Algorithms and Data Structures

3rd Semester Software Technology Engineering

Eduard Fekete

December 8, 2025

Contents

1 Introduction

Algorithms are formal, deterministic procedures transforming input to output through a finite sequence of well-defined steps. They encode computation as logic, not as syntax. What differentiates a good algorithm is not correctness alone but asymptotic efficiency under resource constraints: time, space, and often communication cost. In modern software systems, the majority of practical engineering failures originate not from correctness errors but from asymptotic ignorance. Understanding runtime growth is predictive power - it allows engineering before scaling breaks.

Data structures are engineered spatial encodings that give certain classes of algorithms structural leverage. Their entire purpose is to reduce entropy in access patterns: accelerating lookup, minimizing recomputation, avoiding redundancy, reducing cache misses, exploiting sparsity, and transforming unstructured data into shape. Lists, trees, heaps, graphs, hash tables, tries - these are not vocabulary items but complexity tradeoff mechanisms. Every structural choice rewrites the computational geometry of a problem.

The field is therefore not a set of memorized templates but a combinatorial design discipline. First principles create the ability to compute systematically: modeling problems as state machines, reducing problems to known primitives, selecting structure to match access semantics, and then proving bounds analytically.

2 Basics

2.1 Performance vs Complexity

- **Performance:** empirical measurement of resource usage (time, space) for specific input sizes and distributions
- **Complexity:** theoretical analysis of resource usage as input size approaches infinity, focusing on growth

2.2 Pseudocode conventions

- **Indentation** indicates block structure.
- **Loops:** counters typically start at 1.
- `A[i : j]` contains `A[i]`, `A[i + 1]`, ..., `A[j]`.
- Pass parameters to a procedure **by value**: the procedure receives its own copy of the parameters.
- Return multiple values at once without bundling them into an object.
- Use clear integer division in pseudocode (e.g. floor((low+high)/2)) and document any 0- or 1-based indexing convention at the top of each example.

2.2.1 Example:

```
SUM-ARRAY(A, n)
// this is a comment
1. sum = 0
2. for i = 1 to n
3.     sum = sum + A[i]
4. return sum
```

3 Time Complexity

3.1 Notations

Notation	Name	Meaning
O	Big O	Upper bound on growth rate
Ω	Big Omega	Lower bound on growth rate
Θ	Big Theta	Tight bound on growth rate
o	Little o	Strict upper bound on growth rate
ω	Little omega	Strict lower bound on growth rate

3.2 Definitions

$f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

$f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$

$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

$f(n) = o(g(n))$ if for any positive constant $c > 0$, there exists a constant n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$

$f(n) = \omega(g(n))$ if for any positive constant $c > 0$, there exists a constant n_0 such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$

3.3 Asymptotic Analysis

Step	Description
1.	Identify the dominant term
2.	Remove constant factors
3.	Express in standard form (in terms of input size n)

3.4 Common Growth Rates

Complexity	Growth Rate	Example Functions
$O(1)$	Constant	5, 42, -3
$O(\log n)$	Logarithmic	$\log n$, $\log_2 n$, $\log_{10} n$
$O(n)$	Linear	n, $3n + 2$, $0.5n - 7$
$O(n \log n)$	Linearithmic	$n \log n$, $2n \log_2 n$
$O(n^2)$	Quadratic	n^2, $3n^2 + 2n + 1$
$O(2^n)$	Exponential	2^n, 3^n
$O(n!)$	Factorial	$n!$

3.5 Using Limits

This is an informal approach, not a formal definition.

