network: Given flow f, define residual capacity c_f as:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Then the residual graph is $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.

Cuts and Optimality:

- A **cut** (S, T) of the network is a partition of V with $s \in S$, $t \in T$.
- The **flow across the cut** is:
$$f(S, T) = {}_{u \in S, v \in T} f(u, v) - {}_{u \in T, v \in S} f(u, v)$$
- The **capacity of the cut** is:
$$c(S, T) = {}_{u \in S, v \in T} c(u, v)$$
- For any flow f and any cut (S, T), we have: $|f| \leq c(S, T)$.

Max-Flow Min-Cut Theorem: The value of the maximum flow equals the capacity of the minimum cut.
Augmenting Path Is a path from the source to the sink in the residual graph such that every edge on the path has available capacity

Bipartite Graphs

A **bipartite graph** (or bigraph) is a graph $G = (U, V, E)$ whose vertices can be partitioned into two disjoint sets U and V such that every edge connects a vertex from U to one in V.

Properties:

- U and V are called the *parts* of the graph.
- Bipartiteness can be tested using BFS:
  - Label the source s as even.
  - During BFS, label each unvisited neighbor of a vertex with the opposite parity (even $\leftrightarrow$ odd).
  - If a conflict arises (a vertex is visited twice with the same parity), the graph is not bipartite.

Bipartite Match via Max-Flow

- **1. Add a source node** s and connect it to all nodes in the left partition (say U) and do same for right partition to sink.
- **2. For each edge** (u, v) in the bipartite graph (with $u \in U$, $v \in V$), add a directed edge from u to v.
- **3. Assign a capacity of 1** to all edges.
- **4. Run Ford–Fulkerson** from s to t.

Spanning Trees

Spanning Tree: A *spanning tree* T of an undirected graph $G = (V, E)$ is a subgraph that:

- Includes all vertices of G.
- Is a tree: connected and acyclic.

Minimum Spanning Tree (MST): An MST is a spanning tree of a weighted graph with the *minimum total edge weight* among all spanning trees of the graph.
Key Properties:

- Every connected undirected graph has at least one MST.
- An MST connects all possible vertices using the lightest possible total edge weight without forming cycles.

Prim's Algorithm

Goal: Find a Minimum Spanning Tree (MST) of a connected, weighted undirected graph.
Idea:

1. Start from an arbitrary root vertex r.
2. Maintain a growing tree T, initialized with r.
3. Repeatedly add the minimum-weight edge that connects a vertex in T to a vertex outside T.

Data Structures: Uses a min-priority queue to select the next lightest edge crossing the cut.

```
procedure Prim(G, w, r)
    let Q be a new min-priority queue
    for each u ∈ G.V do
        u.key ← ∞
        u.π ← NIL
        Insert(Q, u)
    end for
    Decrease-Key(Q, r, 0)
    while Q ≠ ∅ do
        u ← Extract-Min(Q)
        for each v ∈ G.adj[u] do
            if v ∈ Q and w(u, v) < v.key then
                v.π ← u
                Decrease-Key(Q, v, w(u, v))
            end if
        end for
    end while
end procedure
```

Runtime: $\Theta((V + E) \log V)$ with binary heaps, $\Theta(E + V \log V)$ with Fibonacci heaps.

Kruskal's Algorithm

Goal: Find a Minimum Spanning Tree (MST) in a connected, weighted undirected graph.
Idea:

1. Start with an empty forest A (each vertex is its own tree).
2. Sort all edges in non-decreasing order of weight.
3. For each edge (u, v), if u and v are in different trees (i.e., no cycle is formed), add the edge to A.
4. Use a disjoint-set (Union-Find) data structure to efficiently check and merge trees.

```
procedure Kruskal(G, w)
    A ← ∅
    for each v ∈ G.V do
        Make-Set(v)
    end for
    sort the edges of G.E into non-decreasing order by weight w
    for each (u, v) ∈ sorted edge list do
        if Find-Set(u) ≠ Find-Set(v) then
            A ← A ∪ {(u, v)}
            Union(u, v)
        end if
    end for
    return A
end procedure
```

Runtime: $\Theta(|E| \log |E|)$ due to sorting, plus nearly linear time for Union-Find operations (with union by rank and path compression).

