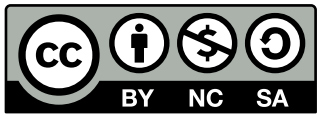
Data Structures and Algorithm Analysis

Kai Chen

December 23, 2025

Part of the SUSTech-Kai-Notes Initiative



This document was typeset using LaTeX.

Preface

These notes are compatible with the CS217 Data Structures and Algorithm Analysis (H) (Fall 2025) at SUSTech, and part of the course-notes-and-resources initiative: SUSTech-Kai-Notes.

Specific focus is placed on the mathematical analysis of algorithms, data structure implementation details, and complexity theory.

Contents

1 Getting Started

1.1 Foundations of Algorithms

1.1.1 Definitions

DEFINITION 1.1 (Algorithm)**.** An **algorithm** is a well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**. It acts as a tool for solving a well-specified computational problem.

Problem 1.1 (The Sorting Problem)**.** **Input**: A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$. **Output**: A permutation (reordering) $\langle a_1', a_2', ..., a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n$.

DEFINITION 1.2 (Instance)**.** An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution. For example, the sequence $(31, 41, 59, 26, 41, 58)$ is an instance of the sorting problem.

Remark 1.1. We describe algorithms using **pseudocode**. This serves two purposes:

1. To show that algorithms exist independently of any particular programming language.
2. To focus on ideas and logic rather than syntax issues or error-handling.

1.1.2 Correctness

DEFINITION 1.3 (Correctness)**.** An algorithm is **correct** if, for every input instance, it halts with the correct output. A correct algorithm solves the problem.

Ideally, algorithms should be accompanied by a proof of correctness, rather than just testing on a few instances.

1.2 The Computational Model

To analyze the running time of algorithms effectively, we need a model that abstracts away specific hardware details (like clock rate, memory hierarchy, or multi-core architecture).

DEFINITION 1.4 (RAM Model). The **Random-Access Machine (RAM)** model is a generic model of computation where instructions are executed one after another (no concurrent operations).

Property 1.1 (Elementary Operations). *The RAM model assumes that each elementary operation takes the same amount of time (a constant independent of the problem size). These operations include:*

- ***Arithmetic**: Add, subtract, multiply, divide, remainder, floor, ceiling.*
- ***Data Movement**: Variable assignments (storing, retrieving), copy.*
- ***Logical**: Comparisons, shifts, logical operations.*
- ***Control**: Conditional and unconditional branches (loops), subroutine calls and returns.*

1.3 Insertion Sort

1.3.1 The Concept

Insertion Sort works the way many people sort a hand of playing cards.

- We start with an empty left hand and the cards face down on the table.
- We remove one card at a time from the table and insert it into the correct position in the left hand.
- To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted.

1.3.2 The Algorithm

1.4 Correctness via Loop Invariants

To prove that an algorithm (especially one with loops) is correct, we use the technique of **Loop Invariants**.

Algorithm 1 Insertion Sort

```
1: procedure InsertionSort(A)
2:     for j = 2 to A.length do                    ▷ Iterate from the second element
3:         key = A[j]
4:                                  ▷ Insert A[j] into the sorted sequence A[1...j-1]
5:         i = j - 1
6:         while i > 0 and A[i] > key do
7:             A[i+1] = A[i]                       ▷ Shift element to the right
8:             i = i - 1
9:         end while
10:        A[i+1] = key                            ▷ Insert key into correct position
11:    end for
12: end procedure
```

DEFINITION 1.5 (Loop Invariant)**.** A loop invariant is a statement that is always true and reflects the progress of the algorithm towards producing a correct output.

To use a loop invariant to prove correctness, we must show three things:

1. **Initialization**: The invariant is true before the first iteration of the loop.
2. **Maintenance**: If the invariant is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination**: When the loop terminates, the invariant provides a useful property that helps show that the algorithm is correct.

1.4.1 Proof of Insertion Sort

THEOREM 1.2. *Algorithm 1 correctly sorts the array A.*

Proof. We use the following loop invariant:

> *"At the start of each iteration of the* ***for*** *loop (lines 1-8), the subarray* $A[1 \dots j-1]$ *consists of the elements originally in* $A[1 \dots j-1]$*, but in sorted order."*

Initialization: We start with $j = 2$. The subarray $A[1 \dots j-1]$ is $A[1 \dots 1]$, which consists of the single element $A[1]$. A single element is trivially sorted. Thus, the invariant holds.

Maintenance: The body of the **for** loop works by moving $A[j-1], A[j-2], \dots$ one position to the right until the proper position for $A[j]$ (stored in *key*) is found. The *key*

is then placed into this position. At this point, the subarray $A[1 \dots j]$ consists of the elements originally in $A[1 \dots j]$, but in sorted order. Incrementing j for the next iteration preserves the invariant.

Termination: The loop terminates when $j = n+1$. Substituting $n+1$ for j in the invariant, we have that the subarray $A[1 \dots n]$ consists of the elements originally in $A[1 \dots n]$, but in sorted order. Since $A[1 \dots n]$ is the entire array, the algorithm is correct. □

1.5 Runtime Analysis

We analyze the running time $T(n)$ by summing the cost of each statement multiplied by the number of times it is executed.