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = o(g(n))$
- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 < c < \infty$, then $f(n) = \Theta(g(n))$
- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f(n) = \omega(g(n))$

4 Binary Search

4.1 Pseudocode

```
BINARY-SEARCH(A, n, target)
1. low = 1
2. high = n
3. while low <= high
4.     mid = floor((low + high) / 2)  // Use floor for integer division
5.     if A[mid] == target
6.         return mid
7.     else if A[mid] < target
8.         low = mid + 1
9.     else
10.        high = mid - 1
11. return NOT-FOUND
```

4.2 Time complexity analysis

4.2.1 Average Case

- Each iteration halves the search space
- Number of iterations: $\log_2 n$
- Each iteration takes constant time $O(1)$
- Total time complexity: $O(\log n)$

4.2.2 Worst Case

- Target not found
- Number of iterations: $\log_2 n$
- Each iteration takes constant time $O(1)$
- Total time complexity: $O(\log n)$

4.2.3 Best Case

- Target found at the middle index
- Number of iterations: 1
- Each iteration takes constant time $O(1)$
- Total time complexity: $O(1)$

5 Recursion

A recursive algorithm is one that calls itself to solve smaller instances of the same problem until reaching a base case. Often used for so called "divide and conquer" algorithms.

Base case: the condition under which the recursion stops. **Recursive case:** the part of the algorithm where the function calls itself with modified parameters.

5.1 Example: Factorial

```
FACTORIAL(n)
1. if n == 0
2.     return 1
3. else
4.     return n * FACTORIAL(n - 1)
```

5.2 Time complexity analysis

- Each call to `FACTORIAL` makes one recursive call with `n - 1`
- Number of calls: $n + 1$ (from `n` down to `0`)
- Each call takes constant time $O(1)$
- Total time complexity: $O(n)$

5.3 Space complexity analysis*

- Each call to `FACTORIAL` adds a new frame to the call stack.
- Maximum depth of recursion: $n + 1$.
- Each frame takes constant space $O(1)$.
- Total space complexity: $O(n)$.

6 Lists

A list is a linear collection of elements with dynamic size.

6.1 ListADT Operations

Operation	Description
`insert(i, x)`	Insert element `x` at index `i`
`insert at end`	Append element `x` to the end of the list
`insert at front`	Prepend element `x` to the start of the list
`delete(i)`	Remove element at index `i`
`delete at end`	Remove last element of the list
`delete at front`	Remove first element of the list
`get(i)`	Retrieve element at index `i`
`set(i, x)`	Update element at index `i` to `x`
`size()`	Return the number of elements in the list

6.2 Singly Linked List

Each node contains:

- Data
- Next pointer

6.3 Doubly Linked List

Each node contains:

- Data
- Next pointer
- Previous pointer

6.4 Circular Linked List

Each node contains:

- Data
- Next pointer (last node points to first node)
- Previous pointer (for doubly circular linked list)

6.5 Array List

Elements are stored in a contiguous block of memory. When the array is full, a new larger array is created, and elements are copied over.

6.6 Time Complexity

Operation	Singly Linked List	Doubly Linked List	Array List
Access (get/set)	O(n)	O(n)	O(1)
Search	O(n)	O(n)	O(n)
Insertion (at i)	O(n)	O(n)	O(n) (O(1) amortized at end)
Deletion (at i)	O(n)	O(n)	O(n) (O(1) at end)

Note: For linked lists, insertion/deletion at known positions (e.g., head/tail) is O(1); otherwise, O(n) to locate the index. For Array List, insertion/deletion at arbitrary positions is O(n), O(1) amortized for append/remove at end.

7 Stacks and Queues

7.1 Stacks

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. Elements are added and removed from the top of the stack.

7.1.1 StackADT Operations

- `push(item)`: Add an item to the top of the stack.
- `pop()`: Remove and return the item at the top of the stack.
- `peek()`: Return the item at the top of the stack without removing it.
- `isEmpty()`: Check if the stack is empty.

7.1.2 Time Complexity

Operation	Time Complexity
`push`	O(1)
`pop`	O(1)
`peek`	O(1)
`isEmpty`	O(1)

7.2 Queues

A queue is a linear data structure that follows the First In First Out (FIFO) principle. Elements are added to the back and removed from the front.