Edge Disjoint Paths using Max Flow

You can use the Max-Flow algorithm to find all edge-disjoint paths from a source to a sink by assigning a capacity of 1 to every edge and running Ford–Fulkerson. The maximum flow value will be equal to the number of edge-disjoint s–t paths.

Bellman-Ford Algorithm

Goal: Compute shortest paths from a single source s to all other vertices in a weighted graph $G = (V, E)$, allowing negative edge weights.
Key Idea:

- Relax all edges repeatedly (up to $|V| - 1$ times).
- After that, check for negative-weight cycles: if we can still relax an edge, a negative cycle exists.

Initialization:

```
procedure Init-Single-Source(G, s)
    for each v ∈ G.V do
        v.d ← ∞                     ▷ distance estimate
        v.π ← NIL                   ▷ predecessor
    end for
    s.d ← 0
end procedure
```

Relaxation:

```
procedure Relax(u, v, w)
    if v.d > u.d + w(u, v) then
        v.d ← u.d + w(u, v)
        v.π ← u
    end if
end procedure
```

Main Algorithm:

```
procedure Bellman-Ford(G, w, s)
    Init-Single-Source(G, s)
    for i ← 1 to |G.V| − 1 do
        for each edge (u, v) ∈ G.E do
            Relax(u, v, w)
        end for
    end for
    for each edge (u, v) ∈ G.E do
        if v.d > u.d + w(u, v) then
            return false            ▷ Negative-weight cycle detected
        end if
    end for
    return true
end procedure
```

Runtime: $\Theta(|V||E|)$
Handles: Negative weights (but no negative cycles)

Dijkstra's Algorithm

Goal: Compute the shortest paths from a single source s to all other vertices in a weighted graph $G = (V, E)$ with non-negative edge weights.
Key Idea:

- Greedily grow a set S of vertices with known shortest paths.
- At each step, pick the vertex $u \notin S$ with the smallest tentative distance $u.d$.
- Relax all edges (u, v) from u to update distance estimates.

Pseudocode:

```
procedure Dijkstra(G, w, s)
    Init-Single-Source(G, s)
    S ← ∅
    Q ← G.V                 ▷ insert all vertices into priority queue Q
    while Q ≠ ∅ do
        u ← Extract-Min(Q)
        S ← S ∪ {u}
        for each v ∈ Adj[u] do
            Relax(u, v, w)
        end for
    end while
end procedure
```

Runtime: $\Theta(|E| \log |V|)$

HTable

Hash tables are a data structure that use a function $h(k)$ mapping keys to indices in the range 1 to p, such that each element is stored at index $h(k)$. Collisions are managed using chaining (linked lists), leading to:

- $\mathcal{O}(1)$ insertion
- $\mathcal{O}(N + E)$ search (in worst-case)
- $\mathcal{O}(1)$ deletion

The Hiring Problem

Problem: We interview n candidates for a position, one by one in random order. After each interview, we decide immediately whether to hire the candidate. We want to compute the expected number of times we hire someone (i.e., when they are better than all previous candidates).
Indicator Random Variable: Given a sample space and an event A, the indicator random variable for A is defined as:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur.} \end{cases}$$

Then:

$$[I\{A\}] = P(A)$$

Expected Number of Hires: Let X be the total number of hires. Define $X = {}_{i=1}^{n} I_i$, where $I_i = 1$ if the i-th candidate is hired (i.e., better than all previous $i - 1$), and 0 otherwise.

$$[X] = {}_{i=1}^{n} [I_i] = {}_{i=1}^{n} \frac{1}{i} = H_n$$

Conclusion: The expected number of hires is $\Theta(\log n)$, even though there are n candidates.

