Algorithm Analysis and Design

Complete Reference Guide

Comprehensive Coverage of All Topics

August 5, 2025

Contents

1 Complexity Analysis and Recurrence Relations

1.1 Asymptotic Notations

Asymptotic notations describe the behavior of functions as input size approaches infinity.

1.1.1 Big-O Notation (O)

$$f(n) = O(g(n)) \text{ if } \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Definition: Upper bound - worst case complexity.

1.1.2 Big-Omega Notation (Omega)

$f(n) = \Omega(g(n))$ if $\exists c > 0, n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$
Definition: Lower bound - best case complexity.

1.1.3 Big-Theta Notation (Theta)

$f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
Definition: Tight bound - average case complexity.

1.1.4 Little-o Notation (o)

$$f(n) = o(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

1.1.5 Little-omega Notation (omega)

$f(n) = \omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

1.2 Growth of Functions

Common growth rates in ascending order:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

1.3 Methods to Solve Recurrence Relations

1.3.1 Substitution Method

Steps:

1. Guess the form of solution
2. Verify by mathematical induction
3. Solve for constants

Example: $T(n) = 2T(n/2) + n$
Guess: $T(n) = O(n \log n)$, so $T(n) \leq c \cdot n \log n$
Proof by induction:

$$\begin{aligned}
T(n) &= 2T(n/2) + n && (1)\\
&\leq 2c \cdot \frac{n}{2} \log(\frac{n}{2}) + n && (2)\\
&= cn(\log n - \log 2) + n && (3)\\
&= cn \log n - cn + n && (4)\\
&\leq cn \log n \text{ (if } c \geq 1) && (5)
\end{aligned}$$

1.3.2 Recursion Tree Method

Steps:

1. Draw recursion tree
2. Calculate cost at each level
3. Sum costs across all levels

Example: $T(n) = 2T(n/2) + n$

$$\begin{aligned}
&\text{Level 0: } n && (6)\\
&\text{Level 1: } 2 \cdot \frac{n}{2} = n && (7)\\
&\text{Level 2: } 4 \cdot \frac{n}{4} = n && (8)\\
&\vdots && (9)\\
&\text{Level } \log n : n \cdot 1 = n && (10)
\end{aligned}$$

Total cost: $n \cdot (\log n + 1) = O(n \log n)$

1.3.3 Master Method

For recurrences of the form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$
Let $c = \log_b a$
Case 1: If $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^c)$
Case 2: If $f(n) = \Theta(n^c \log^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^c \log^{k+1} n)$
Case 3: If $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$ and $af(n/b) \leq cf(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$

2 Graph Traversal Algorithms

2.1 Breadth First Search (BFS)

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

Algorithm 1 Breadth First Search

```
BFS(G, s)
Initialize queue Q and visited array
visited[s] ← true
Q.enqueue(s)
while Q is not empty do
    u ← Q.dequeue()
    Process vertex u
    for each vertex v adjacent to u do
        if visited[v] = false then
            visited[v] ← true
            Q.enqueue(v)
        end if
    end for
end while
```

Algorithm 2 Depth First Search

```
DFS(G, u)
visited[u] ← true
Process vertex u
for each vertex v adjacent to u do
    if visited[v] = false then
        DFS(G, v)
    end if
end for
```

2.2 Depth First Search (DFS)

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

2.3 Topological Sort

Application: DAG (Directed Acyclic Graph) ordering **Time Complexity:** $O(V + E)$

Algorithm 3 Topological Sort using DFS

```
Initialize stack S and visited array
for each vertex u in G do
    if visited[u] = false then
        TopologicalSortUtil(u, visited, S)
    end if
end for
Print contents of stack S
```

Algorithm 4 Topological Sort Utility

```
TopologicalSortUtil(u, visited, S)
visited[u] ← true
for each vertex v adjacent to u do
    if visited[v] = false then
        TopologicalSortUtil(v, visited, S)
    end if
end for
S.push(u)
```

2.4 Strongly Connected Components (Kosaraju's Algorithm)

Time Complexity: $O(V + E)$

Algorithm 5 Kosaraju's Algorithm

```
Perform DFS on original graph G and store vertices in stack by finish time
Create transpose graph G^T
Pop vertices from stack and perform DFS on G^T
Each DFS tree in step 3 is a strongly connected component
```