7.2.1 QueueADT Operations

- `enqueue(item)`: Add an item to the back of the queue.
- `dequeue()`: Remove and return the item at the front of the queue.
- `peek()`: Return the item at the front of the queue without removing it.
- `isEmpty()`: Check if the queue is empty.

7.2.2 Time Complexity

Operation	Time Complexity
`enqueue`	O(1)
`dequeue`	O(1)
`peek`	O(1)
`isEmpty`	O(1)

8 Sets, Maps, and Tuples

8.1 Sets

A set is an abstract data type that can store unique values, without any particular order. Common operations include:

- Insertion
- Deletion
- Membership testing (Contains?)
- Proper subset testing (IsSubset?)

8.1.1 Boolean Operations on Sets

- Union
- Intersection
- Difference
- Symmetric Difference

For example, given two sets A and B:

- **Union (A $\cup$ B):** The set of elements that are in A, in B, or in both.
- **Intersection (A $\cap$ B):** The set of elements that are in both A and B.
- **Difference (A - B):** The set of elements that are in A but not in B.
- **Symmetric Difference (A δ B):** The set of elements that are in either A or B but not in both.

8.2 Maps

A map (or dictionary) is an abstract data type that stores key-value pairs. Keys must be unique. Common operations include:

- Insertion (Put)
- Deletion (Remove)
- Lookup (Get)

8.3 Tuples

A tuple is an ordered collection of elements, which can be of different types. Tuples are immutable, meaning their contents cannot be changed after creation. Common operations include:

- Accessing elements by index
- Unpacking elements into variables

9 Hashing and Hash Tables

9.1 Hash Function

A hash function is a function that takes an input (or 'key') and returns a fixed-size string of bytes. The output, typically a hash code, is used to index into an array (the hash table) to store the associated value.

9.1.1 What Makes a Good Hash Function?

- Deterministic: same input always produces the same output
- Uniform distribution: minimizes collisions
- Efficient to compute
- Minimizes collisions: different inputs should produce different outputs

9.2 Hash Table

A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

9.2.1 Load Factor

The load factor (α) of a hash table is defined as the ratio of the number of elements (n) to the number of buckets (m):

$$\alpha = \frac{n}{m}$$

A higher load factor increases the likelihood of collisions, which can degrade performance.

If the load factor exceeds a certain threshold (commonly 0.7), the hash table is typically resized (doubled in size) and all existing elements are rehashed to maintain efficient operations.

9.2.2 Collision Resolution Strategies

9.2.2.1 Chaining Chaining involves maintaining a list of all elements that hash to the same index. Each bucket in the hash table contains a collection of entries that map to that index.

Problem: If many keys hash to the same index, the collection (often implemented as a linked list) can become long, leading to O(n) search time in the worst case.

9.2.2.2 Open Addressing Open addressing involves finding another open slot within the array when a collision occurs. This can be done using various probing techniques. During searching, the same probing sequence is followed until the desired key is found or an empty slot is encountered.

9.2.2.2.1 Linear Probing In linear probing, when a collision occurs, the algorithm checks the next slot in the array (i.e., index + 1) and continues checking subsequent slots until an empty slot is found.

Formula: `new_index = (original_index + i) % table_size`, where `i` is the number of collisions encountered so far.

9.2.2.2.2 Quadratic Probing In quadratic probing, the algorithm checks slots at increasing quadratic distances from the original hashed index (i.e., index + 1^2, index + 2^2, index + 3^2, etc.) until an empty slot is found.

Formula: `new_index = (original_index + i^2) % table_size`, where `i` is the number of collisions encountered so far.

9.2.2.2.3 Double Hashing In double hashing, a second hash function is used to determine the step size for probing. When a collision occurs, the algorithm uses this second hash function to calculate the next index to check. If a collision occurs again, the step size is added to the current index to find the next slot.