Quick Sort

Quick Sort is a divide-and-conquer algorithm with the following steps:

1. **Divide:** Partition $A[p, \ldots, r]$ into two (possibly empty) subarrays $A[p, \ldots, q-1]$ and $A[q+1, \ldots, r]$ such that each element in the first subarray is $\leq A[q]$ and each element in the second subarray is $\geq A[q]$.
2. **Conquer:** Recursively sort the two subarrays by calling Quick Sort on them.
3. **Combine:** No work is needed to combine the subarrays since the sorting is done in-place.

```
procedure Partition(A, p, r)
    x ← A[r]
    i ← p − 1
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            exchange A[i] with A[j]
        end if
    end for
    exchange A[i + 1] with A[r]
    return i + 1
end procedure
```

```
procedure Quick-Sort(A, p, r)
    if p < r then
        q ← Partition(A, p, r)
        Quick-Sort(A, p, q − 1)
        Quick-Sort(A, q + 1, r)
    end if
end procedure
```

```
procedure Randomized-Partition(A, p, r)
    i ← Random(p, r)
    exchange A[r] with A[i]
    return Partition(A, p, r)
end procedure
```

```
procedure Randomized-Quick-Sort(A, p, r)
    if p < r then
        q ← Randomized-Partition(A, p, r)
        Quick-Sort(A, p, q − 1)
        Quick-Sort(A, q + 1, r)
    end if
end procedure
```

Random Runtime: $\Theta(|N| \log |N|)$
Worst Runtime: $\Theta(|N|^2)$

Online Algorithms

An **online algorithm** processes its input piece-by-piece in a serial fashion, i.e., it does not have access to the entire input from the start. Instead, it must make decisions based only on the current and past inputs without knowledge of future inputs.
Characteristics:

- Decisions are made in real-time.
- Cannot revise past decisions once new input arrives.
- Often evaluated using *competitive analysis*, comparing performance to an optimal offline algorithm.

Competitive Ratio: If C_{online} is the cost incurred by the online algorithm and C_{opt} is the cost incurred by an optimal offline algorithm, then the competitive ratio is defined as:

$$\text{Competitive Ratio} = \max_{\text{input}} \frac{C_{\text{online}}}{C_{\text{opt}}}$$

An algorithm is said to be r-competitive if this ratio is at most r for all inputs.

Weighted Majority Algorithm

The **Weighted Majority Algorithm (WMA)** is an online learning algorithm that maintains a set of "experts" (prediction strategies), each assigned a weight. The algorithm predicts based on a weighted vote of the experts and penalizes those who make incorrect predictions.
Setup:

- n experts, each with an initial weight $w_i \leftarrow 1$.
- At each time step t, each expert i makes a prediction.
- The algorithm makes its own prediction based on a weighted majority.
- After the outcome is revealed, experts that predicted incorrectly are penalized by multiplying their weight by a factor $\beta \in (0, 1)$.

Guarantees: If there is an expert that makes at most m mistakes, then the number of mistakes made by the algorithm is at most:

$$M \leq (1 + \log n) \cdot m$$

(up to constant factors depending on β)
Use cases: Binary prediction problems, stock forecasting, game playing.

Hedge Algorithm

The **Hedge Algorithm** generalizes Weighted Majority to handle real-valued losses and probabilistic predictions. It is used in adversarial multi-armed bandit and online optimization settings.
Setup:

- n actions (or experts), each with weight $w_i^{(t)}$ at round t.
- At each time step, the algorithm picks a probability distribution $p^{(t)}$ over actions, where:
$$p_i^{(t)} = \frac{w_i^{(t)}}{{}_j w_j^{(t)}}$$
- After observing losses $\ell_i^{(t)} \in [0, 1]$, weights are updated as:
$$w_i^{(t+1)} = w_i^{(t)} \cdot e^{-\eta \ell_i^{(t)}}$$
where $\eta > 0$ is the learning rate.

Guarantees: For any expert i, the regret after T rounds is bounded by:

$$\text{Regret} \leq \eta T + \frac{\log n}{\eta}$$

Setting $\eta = \sqrt{\frac{\log n}{T}}$ gives regret of order $O(\sqrt{T \log n})$.
Use cases: Adversarial learning, portfolio selection, online convex optimization.