2.5 Euler Path and Circuit

Euler Path: Visits every edge exactly once **Euler Circuit:** Euler path that starts and ends at same vertex

Conditions:

- **Euler Circuit:** All vertices have even degree
- **Euler Path:** Exactly 0 or 2 vertices have odd degree

2.6 Articulation Points (Cut Vertices)

Definition: Vertex whose removal increases number of connected components

Algorithm 6 Tarjan's Algorithm for Articulation Points

```
Initialize disc[], low[], parent[], visited[]
time ← 0
for each vertex u do
    if visited[u] = false then
        APUtil(u)
    end if
end for
```

Key Formula: $low[u] = \min(low[u], low[v])$ for tree edges, $low[u] = \min(low[u], disc[v])$ for back edges

2.7 Bridge Detection

Definition: Edge whose removal increases number of connected components

Condition: Edge (u, v) is bridge if $low[v] > disc[u]$

2.8 Bi-connected Components

Definition: Maximal bi-connected subgraph

Algorithm 7 Bi-connected Components

```
Use modified Tarjan's algorithm
Maintain stack of edges
When articulation point found, pop edges until current edge
Each set of popped edges forms a bi-connected component
```

3 Shortest Path Algorithms

3.1 Dijkstra's Algorithm

Application: Single-source shortest path with non-negative weights **Time Complexity:** $O((V + E) \log V)$ with binary heap

3.2 Bellman-Ford Algorithm

Application: Single-source shortest path with negative weights **Time Complexity:** $O(VE)$

3.3 Floyd-Warshall Algorithm

Application: All-pairs shortest path **Time Complexity:** $O(V^3)$

Recurrence Relation:

$$dist^{(k)}[i][j] = \min(dist^{(k-1)}[i][j], dist^{(k-1)}[i][k] + dist^{(k-1)}[k][j])$$

Algorithm 8 Dijkstra's Algorithm

Initialize $dist[s] = 0$, $dist[v] = \infty$ for all $v \neq s$
Create min-heap Q with all vertices
while Q is not empty **do**
  $u \leftarrow ExtractMin(Q)$
  for each vertex v adjacent to u **do**
    if $dist[u] + weight(u, v) < dist[v]$ **then**
      $dist[v] \leftarrow dist[u] + weight(u, v)$
      $DecreaseKey(Q, v, dist[v])$
    end if
  end for
end while

Algorithm 9 Bellman-Ford Algorithm

Initialize $dist[s] = 0$, $dist[v] = \infty$ for all $v \neq s$
for $i = 1$ to $|V| - 1$ **do**
  for each edge (u, v) in E **do**
    if $dist[u] + weight(u, v) < dist[v]$ **then**
      $dist[v] \leftarrow dist[u] + weight(u, v)$
    end if
  end for
end for
// Check for negative cycles
for each edge (u, v) in E **do**
  if $dist[u] + weight(u, v) < dist[v]$ **then**
    return "Negative cycle detected"
  end if
end for

Algorithm 10 Floyd-Warshall Algorithm

Initialize $dist[i][j] = weight(i, j)$ if edge exists, ∞ otherwise
$dist[i][i] = 0$ for all i
for $k = 1$ to n **do**
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**
      if $dist[i][k] + dist[k][j] < dist[i][j]$ **then**
        $dist[i][j] \leftarrow dist[i][k] + dist[k][j]$
      end if
    end for
  end for
end for

3.4 Shortest Path in DAG

Time Complexity: $O(V + E)$

Algorithm 11 Shortest Path in DAG

```
Topologically sort the vertices
Initialize dist[s] = 0, dist[v] = ∞ for all v ≠ s
for each vertex u in topological order do
    for each vertex v adjacent to u do
        if dist[u] + weight(u, v) < dist[v] then
            dist[v] ← dist[u] + weight(u, v)
        end if
    end for
end for
```

4 Divide & Conquer Algorithms

4.1 Counting Inversions using Merge Sort

Inversion: Pair (i, j) where $i < j$ but $arr[i] > arr[j]$ **Time Complexity:** $O(n \log n)$