Formula: `new_index = (original_index + i * hash2(key)) % table_size`, where `i` is the number of collisions encountered so far.

9.2.3 Time Complexity

Operation	Average Case	Worst Case
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)
Search	O(1)	O(n)

10 Trees

A tree is a hierarchical data structure consisting of nodes, where each node contains a value and references to its child nodes. The topmost node is called the root, and nodes without children are called leaves.

10.1 Terminology

Term	Description
Node	An element of the tree containing data
Edge	A connection between two nodes
Root	The topmost node of the tree
Leaf	A node with no children
Parent	A node that has children
Child	A node that has a parent
Sibling	Nodes that share the same parent
Subtree	A tree consisting of a node and its descendants
Height	The length of the longest path from a node to a leaf
Depth	The length of the path from the root to a node
Level	The depth of a node
Path	A sequence of nodes and edges connecting a node to a descendant

10.1.1 Full Tree

A full tree is a tree in which every node has either 0 or 2 children.

10.1.2 Complete Tree

A complete tree is a binary tree in which all levels are fully filled except possibly for the last level, which is filled from left to right.

10.1.3 Balanced Tree

A balanced tree is a binary tree in which the heights of the two child subtrees of any node differ by at most one.

10.1.4 Perfect Tree

A perfect tree is a binary tree in which all internal nodes have exactly two children and all leaves are at the same level.

10.2 Rotations

Rotations are operations that change the structure of the tree without violating the binary search tree properties. They are used to maintain balance after insertions and deletions.

10.3 N-ary Trees

An n-ary tree is a tree data structure in which each node can have at most n children.

10.4 Binary Trees

A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child. Binary trees are a special case of n-ary trees where $n = 2$.

10.4.1 Binary Tree ADT

Operation	Description
get root	Return the root node of the binary tree
is empty	Check if the binary tree is empty
size	Return the number of nodes in the tree
contains	Check if a value exists in the tree
insert	Insert a value into the binary tree
delete	Remove a value from the binary tree
in order	Traverse the tree in in-order fashion
pre order	Traverse the tree in pre-order fashion
post order	Traverse the tree in post-order fashion
level order	Traverse the tree in level-order fashion
height	Return the height of the binary tree

10.4.2 Binary Tree with Linked Nodes

Each node contains:

- Data
- Left child pointer
- Right child pointer
- Parent pointer (optional)

10.4.3 Binary Tree with Array Representation

- Root node at index 0
- For a node at index `i`:
  - Left child at index `2i + 1`
  - Right child at index `2i + 2`
  - Parent at index `(i - 1) / 2` (if `i > 0`)

10.4.4 Traversal Methods

- **In-order Traversal:** Visit left subtree, root, right subtree
- **Pre-order Traversal:** Visit root, left subtree, right subtree
- **Post-order Traversal:** Visit left subtree, right subtree, root
- **Level-order Traversal:** Visit nodes level by level from left to right

10.5 Binary Search Trees

A binary search tree (BST) is a binary tree that maintains the following properties:

- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- Both the left and right subtrees must also be binary search trees.

10.6 Red-Black Trees

A red-black tree is a self-balancing binary search tree where each node has an additional color attribute (red or black) that helps maintain balance during insertions and deletions. The following properties must be maintained:

1. Each node is either red or black.
2. The root is always black.

3. All leaves (NIL nodes) are black.
4. If a red node has children, then both children must be black (no two red nodes can be adjacent).
5. Every path from a node to its descendant leaves must contain the same number of black nodes.

10.6.1 Unique Properties

- **Black Height:** The number of black nodes from a node to its leaves is the same for all paths.
- **Balancing:** The longest path from the root to a leaf is no more than twice as long as the shortest path, ensuring logarithmic height.