- Let c_k be the cost of executing line k.
- Let n be the size of the array A.
- Let t_j be the number of times the **while** loop test is executed for a given value of j.

1.5.1 Detailed Cost Breakdown

- Line 1 (for loop header): Executed n times (checking j from 2 to $n+1$). Cost: $c_1 n$.
- Line 2 ($key = A[j]$): Executed $n-1$ times. Cost: $c_2(n-1)$.
- Line 4 ($i = j-1$): Executed $n-1$ times. Cost: $c_4(n-1)$.
- Line 5 (while test): Executed $\sum_{j=2}^{n} t_j$ times. Cost: $c_5 \sum_{j=2}^{n} t_j$.
- Line 6 (loop body, shift): Executed $\sum_{j=2}^{n}(t_j - 1)$ times. Cost: $c_6 \sum_{j=2}^{n}(t_j - 1)$.
- Line 7 (loop body, decrement): Executed $\sum_{j=2}^{n}(t_j - 1)$ times. Cost: $c_7 \sum_{j=2}^{n}(t_j - 1)$.
- Line 8 (assignment): Executed $n-1$ times. Cost: $c_8(n-1)$.

The total running time $T(n)$ is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

1.5.2 Best Case Analysis

The **best case** occurs when the array is already sorted. In this scenario, $A[i] \leq key$ immediately for every j. Thus, the **while** loop test is executed only once ($t_j = 1$), and the loop body is never executed.

Substituting $t_j = 1$:

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} 1 = n - 1$$

$$\sum_{j=2}^{n} (t_j - 1) = 0$$

The runtime becomes:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

This is a **linear function** of n: $T(n) = an + b = \Theta(n)$.

1.5.3 Worst Case Analysis

The **worst case** occurs when the array is reverse sorted. In this scenario, we must compare the key with every element in the sorted subarray $A[1 \ldots j-1]$. Thus, $t_j = j$.

We use the following summation formulas:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Substituting these into the total runtime equation:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

Grouping terms by powers of n:

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

This is a **quadratic function** of n: $T(n) = an^2 + bn + c = \Theta(n^2)$.

2 Runtime and Asymptotic Notation

2.1 Recap: Runtime of Insertion Sort

In the previous lecture, we derived the running time function $T(n)$ for Insertion Sort on an input of size n. The exact formula depended on specific costs c_i for each machine instruction:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + \dots$$

This level of detail is often too messy and hardware-dependent. We observed two extreme cases:

- **Best Case** (Sorted Array): $T(n) = an + b$ (Linear).
- **Worst Case** (Reverse Sorted Array): $T(n) = an^2 + bn + c$ (Quadratic).

2.1.1 Why Focus on the Worst Case?

1. **Guarantee**: The worst-case runtime gives us an absolute upper bound. The algorithm will never take longer than this.
2. **Frequency**: For some algorithms, the worst case occurs fairly often (e.g., searching for a non-existent item in a database).
3. **Average Case**: Often, the "average" case is roughly as bad as the worst case. For Insertion Sort, the average case is also quadratic.

2.2 Asymptotic Analysis

2.2.1 The Philosophy of Asymptotics

To compare algorithms effectively, we need a metric that is independent of:

- The specific machine (CPU speed, memory architecture).
- The programming language or compiler efficiency.
- Small input sizes (where most algorithms are fast enough).

Key Idea: We focus on the **rate of growth** of the running time as the input size n approaches infinity ($n \to \infty$).

- We ignore **lower-order terms**: For large n, the highest power dominates (e.g., in $n^2 + 100n$, the n^2 term is overwhelmingly larger).
- We ignore **leading constants**: Constants depend on hardware. We want to know if the algorithm scales linearly, quadratically, etc.

2.3 Asymptotic Notations

We use five standard notations to describe the asymptotic behavior of functions. Let $f(n)$ be the algorithm's runtime and $g(n)$ be a reference function.

2.3.1 Big-Theta Notation (Θ): Tight Bound

DEFINITION 2.1 (Big-Theta Θ)**.** For a given function $g(n)$, $\Theta(g(n))$ is the set of functions:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0 \text{ and } n_0 \geq 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

Intuition: For sufficiently large n, $f(n)$ is "sandwiched" between two constant multiples of $g(n)$.

- We say $f(n)$ grows **at the same rate** as $g(n)$.
- Example: $2n^2 + 3n + 1 = \Theta(n^2)$.
- Note: We often write $f(n) = \Theta(g(n))$ (equality) to mean $f(n) \in \Theta(g(n))$ (set membership).

2.3.2 Big-O Notation (O): Upper Bound

DEFINITION 2.2 (Big-O O)**.** For a given function $g(n)$, $O(g(n))$ is the set of functions:

$$O(g(n)) = \{f(n) : \exists c > 0 \text{ and } n_0 \geq 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Intuition: $f(n)$ grows **no faster than** $g(n)$. It gives an asymptotic **ceiling**.