Algorithm 12 Count Inversions

```
MergeAndCount(arr, temp, left, mid, right)
i ← left, j ← mid + 1, k ← left, inv_count ← 0
while i ≤ mid and j ≤ right do
    if arr[i] ≤ arr[j] then
        temp[k++] ← arr[i++]
    else
        temp[k++] ← arr[j++]
        inv_count ← inv_count + (mid − i + 1)
    end if
end while
Copy remaining elements
return inv_count
```

4.2 Closest Pair of Points

Time Complexity: $O(n \log n)$

4.3 Computing A to the power k mod M using Divide and Conquer

Time Complexity: $O(\log k)$

Formula:

$$A^k = \begin{cases} 1 & \text{if } k = 0 \\ (A^{k/2})^2 & \text{if } k \text{ is even} \\ A \times A^{k-1} & \text{if } k \text{ is odd} \end{cases}$$

Algorithm 13 Closest Pair of Points

```
Sort points by x-coordinate
ClosestPairRec(P_x, P_y, n)
if n ≤ 3 then
    return brute force solution
end if
Divide points into left and right halves
d_l ← ClosestPairRec(P_x_left, P_y_left, n/2)
d_r ← ClosestPairRec(P_x_right, P_y_right, n/2)
d ← min(d_l, d_r)
Create strip of points within distance d from dividing line
Check distances in strip and update d if necessary
return d
```

Algorithm 14 Fast Modular Exponentiation

```
FastPower(A, k, M)
if k = 0 then
    return 1
end if
if k is even then
    temp ← FastPower(A, k/2, M)
    return (temp × temp) mod M
else
    return (A × FastPower(A, k − 1, M)) mod M
end if
```

4.4 Finding k-th Smallest Element (Quickselect)

Average Time Complexity: $O(n)$ **Worst Case:** $O(n^2)$

Algorithm 15 Randomized Quickselect

```
QuickSelect(arr, left, right, k)
if left = right then
    return arr[left]
end if
pivotIndex ← RandomizedPartition(arr, left, right)
if k = pivotIndex then
    return arr[k]
else if k < pivotIndex then
    return QuickSelect(arr, left, pivotIndex − 1, k)
else
    return QuickSelect(arr, pivotIndex + 1, right, k)
end if
```

5 Greedy Algorithms

5.1 Elements and Properties

Greedy Choice Property: Locally optimal choices lead to globally optimal solution
Optimal Substructure: Optimal solution contains optimal solutions to subproblems

5.2 Fractional Knapsack

Time Complexity: $O(n \log n)$

Algorithm 16 Fractional Knapsack

```
Calculate value-to-weight ratio for each item
Sort items by ratio in descending order
totalValue ← 0, currentWeight ← 0
for each item i in sorted order do
    if currentWeight + weight[i] ≤ capacity then
        currentWeight ← currentWeight + weight[i]
        totalValue ← totalValue + value[i]
    else
        remainingCapacity ← capacity − currentWeight
        totalValue ← totalValue + value[i] × remainingCapacity / weight[i]
        BREAK
    end if
end for
return totalValue
```

5.3 Job Scheduling with Deadline

Time Complexity: $O(n^2)$

Algorithm 17 Job Scheduling with Deadline

```
Sort jobs by profit in descending order
Initialize result[] array and slot[] boolean array
for each job i do
    for j = min(n, job[i].deadline) − 1 down to 0 do
        if slot[j] = false then
            result[j] ← i
            slot[j] ← true
            BREAK
        end if
    end for
end for
```

5.4 Minimum Spanning Tree

5.4.1 Prim's Algorithm

Time Complexity: $O(E \log V)$ with binary heap

Algorithm 18 Prim's Algorithm

```
Initialize key[v] = ∞ for all vertices, key[0] = 0
Create min-heap Q with all vertices
Initialize parent[] array
while Q is not empty do
    u ← ExtractMin(Q)
    for each vertex v adjacent to u do
        if v ∈ Q and weight(u, v) < key[v] then
            parent[v] ← u
            key[v] ← weight(u, v)
            DecreaseKey(Q, v, key[v])
        end if
    end for
end while
```

5.4.2 Kruskal's Algorithm

Time Complexity: $O(E \log E)$

6 Dynamic Programming

6.1 Basic Principles

Optimal Substructure: Optimal solution contains optimal solutions to subproblems
Overlapping Subproblems: Same subproblems are solved multiple times