10.6.2 Red-Black Tree Rotations

- **Left Rotation:** A left rotation on a node `x` makes `x`'s right child `y` the new root of the subtree, with `x` becoming the left child of `y`. The left subtree of `y` becomes the right subtree of `x`.
- **Right Rotation:** A right rotation on a node `y` makes `y`'s left child `x` the new root of the subtree, with `y` becoming the right child of `x`. The right subtree of `x` becomes the left subtree of `y`.

10.7 AVL Trees*

An AVL tree is a self-balancing binary search tree where the heights of the two child subtrees of any node differ by at most one. If the balance factor (height of left subtree - height of right subtree) becomes greater than 1 or less than -1, the tree is rebalanced through rotations.

10.7.1 AVL Tree Rotations

- **Single Right Rotation (LL Rotation):** Used when a node is inserted into the left subtree of the left child.
- **Single Left Rotation (RR Rotation):** Used when a node is inserted into the right subtree of the right child.
- **Left-Right Rotation (LR Rotation):** Used when a node is inserted into the right subtree of the left child.
- **Right-Left Rotation (RL Rotation):** Used when a node is inserted into the left subtree of the right child.

10.8 Tries*

A trie, or prefix tree, is a tree-like data structure that stores a dynamic set of strings, where each node represents a common prefix shared by some strings. Each edge corresponds to a character in the string, and the path from the root to a node represents a prefix of the strings stored in the trie.

10.8.1 Example Operations*

- Insertion: Build paths for each character in the string.
- Search: Traverse the trie following the string's characters; if a node marks the end of a word, it's found.
- Time Complexity: O(m), where m is the string length, independent of the number of strings stored.

10.9 Huffman Coding

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to input characters based on their frequencies. Characters that occur more frequently are assigned shorter codes, while less frequent characters are assigned longer codes. The result is a prefix-free binary code that minimizes the total number of bits used for encoding.

10.9.1 Huffman Tree Construction

1. Create a leaf node for each character and build a priority queue (min-heap) of all leaf nodes based on their frequencies.
2. While there is more than one node in the queue:
   - Remove the two nodes of the lowest frequency from the queue.
   - Create a new internal node with these two nodes as children and a frequency equal to the sum of their frequencies.
   - Insert the new node back into the priority queue.
3. The remaining node is the root of the Huffman tree.

11 Sorting Algorithms

Sorting algorithms arrange the elements of a list or array in a specific order, typically in ascending or descending numerical order or lexicographical order for strings.

11.1 Properties of Sorting Algorithms

Property	Description
Stability	A stable sort maintains the relative order of records with equal keys.
Adaptability	An adaptive sort takes advantage of existing order in the input to improve performance.
In-place	An in-place sort requires only a constant amount of additional memory.
Time Complexity	The amount of time taken to sort the elements, often expressed in Big O notation.
Space Complexity	The amount of memory space required to perform the sort.

11.2 Selection Sort

Selection sort is a simple comparison-based sorting algorithm. It works by dividing the input list into two parts: a sorted sublist and an unsorted sublist. The algorithm repeatedly selects the smallest (or largest) element from the unsorted sublist and moves it to the end of the sorted sublist.

11.2.1 Pseudocode

```
SELECTION-SORT(A, n)
1. for i = 1 to n - 1
2.     minIndex = i
3.     for j = i + 1 to n
4.         if A[j] < A[minIndex]
5.             minIndex = j
6.     swap A[i] with A[minIndex]
```

11.2.2 Properties

Property	Status
Time Complexity	O(n^2) in all cases
Space Complexity	O(1) (in-place)
Stability	Not stable
Adaptability	Not adaptive
In-place	Yes

11.3 Insertion Sort

Insertion sort is a simple comparison-based sorting algorithm that builds a sorted array one element at a time. It works by taking elements from the unsorted portion of the array and inserting them into their correct position in the sorted portion.