- Note: $f(n) = \Theta(g(n)) \implies f(n) = O(g(n))$.
- Ideally, we want the tightest upper bound, but loose bounds are technically correct (e.g., $2n = O(n^2)$ is true, but weak).

2.3.3 Big-Omega Notation (Ω): Lower Bound

DEFINITION 2.3 (Big-Omega Ω). For a given function $g(n)$, $\Omega(g(n))$ is the set of functions:

$$\Omega(g(n)) = \{f(n) : \exists c > 0 \text{ and } n_0 \geq 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Intuition: $f(n)$ grows **at least as fast as** $g(n)$. It gives an asymptotic **floor**.

THEOREM 2.1 (Relationship Theorem). *For any two functions $f(n)$ and $g(n)$:*

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ AND } f(n) = \Omega(g(n))$$

2.3.4 Strict Bounds (o and ω)

While O and Ω correspond to $\leq$ and $\geq$, the notations o and ω correspond to $<$ and $>$.

DEFINITION 2.4 (Little-o o). $f(n) = o(g(n))$ if for **any** constant $c > 0$, there exists $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$. Alternatively defined by limits:

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

Meaning: $f(n)$ becomes insignificant relative to $g(n)$ as n grows.

DEFINITION 2.5 (Little-omega ω). $f(n) = \omega(g(n))$ if for **any** constant $c > 0$, there exists $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$. Alternatively:

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

Meaning: $f(n)$ grows strictly faster than $g(n)$.

2.4 Analogy with Real Numbers

We can map the asymptotic relationships between functions to comparison operators for real numbers:

Notation	Analogy	Intuition
$f(n) = O(g(n))$	$f \leq g$	Upper bound (tight or loose)
$f(n) = \Omega(g(n))$	$f \geq g$	Lower bound (tight or loose)
$f(n) = \Theta(g(n))$	$f = g$	Tight bound (equal growth rate)
$f(n) = o(g(n))$	$f < g$	Strictly smaller growth
$f(n) = \omega(g(n))$	$f > g$	Strictly larger growth

Table 1: Comparison of Asymptotic Notations

2.5 Properties and Growth Hierarchy

2.5.1 Properties

- **Transitivity**: If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$. (Holds for all 5 notations).
- **Reflexivity**: $f(n) = \Theta(f(n))$. (Holds for O, Ω).
- **Symmetry**: $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$.
- **Transpose Symmetry**: $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

2.5.2 Common Growth Functions

Ordered from slowest to fastest growth:

1. $O(1)$: Constant time.
2. $O(\lg n)$: Logarithmic time.
3. $O(\sqrt{n})$: Square root.
4. $O(n)$: Linear time.
5. $O(n \lg n)$: Linearithmic (often seen in efficient sorting).
6. $O(n^2)$: Quadratic.

7. $O(n^3)$: Cubic.
8. $O(2^n)$: Exponential.
9. $O(n!)$: Factorial.

Remark 2.1 (Dominance Rules).

- **Polylogarithms vs Polynomials**: For any constants $a, b > 0$, $(\lg n)^a = o(n^b)$. (Any positive polynomial power beats any polylog).
- **Polynomials vs Exponentials**: For any constants $a > 0, b > 1$, $n^a = o(b^n)$. (Any exponential with base > 1 beats any polynomial).

2.6 Application to Insertion Sort

Let $T(n)$ be the running time of Insertion Sort on an array of size n.

- **Worst Case**: Occurs when the array is reverse sorted.

$$T(n) = an^2 + bn + c \implies T(n) = \Theta(n^2)$$

- **Best Case**: Occurs when the array is already sorted.

$$T(n) = an + b \implies T(n) = \Theta(n)$$

Note 2.1 (Common Misconception). It is **incorrect** to say "The running time of Insertion Sort is $\Theta(n^2)$". Why? Because for the best-case input, the runtime is linear, not quadratic.

- **Correct Statement 1**: The *worst-case* running time is $\Theta(n^2)$.
- **Correct Statement 2**: For *any* input, the running time is $O(n^2)$ (upper bound) and $\Omega(n)$ (lower bound).

3 Divide-and-Conquer

3.1 The Paradigm Shift

In previous lectures, we explored **Insertion Sort**, which follows an **Incremental Approach**: we process elements one by one and insert them into a growing sorted subarray. To break the $\Theta(n^2)$ barrier, we need a fundamentally different strategy: **Divide-and-Conquer**.

3.1.1 Divide-and-Conquer Strategy

This paradigm involves three recursive steps:

1. **Divide**: Break the problem into several smaller subproblems that are similar to the original problem but smaller in size.
2. **Conquer**: Solve the subproblems recursively. If the subproblem sizes are small enough (the base case), solve them directly.
3. **Combine**: Merge the solutions to the subproblems to create a solution to the original problem.

3.1.2 Motivating Example: Binary Search

Consider finding a number x in a **sorted** array of size n.

- **Linear Search**: Scanning from start to end takes $\Theta(n)$ in the worst case.
- **Binary Search**: Check the middle element. If x is smaller, discard the right half; otherwise, discard the left half.

$$T(n) = T(n/2) + \Theta(1) \implies T(n) = \Theta(\lg n)$$

Insight: By dividing the problem size by a constant factor (2) at each step, we exponentially reduce the work, achieving logarithmic complexity.