Algorithm 19 Kruskal's Algorithm

```
Sort all edges by weight in ascending order
Initialize Union-Find data structure
MST ← ∅
for each edge (u, v) in sorted order do
    if Find(u) ≠ Find(v) then
        MST ← MST ∪ {(u, v)}
        Union(u, v)
        if |MST| = V − 1 then
            BREAK
        end if
    end if
end for
return MST
```

Comparison with Other Paradigms:

- **vs Divide & Conquer:** DP has overlapping subproblems, D&C has independent subproblems
- **vs Greedy:** DP considers all possibilities, Greedy makes locally optimal choices

6.2 Memoization vs Tabulation

Memoization: Top-down approach, store results of subproblems **Tabulation:** Bottom-up approach, build solution iteratively

6.3 Coin Change Problems

6.3.1 Minimum Coins

Time Complexity: $O(n \times amount)$

Algorithm 20 Minimum Coins

```
CoinChange(coins[], amount)
Initialize dp[0...amount] with ∞, dp[0] = 0
for i = 1 to amount do
    for each coin c in coins do
        if i ≥ c then
            dp[i] = min(dp[i], dp[i − c] + 1)
        end if
    end for
end for
return dp[amount]
```

Recurrence: $dp[i] = \min(dp[i], dp[i - coin] + 1)$ for all valid coins

6.3.2 Coin Change Ways

Algorithm 21 Number of Ways to Make Change

Initialize $dp[0...amount]$ with 0, $dp[0] = 1$
for each coin c in $coins$ **do**
  for $i = c$ to $amount$ **do**
    $dp[i] = dp[i] + dp[i - c]$
  end for
end for
return $dp[amount]$

6.4 Longest Increasing Subsequence (LIS)

Time Complexity: $O(n^2)$ basic DP, $O(n \log n)$ optimized

Algorithm 22 LIS using DP

Initialize $lis[0...n - 1]$ with 1
for $i = 1$ to $n - 1$ **do**
  for $j = 0$ to $i - 1$ **do**
    if $arr[i] > arr[j]$ and $lis[i] < lis[j] + 1$ **then**
      $lis[i] = lis[j] + 1$
    end if
  end for
end for
return $\max(lis[0...n - 1])$

Recurrence: $LIS[i] = \max(LIS[j] + 1)$ for all $j < i$ where $arr[j] < arr[i]$

6.5 Longest Common Subsequence (LCS)

Time Complexity: $O(mn)$

Algorithm 23 LCS using DP

Initialize $dp[0...m][0...n]$ with 0
for $i = 1$ to m **do**
  for $j = 1$ to n **do**
    if $X[i - 1] = Y[j - 1]$ **then**
      $dp[i][j] = dp[i - 1][j - 1] + 1$
    else
      $dp[i][j] = \max(dp[i - 1][j], dp[i][j - 1])$
    end if
  end for
end for
return $dp[m][n]$

Recurrence:

$$LCS[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS[i-1][j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(LCS[i-1][j], LCS[i][j-1]) & \text{otherwise} \end{cases}$$

6.6 0/1 Knapsack

Time Complexity: $O(nW)$

Algorithm 24 0/1 Knapsack

Initialize $dp[0...n][0...W]$ with 0
for $i = 1$ to n **do**
  for $w = 1$ to W **do**
    if $weight[i-1] \leq w$ **then**
      $dp[i][w] = \max(dp[i-1][w], dp[i-1][w - weight[i-1]] + value[i-1])$
    else
      $dp[i][w] = dp[i-1][w]$
    end if
  end for
end for
return $dp[n][W]$

Recurrence:

$$dp[i][w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ dp[i-1][w] & \text{if } weight[i-1] > w \\ \max(dp[i-1][w], dp[i-1][w - weight[i-1]] + value[i-1]) & \text{otherwise} \end{cases}$$