11.3.1 Pseudocode

```
INSERTION-SORT(A, n)
1. for i = 2 to n
2.     key = A[i]
3.     j = i - 1
```

```
4.      while j > 0 and A[j] > key
5.          A[j + 1] = A[j]
6.          j = j - 1
7.      A[j + 1] = key
```

11.3.2 Properties

Property	Status
Time Complexity	O(n^2) worst case, O(n) best case
Space Complexity	O(1) (in-place)
Stability	Stable
Adaptability	Adaptive
In-place	Yes

11.4 Merge Sort

Merge sort is a divide-and-conquer sorting algorithm that divides the input array into two halves, recursively sorts each half, and then merges the sorted halves back together.

11.4.1 Pseudocode

```
MERGE-SORT(A, p, r)
1. if p < r
2.      q = floor((p + r) / 2)
3.      MERGE-SORT(A, p, q)
4.      MERGE-SORT(A, q + 1, r)
5.      MERGE(A, p, q, r)
```

11.4.2 Properties

Property	Status
Time Complexity	O(n log n) in all cases
Space Complexity	O(n)
Stability	Stable
Adaptability	Not adaptive
In-place	No

11.5 Quick Sort

Quick sort is a divide-and-conquer sorting algorithm that selects a 'pivot' element from the array and partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

11.5.1 Pseudocode

```
// Sorts (a portion of) an array, divides it into partitions, then sorts those
algorithm quicksort(A, lo, hi) is
  // Ensure indices are in correct order
  if lo >= hi || lo < 0 then
    return

  // Partition array and get the pivot index
```

```
  p := partition(A, lo, hi)

  // Sort the two partitions
  quicksort(A, lo, p - 1) // Left side of pivot
  quicksort(A, p + 1, hi) // Right side of pivot

// Divides array into two partitions
algorithm partition(A, lo, hi) is
  pivot := A[hi] // Choose the last element as the pivot

  // Temporary pivot index
  i := lo

  for j := lo to hi - 1 do
    // If the current element is less than or equal to the pivot
    if A[j] <= pivot then
      // Swap the current element with the element at the temporary pivot index
      swap A[i] with A[j]
      // Move the temporary pivot index forward
      i := i + 1

  // Swap the pivot with the last element
  swap A[i] with A[hi]
  return i // the pivot index
```

11.5.2 Properties

Property	Status
Time Complexity	O(n log n) average case, O(nˆ2) worst case
Space Complexity	O(log n) due to recursion stack
Stability	Not stable
Adaptability	Not adaptive
In-place	Yes

11.6 Bubble Sort*

Bubble sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

11.6.1 Pseudocode

```
BUBBLE-SORT(A, n)
1. for i = 1 to n - 1
2.     for j = 1 to n - i
3.         if A[j] > A[j + 1]
4.             swap A[j] with A[j + 1]
```

11.6.2 Properties

Property	Status
Time Complexity	O(nˆ2) worst case, O(n) best case

Property	Status
Space Complexity	O(1) (in-place)
Stability	Stable
Adaptability	Adaptive
In-place	Yes

12 Divide and Conquer Algorithms

Divide and conquer is an algorithm design paradigm that works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

All divide and conquer algorithms follow three main steps:

1. **Divide:** Split the problem into smaller sub-problems.
2. **Conquer:** Solve each sub-problem recursively.
3. **Combine:** Merge the solutions of the sub-problems to solve the original problem.

12.0.1 Example: Maximum Subarray Problem

Find the contiguous subarray within a one-dimensional array of numbers that has the largest sum.

12.0.1.1 Pseudocode

```
MAX-SUBARRAY(A, low, high)
1. if low == high
2.     return A[low]
3. mid = floor((low + high) / 2)
4. left_sum = MAX-SUBARRAY(A, low, mid)
5. right_sum = MAX-SUBARRAY(A, mid + 1, high)
6. cross_sum = MAX-CROSSING-SUBARRAY(A, low, mid, high)
7. return max(left_sum, right_sum, cross_sum)
```

Time Complexity: O(n log n)

13 Greedy Algorithms

Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. They are efficient for problems with optimal substructure and the greedy choice property.