3.2 Merge Sort

Merge Sort is the archetypal divide-and-conquer sorting algorithm.

3.2.1 The Algorithm

1. **Divide**: Split the n-element sequence into two subsequences of size $n/2$.
2. **Conquer**: Recursively sort the two subsequences using Merge Sort.
3. **Combine**: Merge the two sorted subsequences to produce the final sorted answer.

Base Case: A sequence of length 1 is trivially sorted.

3.2.2 The Merge Procedure

The core logic lies in the **Combine** step. The procedure `Merge(A, p, q, r)` assumes that the subarrays $A[p \dots q]$ and $A[q+1 \dots r]$ are already sorted. It merges them into a single sorted subarray $A[p \dots r]$.

Mechanism (Two Pointers): We view the two subarrays as piles of cards face up. We compare the top cards of each pile, pick the smaller one, place it in the output pile, and repeat. To avoid checking for empty piles constantly, we can use a **sentinel value** (∞) at the end of each subarray.

Runtime of Merge: The procedure performs a constant amount of work for each element in the subarrays. Specifically, the **for** loop runs $n = r - p + 1$ times.

$$T_{\text{merge}}(n) = \Theta(n)$$

3.2.3 Correctness of Merge

We prove correctness using a **Loop Invariant**:

> *At the start of each iteration of the **for** loop, the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of L and R, in sorted order. Furthermore, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied*

Algorithm 2 Merge Procedure

```
1: procedure MERGE(A, p, q, r)
2:     n_1 = q − p + 1
3:     n_2 = r − q
4:     Let L[1 . . . n_1 + 1] and R[1 . . . n_2 + 1] be new arrays
5:     for i = 1 to n_1 do
6:         L[i] = A[p + i − 1]
7:     end for
8:     for j = 1 to n_2 do
9:         R[j] = A[q + j]
10:    end for
11:    L[n_1 + 1] = ∞; R[n_2 + 1] = ∞                    ▷ Sentinels
12:    i = 1; j = 1
13:    for k = p to r do
14:        if L[i] ≤ R[j] then
15:            A[k] = L[i]
16:            i = i + 1
17:        else
18:            A[k] = R[j]
19:            j = j + 1
20:        end if
21:    end for
22: end procedure
```

back to A.

- **Initialization**: For $k = p$, the subarray $A[p \ldots p-1]$ is empty, which trivially satisfies the invariant.
- **Maintenance**: Suppose $L[i] \leq R[j]$. Then $L[i]$ is the smallest uncopied element. We copy it to $A[k]$. Now $A[p \ldots k]$ contains the $k-p+1$ smallest elements. Incrementing k and i maintains the invariant.
- **Termination**: When the loop ends, $k = r + 1$. The subarray $A[p \ldots r]$ contains all elements (except sentinels) in sorted order.

3.3 Analysis of Merge Sort

3.3.1 Recurrence Relation

Let $T(n)$ be the time to sort n numbers.

- **Divide**: Computing the middle index takes constant time $\Theta(1)$.

- **Conquer**: We recursively solve two subproblems, each of size $n/2$, contributing $2T(n/2)$.
- **Combine**: The `Merge` procedure takes time linear in the number of elements, $\Theta(n)$.

Thus, the recurrence is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

3.3.2 Recursion Tree Visualization

To solve $T(n) = 2T(n/2) + cn$, visualize a tree:

- **Top Level**: Cost cn.
- **Level 1**: Two subproblems of size $n/2$. Cost $2 \times c(n/2) = cn$.
- **Level 2**: Four subproblems of size $n/4$. Cost $4 \times c(n/4) = cn$.
- ...
- **Depth**: The tree splits until $n = 1$. The height is $\lg n$.
- **Total Cost**: There are $\lg n + 1$ levels, each costing cn.

$$\text{Total} \approx cn \times \lg n = \Theta(n \lg n)$$

3.3.3 Comparison with Insertion Sort

- **Time**: Merge Sort is $\Theta(n \lg n)$ in the worst, best, and average cases. This is asymptotically faster than Insertion Sort's worst case $\Theta(n^2)$.
- **Space**: Merge Sort is **not** in-place. It requires $\Theta(n)$ auxiliary space for the 'L' and 'R' arrays in the 'Merge' step. Insertion Sort is in-place ($O(1)$ space). This is the main drawback of Merge Sort.

3.4 The Master Theorem

The Master Theorem provides a "cookbook" method to solve recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ (number of subproblems), $b > 1$ (shrinkage factor), and $f(n)$ (cost of divide/combine).

3.4.1 The Watershed Function

We compare the driving function $f(n)$ with the **watershed function** $n^{\log_b a}$.

- $n^{\log_b a}$ represents the rate of growth of the leaves (the cost of the base cases).
- $f(n)$ represents the cost of the work done at the root (divide and combine).

It's a "tug-of-war" between the root work and the leaf work.