6.7 Matrix Chain Multiplication

Time Complexity: $O(n^3)$

Algorithm 25 Matrix Chain Multiplication

Initialize $dp[1...n][1...n]$ with 0 on diagonal
for length $l = 2$ to n **do**
  for $i = 1$ to $n - l + 1$ **do**
    $j = i + l - 1$
    $dp[i][j] = \infty$
    for $k = i$ to $j - 1$ **do**
      $cost = dp[i][k] + dp[k+1][j] + p[i-1] \times p[k] \times p[j]$
      if $cost < dp[i][j]$ **then**
        $dp[i][j] = cost$
      end if
    end for
  end for
end for
return $dp[1][n]$

Recurrence: $dp[i][j] = \min_{i \leq k < j}(dp[i][k] + dp[k+1][j] + p_{i-1} \times p_k \times p_j)$

6.8 Applications of Dynamic Programming

- Edit Distance (Levenshtein Distance)

- Subset Sum Problem
- Palindrome Partitioning
- Maximum Sum Subarray (Kadane's Algorithm)
- Optimal Binary Search Tree
- Travelling Salesman Problem (TSP)

7 Network Flow

7.1 Flow Networks

Flow Network: Directed graph $G = (V, E)$ with:

- Source vertex s (no incoming edges)
- Sink vertex t (no outgoing edges)
- Capacity function $c : E \to \mathbb{R}^+$

Flow Properties:

1. **Capacity Constraint:** $0 \leq f(u, v) \leq c(u, v)$ for all $(u, v) \in E$
2. **Flow Conservation:** $\sum_{v \in V} f(u, v) = 0$ for all $u \in V \setminus \{s, t\}$

Value of Flow: $|f| = \sum_{v \in V} f(s, v)$

7.2 Max-Flow Min-Cut Theorem

Cut: Partition of vertices (S, T) where $s \in S$ and $t \in T$
Capacity of Cut: $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$
Theorem: Maximum flow value equals minimum cut capacity $\max |f| = \min c(S, T)$

7.3 Ford-Fulkerson Method

Time Complexity: $O(E \cdot |f^*|)$ where $|f^*|$ is maximum flow

Algorithm 26 Ford-Fulkerson Method

```
Initialize flow f(u, v) = 0 for all edges (u, v)
while there exists augmenting path P from s to t in residual graph do
    Find bottleneck capacity c_f(P) = min{c_f(u, v) : (u, v) ∈ P}
    for each edge (u, v) in P do
        f(u, v) = f(u, v) + c_f(P)
        f(v, u) = f(v, u) − c_f(P)
    end for
end while
return flow f
```

Residual Graph: $G_f = (V, E_f)$ where $E_f = \{(u, v) : c_f(u, v) > 0\}$

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

Limitation: Can be slow with irrational capacities (infinite runtime possible)

7.4 Edmonds-Karp Algorithm

Improvement: Use BFS to find shortest augmenting path **Time Complexity:** $O(VE^2)$

Algorithm 27 Edmonds-Karp Algorithm

```
Initialize flow f(u,v) = 0 for all edges
while BFS finds augmenting path from s to t do
    path_flow = ∞
    Trace path and find minimum residual capacity
    Update flow along the path
    max_flow = max_flow + path_flow
end while
return max_flow
```

Algorithm 28 BFS for Augmenting Path

```
Initialize parent[] array and visited array
queue.push(s), visited[s] = true
while queue is not empty do
    u = queue.pop()
    for each vertex v adjacent to u do
        if visited[v] = false and c_f(u,v) > 0 then
            parent[v] = u
            visited[v] = true
            queue.push(v)
            if v = t then
                return true
            end if
        end if
    end for
end while
return false
```

7.5 Maximum Bipartite Matching

Bipartite Graph: $G = (U \cup V, E)$ where no edges within U or V

Reduction to Max Flow:

1. Create source s connected to all vertices in U with capacity 1
2. Create sink t connected from all vertices in V with capacity 1
3. Set capacity 1 for all original edges

Time Complexity: $O(VE)$

Algorithm 29 Maximum Bipartite Matching using DFS

```
Initialize match[] array with −1
result = 0
for each vertex u in left set do
    Initialize visited[] array with false
    if dfs(u, visited, match) then
        result = result + 1
    end if
end for
return result
```

Algorithm 30 DFS for Augmenting Path in Bipartite Matching

```
dfs(u, visited, match)
for each vertex v adjacent to u do
    if visited[v] = true then
        CONTINUE
    end if
    visited[v] = true
    if match[v] = −1 OR dfs(match[v], visited, match) then
        match[v] = u
        return true
    end if
end for
return false
```