Checklist: before applying a greedy approach, verify - the problem has optimal substructure, and - a locally optimal choice can be shown (exchange/induction) to lead to a global optimum.

Common uses: scheduling, MST (Prim/Kruskal), shortest paths on non-negative graphs (Dijkstra), and priority-based construction (Huffman).

14 Graphs

A graph is a non-linear data structure consisting of nodes (vertices) connected by edges. Graphs can be directed or undirected, weighted or unweighted.

14.1 Terminology

- **Vertex (Node):** A fundamental unit of a graph.
- **Edge:** A connection between two vertices.
- **Directed Graph:** Edges have a direction.
- **Undirected Graph:** Edges have no direction.
- **Weighted Graph:** Edges have associated weights.
- **Degree:** Number of edges connected to a vertex (in-degree/out-degree for directed).
- **Path:** Sequence of vertices connected by edges.
- **Cycle:** A path that starts and ends at the same vertex.

14.2 Representations

- **Adjacency List:** List of lists where each list represents the neighbors of a vertex.
- **Adjacency Matrix:** 2D array where matrix[i][j] indicates an edge between i and j.

14.3 Traversal Algorithms

- **Depth-First Search (DFS):** Explores as far as possible along each branch before backtracking. Uses a stack (recursive or explicit). Time: O(V + E).
- **Breadth-First Search (BFS):** Explores all neighbors at the current depth before moving to the next level. Uses a queue. Time: O(V + E).

14.3.1 DFS Pseudocode

```
DFS(G, s)
1. mark s as visited
2. for each neighbor v of s
3.     if v not visited
4.         DFS(G, v)
```

14.3.2 BFS Pseudocode

```
BFS(G, s)
1. create queue Q
2. enqueue s, mark s visited
3. while Q not empty
4.     u = dequeue Q
5.     for each neighbor v of u
6.         if v not visited
7.             enqueue v, mark v visited
```

14.4 Minimum Spanning Tree (MST)

An MST connects all vertices in an undirected weighted graph with minimum total edge weight, without cycles. Used in network design, clustering, and infrastructure planning.

14.4.1 Prim's Algorithm

Greedy: starts from a vertex and repeatedly adds the smallest edge connecting the growing tree to a new vertex.

14.4.1.1 Pseudocode

```
PRIM(G, start)
1. initialize key[v] = infinity for all v, key[start] = 0
2. priority queue Q contains all vertices keyed by key[v]
3. while Q not empty
4.     u = extract min from Q
5.     add u to MST (or mark visited)
6.     for each neighbor v of u
7.         if v in Q and weight(u,v) < key[v]
8.             key[v] = weight(u,v)
9.             decrease-key(Q, v)
```

Typical time: O(E log V) with binary heap (or O((V+E) log V) depending on implementation details).

14.4.2 Kruskal's Algorithm*

Greedy: sorts edges by weight and adds them if they do not create a cycle (Union-Find).

Time: O(E log E) (sorting edges dominates).

14.5 Shortest Paths

14.5.1 Dijkstra's Algorithm

Greedy: finds shortest paths from a single source in graphs with non-negative edge weights. Not suitable for graphs with negative edge weights (use Bellman-Ford or Johnson's algorithm instead).

14.5.1.1 Pseudocode

```
DIJKSTRA(G, s)
1. for each v in V: dist[v] = infinity
2. dist[s] = 0
3. priority queue Q contains (v, dist[v])
4. while Q not empty
5.     (u, d) = extract min from Q
6.     if d > dist[u] continue
7.     for each neighbor v of u
8.         if dist[u] + weight(u,v) < dist[v]
9.             dist[v] = dist[u] + weight(u,v)
10.            decrease-key or insert (v, dist[v]) into Q
```

Typical time: O(E log V).