3.4.2 The Three Cases

THEOREM 3.1 (Master Theorem). 1. ***Case 1 (Leaves Dominate)**: If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then the recursion cost dominates:*

$$T(n) = \Theta(n^{\log_b a})$$

2. ***Case 2 (Balanced)**: If $f(n) = \Theta(n^{\log_b a} \lg^k n)$ (typically $k = 0$), then the cost is distributed evenly across levels:*

$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$$

3. ***Case 3 (Root Dominates)**: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if the regularity condition $af(n/b) \leq cf(n)$ holds, then the root work dominates:*

$$T(n) = \Theta(f(n))$$

3.4.3 Examples

Example 1: Merge Sort

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

- $a = 2, b = 2, f(n) = n$.
- Watershed: $n^{\log_2 2} = n^1 = n$.
- Compare: $f(n) = \Theta(\text{Watershed})$. This is **Case 2** (with $k = 0$).
- Result: $T(n) = \Theta(n \lg n)$.

Example 2

$$T(n) = 9T(n/3) + n$$

- $a = 9, b = 3, f(n) = n$.
- Watershed: $n^{\log_3 9} = n^2$.
- Compare: $f(n) = n = O(n^{2-\epsilon})$. Watershed dominates. This is **Case 1**.
- Result: $T(n) = \Theta(n^2)$.

Example 3

$$T(n) = 3T(n/4) + n \lg n$$

- $a = 3, b = 4, f(n) = n \lg n$.
- Watershed: $n^{\log_4 3} \approx n^{0.793}$.
- Compare: $f(n) = \Omega(n^{0.793+\epsilon})$. Root dominates. This is **Case 3**.
- Regularity: $3(n/4)\lg(n/4) \leq cn \lg n$ holds for $c \approx 3/4$.
- Result: $T(n) = \Theta(n \lg n)$.

4 Heapsort

4.1 Motivation: Smarter Selection

To understand Heapsort, let's revisit **Selection Sort**.

- **Selection Sort Strategy**: Repeatedly find the maximum element in the remaining unsorted array and move it to the correct position.
- **The Inefficiency**: Finding the maximum takes $\Theta(n)$ time because we scan the array linearly. We do this n times, leading to $\Theta(n^2)$.
- **The Waste**: In every iteration, Selection Sort "forgets" all the comparisons it made. It starts from scratch.

Heapsort's Insight: Can we use a data structure to "memorize" the comparison results?

- Instead of scanning linearly, we organize elements so that finding the maximum is trivial ($O(1)$).
- Removing the maximum and "repairing" the structure should be fast ($O(\lg n)$).
- This turns the "Rebuild" process (Selection Sort) into a "Repair" process (Heapsort).

4.2 The Heap Data Structure

4.2.1 Definition

A **Heap** (specifically, a binary heap) is an array object that we view as a **nearly complete binary tree**.

- **Structure**: The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- **Implicit Representation**: We don't use pointers. The tree structure is implicit in the array indices.

For a node at index i:

- Parent$(i) = \lfloor i/2 \rfloor$

- Left$(i) = 2i$
- Right$(i) = 2i + 1$

4.2.2 The Heap Property

There are two kinds of heaps:

1. **Max-Heap**: For every node i other than the root:

$$A[\text{Parent}(i)] \geq A[i]$$

   The largest element is stored at the root. (Used for Heapsort).

2. **Min-Heap**: For every node i other than the root:

$$A[\text{Parent}(i)] \leq A[i]$$

   The smallest element is stored at the root. (Used for Priority Queues).

4.3 Core Operations

4.3.1 Maintaining the Property: Max-Heapify

Problem: Suppose the binary trees rooted at Left(i) and Right(i) are max-heaps, but $A[i]$ might be smaller than its children, violating the max-heap property. **Solution**: Let the value at $A[i]$ "float down" until the property is restored.

Runtime: The running time $T(n)$ depends on the height of the node h. In the worst case, we traverse from the root to a leaf.

$$T(n) = O(h) = O(\lg n)$$

Algorithm 3 Max-Heapify

```
1: procedure MAX-HEAPIFY(A, i)
2:     l = Left(i)
3:     r = Right(i)
4:     largest = i
5:     if l ≤ A.heap-size and A[l] > A[largest] then
6:         largest = l
7:     end if
8:     if r ≤ A.heap-size and A[r] > A[largest] then
9:         largest = r
10:    end if
11:    if largest ≠ i then
12:        Exchange A[i] with A[largest]
13:        MAX-HEAPIFY(A, largest)          ▷ Recursive call on the affected subtree
14:    end if
15: end procedure
```

4.3.2 Building the Heap: Build-Max-Heap

We can convert an arbitrary array $A[1 \dots n]$ into a max-heap by calling `Max-Heapify` in a bottom-up manner. **Observation**: The elements in the subarray $A[\lfloor n/2 \rfloor + 1 \dots n]$ are all leaves. Leaves are trivially max-heaps of size 1. We start from their parents and work up to the root.