7.6 Minimum Path Cover

Definition: Minimum number of vertex-disjoint paths that cover all vertices in DAG

Theorem: For DAG with n vertices, minimum path cover = $n-$ maximum matching in corresponding bipartite graph

Construction:

1. Create bipartite graph with left copy and right copy of vertices
2. Add edge (u_L, v_R) if (u, v) exists in original DAG
3. Find maximum bipartite matching

Formula: Minimum Path Cover = $|V| - |M|$ where M is maximum matching

7.7 Edge Cover

Definition: Set of edges such that every vertex is incident to at least one edge

Minimum Edge Cover:

- For connected graph: $|V| - |M|$ where M is maximum matching
- General formula: $|V| - c$ where c is number of connected components

Algorithm 31 Minimum Edge Cover

```
Find maximum matching M in graph G
cover = M
for each unmatched vertex v do
    Add any edge incident to v to cover
end for
return cover
```

8 Advanced Topics and Complexity Results

8.1 NP-Completeness

Decision Problems:

- **P:** Problems solvable in polynomial time
- **NP:** Problems verifiable in polynomial time
- **NP-Complete:** Hardest problems in NP
- **NP-Hard:** At least as hard as NP-complete problems

Famous NP-Complete Problems:

- 3-SAT (Boolean Satisfiability)
- Hamiltonian Path/Cycle
- Travelling Salesman Problem (Decision version)
- Knapsack Problem (Decision version)
- Graph Coloring
- Clique Problem
- Vertex Cover

8.2 Approximation Algorithms

Approximation Ratio: For minimization problem, algorithm A has ratio ρ if: $\frac{A(I)}{OPT(I)} \leq \rho$

Examples:

- **Vertex Cover:** 2-approximation using maximal matching
- **TSP:** 2-approximation using MST (metric TSP)
- **Set Cover:** H_n-approximation where $H_n = \sum_{i=1}^{n} \frac{1}{i}$

8.3 Parameterized Complexity

Fixed Parameter Tractable (FPT): Algorithm runs in $O(f(k) \cdot n^c)$ time where k is parameter

Examples:

- **Vertex Cover:** $O(2^k \cdot n)$ where k is size of vertex cover
- **Graph Coloring:** $O(2^k \cdot n)$ where k is number of colors

9 Summary of Time Complexities

Algorithm	Time Complexity	Space Complexity
BFS/DFS	$O(V+E)$	$O(V)$
Dijkstra	$O((V+E)\log V)$	$O(V)$
Bellman-Ford	$O(VE)$	$O(V)$
Floyd-Warshall	$O(V^3)$	$O(V^2)$
Kruskal	$O(E\log E)$	$O(V)$
Prim	$O((V+E)\log V)$	$O(V)$
Ford-Fulkerson	$O(E \cdot \lvert f^* \rvert)$	$O(V+E)$
Edmonds-Karp	$O(VE^2)$	$O(V+E)$
Merge Sort	$O(n\log n)$	$O(n)$
Quick Sort (avg)	$O(n\log n)$	$O(\log n)$
Heap Sort	$O(n\log n)$	$O(1)$
Counting Sort	$O(n+k)$	$O(k)$
Radix Sort	$O(d(n+k))$	$O(n+k)$

10 Important Recurrence Relations

$$T(n) = 2T(n/2) + O(n) \quad \Rightarrow O(n\log n) \text{ (Merge Sort)} \tag{11}$$

$$T(n) = 2T(n/2) + O(1) \quad \Rightarrow O(n) \text{ (Binary Search)} \tag{12}$$

$$T(n) = T(n-1) + O(n) \quad \Rightarrow O(n^2) \text{ (Selection Sort)} \tag{13}$$

$$T(n) = T(n-1) + O(1) \quad \Rightarrow O(n) \text{ (Linear Search)} \tag{14}$$

$$T(n) = 2T(n-1) + O(1) \quad \Rightarrow O(2^n) \text{ (Fibonacci)} \tag{15}$$

$$T(n) = 7T(n/2) + O(n^2) \quad \Rightarrow O(n^{\log_2 7}) \text{ (Strassen)} \tag{16}$$