14.6 Topological Sorting

For directed acyclic graphs (DAGs): Orders vertices so all edges point forward. Used in scheduling.

14.6.0.1 Pseudocode (using DFS)

```
TOPOLOGICAL-SORT(G)
1. for each vertex u
2.     if u not visited
3.         DFS(u)  // modified to record finish times
4. output vertices in reverse finish order
```

Time: O(V + E).

14.7 Algorithm Comparison

Algorithm	Purpose	Greedy?	Time Complexity	Applicable To / Notes
DFS	Traversal/Exploration	No	O(V + E)	Any graph; use for reachability, components, cycle detection
BFS	Shortest path (unweighted)	No	O(V + E)	Unweighted graphs; level/order information
Prim	MST	Yes	O(E log V)	Prefer when using adjacency lists and starting from a node; good for dense graphs with appropriate heap
Kruskal	MST	Yes	O(E log E)	Prefer when edges are already listed/sorted; works well with Union-Find
Dijkstra	Shortest paths	Yes	O(E log V)	Non-negative weights only; use Bellman-Ford if negatives exist
Topological Sort	Ordering in DAGs	No	O(V + E)	Scheduling, dependency resolution in DAGs

15 Master Theorem

15.1 Recurrence equation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

- $a > 0$
- $b > 1$

15.2 Benchmark table

Benchmark	Name	Function
Non-recursive work	Driving function	$f(n) = n^d(\log n)^k$
Recursive work	Watershed function	$n^{\log_b a},\ w = \log_b a$

15.3 Master theorem idea

- Which term dominates time complexity of the recurrence?
- Is there a polynomial gap between:

Type	Meaning
Non-recursive work	divide, combine
Recursive work	conquer

15.4 Definitions

Driving function: $f(n) = n^d(\log n)^k$

Watershed exponent: $w = \log_b a$

15.5 Master theorem cases

Case no	Condition	Who dominates	Result for T(n)
1	$w > d$	Recursive - Driving function is smaller than watershed function by polynomial gap	$\Theta(n^w)$
2	$w = d$	Tie - Driving function and watershed function are asymptotically equal	$\Theta\big(n^d \log^{k+1} n\big)$
3	$w < d$	Non-recursive - Watershed function is smaller than driving function by polynomial gap	$\Theta\big(f(n)\big)$

15.6 Procedure

1. Identify a, b, $f(n)$
2. Compute watershed exponent w
3. Rewrite driving function $f(n)$
4. Compare w and d
5. Conclude $\rightarrow$ time complexity of recurrence equation

15.7 Recurrence Equation Simplification*

We start with the recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

We substitute $T(n/b)$ recursively:

$$T(n) = a\left[aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right] + f(n)$$
$$T(n) = a^2T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n)$$

After going down k levels:

$$T(n) = a^kT\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right)$$

Now we stop recursion when the problem size becomes constant:

$$\frac{n}{b^k} = 1 \quad \Rightarrow \quad k = \log_b(n)$$

Substituting $k = \lfloor \log_b n \rfloor$:

$$T(n) = a^{\log_b n} \cdot T(1) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

Since $T(1) = \Theta(1)$:

$$T(n) = \Theta\left(a^{\log_b n}\right) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

We rewrite $a^{\log_b n}$ algebraically using the identity $a^{\log_b n} = n^{\log_b a}$:

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

15.7.1 Special case: f(n) = $\Theta(n^d)$

$$T(n) = \Theta\left(n^{\log_b a}\right) + n^d \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{b^d}\right)^i$$

This sum is geometric. Let $r = \frac{a}{b^d}$:

- if $r < 1$: geometric sum converges $\Rightarrow T(n) = \Theta\left(n^d\right)$
- if $r = 1$: geometric sum is $\Theta(\log n) \Rightarrow T(n) = \Theta\left(n^d \log n\right)$
- if $r > 1$: geometric sum grows like $r^{\log_b n} = n^{\log_b r} \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$