Algorithm 4 Build-Max-Heap

```
1: procedure BUILD-MAX-HEAP(A)
2:     A.heap-size = A.length
3:     for i = ⌊A.length/2⌋ downto 1 do
4:         MAX-HEAPIFY(A, i)
5:     end for
6: end procedure
```

Runtime Analysis (Crucial): A naive bound is $O(n \lg n)$ because we call `Max-Heapify` $O(n)$ times. However, this is not tight.

- Most nodes have small heights.
- `Max-Heapify` takes time proportional to the height h of the node.
- A heap of size n has at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

Total work:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Using the summation formula $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$, we get:

$$T(n) = O(n \cdot 2) = O(n)$$

Conclusion: We can build a heap in **linear time**.

4.4 The Heapsort Algorithm

Now we combine the pieces.

1. Build a max-heap from the input array. ($O(n)$)
2. The maximum element is at $A[1]$. Swap it with $A[n]$ (placing it in its final sorted position).
3. Decrement the heap size (discarding the sorted element).
4. The new root likely violates the heap property. Call `Max-Heapify` to fix it. ($O(\lg n)$)
5. Repeat until the heap size is 1.

Algorithm 5 Heapsort

```
1: procedure HEAPSORT(A)
2:     BUILD-MAX-HEAP(A)
3:     for i = A.length downto 2 do
4:         Exchange A[1] with A[i]
5:         A.heap-size = A.heap-size − 1
6:         MAX-HEAPIFY(A, 1)
7:     end for
8: end procedure
```

4.4.1 Analysis

- **Build-Heap**: $O(n)$.
- **Loop**: Executes $n - 1$ times.
- **Heapify**: Takes $O(\lg n)$ each time.
- **Total Runtime**: $O(n) + (n-1)O(\lg n) = O(n \lg n)$.

4.4.2 Rebuild vs. Repair

- **Selection Sort** (Rebuild): After extracting the max, it sees the remaining $n-1$ elements as a chaotic bag. It spends $O(n)$ to find the next max.
- **Heapsort** (Repair): After extracting the max, it knows the remaining structure is *almost* a valid heap. Only one path from root to leaf is violated. It spends $O(\lg n)$ to repair this specific violation.

4.5 Priority Queues

Heaps are the underlying data structure for efficient Priority Queues. A Max-Priority Queue supports:

- `Insert(S, x)`: Insert element x into set S. ($O(\lg n)$)
- `Maximum(S)`: Return the element with the largest key. ($O(1)$)
- `Extract-Max(S)`: Remove and return the element with the largest key. ($O(\lg n)$)
- `Increase-Key(S, x, k)`: Increase the value of element x's key to k. ($O(\lg n)$ - element floats up).

4.6 Summary

Algorithm	Time (Worst)	Time (Best)	Space	Stable?
Insertion Sort	$O(n^2)$	$O(n)$	$O(1)$	Yes
Merge Sort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$	Yes
Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(1)$	No

Table 2: Heapsort combines the efficiency of Merge Sort with the space efficiency of Insertion Sort.

5 Quicksort

5.1 The Divide-and-Conquer Paradigm Revisited

Quicksort is a sorting algorithm that fits within the divide-and-conquer paradigm, but it represents a philosophical mirror image to Merge Sort.

5.1.1 Structural Comparison

- **Merge Sort**:
  - **Divide**: Trivial. Simply slice the indices in half ($O(1)$).
  - **Conquer**: Recursively sort both halves.
  - **Combine**: **Hard**. The heavy lifting is done here, merging two sorted lists ($O(n)$).
- **Quicksort**:
  - **Divide**: **Hard**. The heavy lifting is done here via `Partition`. We rearrange elements relative to a pivot ($O(n)$).
  - **Conquer**: Recursively sort the subarrays defined by the partition.
  - **Combine**: Trivial. No work is needed; the array is sorted in place ($O(1)$).

5.2 Partitioning: The Core Mechanism

The heart of Quicksort is the `Partition` procedure. It selects a **pivot** element and rearranges the array so that:

- Elements smaller than or equal to the pivot are on the left.
- Elements larger than the pivot are on the right.
- The pivot is placed in its correct sorted position.

5.2.1 The Lomuto Partition Scheme

We maintain two pointers to define dynamic regions in the array:

- i: The boundary of the "Small Region" (elements $\leq x$).
- j: The scanning pointer (current element being inspected).

Algorithm 6 Partition (A, p, r)

```
1: procedure PARTITION(A, p, r)
2:     x = A[r]                          ▷ Select the last element as pivot
3:     i = p − 1                         ▷ Small region is initially empty
4:     for j = p to r − 1 do             ▷ Scan the array
5:         if A[j] ≤ x then
6:             i = i + 1                 ▷ Expand the small region
7:             Exchange A[i] with A[j]   ▷ Swap element into the small region
8:         end if
9:     end for
10:    Exchange A[i + 1] with A[r]       ▷ Place pivot between regions
11:    return i + 1                      ▷ Return pivot index
12: end procedure
```

5.2.2 Trace Example

Let's trace the execution on the array $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

- **Setup**: Pivot $x = 4$. i starts at index 0 (before array).
- **j=1 (2)**: $2 \leq 4$. $i \to 1$. Swap $A[1] \leftrightarrow A[1]$. Array: $[2, 8, 7, \dots]$.
- **j=2 (8)**: $8 > 4$. No swap.
- **j=3 (7)**: $7 > 4$. No swap.
- **j=4 (1)**: $1 \leq 4$. $i \to 2$. Swap $A[2](8) \leftrightarrow A[4](1)$. Array: $[2, 1, 7, 8, 3, \dots]$.
- **j=5 (3)**: $3 \leq 4$. $i \to 3$. Swap $A[3](7) \leftrightarrow A[5](3)$. Array: $[2, 1, 3, 8, 7, \dots]$.
- **End**: Swap pivot $A[8](4)$ with $A[i + 1](A[4] = 8)$.
- **Result**: $[2, 1, 3, \mathbf{4}, 7, 5, 6, 8]$.

Observe: Left of 4 are $\{2, 1, 3\}$ (all ≤ 4). Right of 4 are $\{7, 5, 6, 8\}$ (all > 4).

5.2.3 Loop Invariant Visualization

At the start of each iteration j:

1. $A[p \dots i] \leq x$ (Known Small)
2. $A[i + 1 \dots j - 1] > x$ (Known Large)
3. $A[j \dots r - 1]$ (Unknown)

4. $A[r] = x$ (Pivot)

5.3 Rigorous Runtime Analysis

5.3.1 Worst-Case: The Unbalanced Split

The worst case occurs when the partition routine produces one subproblem with $n - 1$ elements and one with 0 elements.

- **Scenario**: Input is already sorted $(1, 2, \ldots, n)$ or reverse sorted. Pivot (last element) is always an extreme value.
- **Recurrence**: $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + cn$.
- **Summation**: $T(n) = \sum_{k=1}^{n} ck = c\frac{n(n+1)}{2} = \Theta(n^2)$.
- **Visual**: The recursion tree is a "linked list" of height n.

5.3.2 Best-Case: The Perfect Split

- **Scenario**: Pivot is always the median.
- **Recurrence**: $T(n) = 2T(n/2) + cn$.
- **Solution**: $\Theta(n \lg n)$ (Master Theorem Case 2).

5.3.3 Average-Case: The Intuition of Balance

Even a 9-to-1 split yields $O(n \lg n)$.

$$T(n) = T(n/10) + T(9n/10) + cn$$

The tree height is $\log_{10/9} n \approx \lg n$. Since the work at each level is $\leq cn$, the total is $O(n \lg n)$.

5.3.4 Average-Case: Mathematical Proof

Assume all permutations are equally likely. The pivot rank is uniformly distributed in $[0, n-1]$.

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-k-1)) + \Theta(n)$$

By symmetry:

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

We suspect $T(n) \leq an \lg n$. Substituting this into the sum:

$$\sum_{k=0}^{n-1} k \lg k \leq \int_2^n x \ln x \, dx \approx \frac{n^2 \ln n}{2} - \frac{n^2}{4}$$

Substituting back into the expression for $T(n)$ confirms the $O(n \lg n)$ bound.

5.4 Improvements and Variants

5.4.1 Randomized Quicksort

Problem: A fixed pivot (e.g., $A[r]$) is vulnerable to specific input patterns (like sorted arrays). **Solution**: Pick a pivot index randomly from $[p, r]$. Swap A[random] with $A[r]$ before partitioning.

- **Effect**: The worst case is no longer determined by the input order, but by the random number generator (extremely unlikely).
- **Expected Time**: $\Theta(n \lg n)$ for *any* input.

5.4.2 Median-of-3 Partitioning

Approximation of the true median. Select the pivot as the median of:

$$\{A[p], A[(p+r)/2], A[r]\}$$

This reduces the probability of bad splits and guards against sorted inputs without full randomization overhead.

5.4.3 Handling Duplicates

Standard Quicksort can degrade to $O(n^2)$ if all elements are equal. **Improvement**: 3-Way Partitioning (Dutch National Flag problem). Split array into: $\{< x\}, \{= x\}, \{> x\}$. Recursion only continues on the $< x$ and $> x$ parts.

5.4.4 Dual-Pivot Quicksort

(Used in Java's `Arrays.sort` for primitives). Uses **two pivots** (p_1, p_2) to split the array into three segments:

$$\{< p_1\}, \{p_1 \leq \cdots \leq p_2\}, \{> p_2\}$$

This reduces the height of the recursion tree and performs fewer memory accesses on average.

6 Randomisation & Lower Bounds

6.1 Randomisation in Algorithm Design

6.1.1 The Motivation

In previous lectures, we saw that QuickSort is efficient on average but suffers from a $\Theta(n^2)$ worst-case scenario. This worst case is triggered by specific input patterns (e.g., sorted or reverse-sorted arrays) when using a deterministic pivot (like the first or last element).

The Problem: A deterministic algorithm is vulnerable. A malicious adversary can construct an input that always triggers the worst case.

The Solution: Introduce randomness. Instead of assuming the input is random (Average Case Analysis), we make the algorithm itself random (Randomized Algorithm).

- **Goal**: Ensure that the algorithm runs efficiently for *all* inputs.
- **Philosophy**: By picking pivots randomly, we shift the dependency. The runtime no longer depends on the input order but on the sequence of random choices. No specific input can force the worst-case behavior.

6.1.2 Randomised QuickSort

The only change from standard QuickSort is the pivot selection. We choose a pivot uniformly at random from the subarray $A[p \dots r]$.

Algorithm 7 Randomised Partition

```
1: procedure RANDOMISED-PARTITION(A, p, r)
2:     i = RANDOM(p, r)                    ▷ Pick i ∈ [p, r] uniformly
3:     Exchange A[r] with A[i]             ▷ Move random pivot to the end
4:     return PARTITION(A, p, r)           ▷ Call standard partition
5: end procedure
```

Algorithm 8 Randomised QuickSort

```
1: procedure RANDOMISED-QUICKSORT(A, p, r)
2:     if p < r then
3:         q = RANDOMISED-PARTITION(A, p, r)
4:         RANDOMISED-QUICKSORT(A, p, q − 1)
5:         RANDOMISED-QUICKSORT(A, q + 1, r)
6:     end if
7: end procedure
```

6.2 Analysis of Expected Runtime

For a randomized algorithm, we analyze the **Expected Running Time** $E[T(n)]$. We count the number of comparisons, as this dominates the runtime.

6.2.1 Setup: Indicator Random Variables

Let X be the total number of comparisons. Let the sorted elements of the array be $z_1 < z_2 < \cdots < z_n$. We define an indicator random variable X_{ij} for any pair $i < j$:

$$X_{ij} = \mathbb{I}\{z_i \text{ is compared to } z_j\} = \begin{cases} 1 & \text{if } z_i \text{ and } z_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

The total number of comparisons is the sum of all pairs:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

6.2.2 Linearity of Expectation

A powerful property of expectation is that the expectation of a sum is the sum of expectations, *even if the variables are dependent.*

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

Since $E[X_{ij}] = \Pr(z_i \text{ is compared to } z_j)$, we just need to find this probability.

6.2.3 The Probability of Comparison

Critical Insight: When are two elements z_i and z_j compared?

- Elements are compared **only** when one of them is chosen as a pivot.
- Once a pivot x is chosen, all other elements are compared to x and then separated into different sets ($< x$ and $> x$).
- Consider the set of elements $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$.
- If any element $x \in Z_{ij}$ such that $z_i < x < z_j$ is chosen as a pivot *before* z_i or z_j, then z_i and z_j will be separated into different subproblems and **never compared**.
- Thus, z_i and z_j are compared if and only if the *first* element selected as a pivot from the set Z_{ij} is either z_i or z_j.

The set Z_{ij} has $j - i + 1$ elements. Since pivots are chosen uniformly at random, every element in Z_{ij} has an equal chance $(1/(j - i + 1))$ of being the first one picked. There are 2 favorable outcomes (z_i or z_j).

$$\Pr(z_i \text{ is compared to } z_j) = \frac{2}{j - i + 1}$$

6.2.4 Summation and Result

Substituting this back into the expectation sum:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

Let $k = j - i$ (the distance between elements). As j goes from $i + 1$ to n, k goes from 1 to $n - i$.

$$E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

The inner sum is the Harmonic series $H_n \approx \ln n$.

$$E[X] < \sum_{i=1}^{n-1} 2 \ln n = 2(n-1) \ln n = O(n \log n)$$

Conclusion: The expected runtime of Randomized QuickSort is $O(n \log n)$.

6.3 Lower Bounds for Comparison Sorts

We have seen several algorithms (MergeSort, HeapSort) with $O(n \log n)$ worst-case time. Can we do better? Can we sort in $O(n)$?

6.3.1 The Comparison Model

We restrict our analysis to **Comparison Sorts**: algorithms that only gain information about the input sequence by comparing pairs of elements ($a_i < a_j$, etc.). We cannot inspect the values themselves or use them as array indices.

6.3.2 Decision Trees

Any comparison sort can be modeled abstractly as a **Decision Tree**.

- **Root**: The first comparison made by the algorithm.
- **Internal Nodes**: Comparisons $a_i : a_j$.
- **Edges**: The result of the comparison (Left: $\leq$, Right: $>$).
- **Leaves**: A permutation of the input elements (the final sorted order).

An execution of the algorithm corresponds to a path from the root to a leaf. The length of the path is the number of comparisons. The **height** of the tree is the worst-case number of comparisons.

6.3.3 The Lower Bound Theorem

THEOREM 6.1. *Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.*

Proof. Let the decision tree have height h and L leaves.

1. **Necessary Leaves**: There are $n!$ possible permutations of n distinct elements. For the algorithm to be correct, it must be able to produce *any* of these $n!$ permutations

as output. Thus, the tree must have at least $n!$ reachable leaves:

$$L \geq n!$$

2. **Binary Tree Property**: A binary tree of height h has at most 2^h leaves.

$$L \leq 2^h$$

3. **Combining**:

$$2^h \geq n!$$

Taking logarithms base 2:

$$h \geq \lg(n!)$$

4. **Stirling's Approximation**: For large n, $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$.

$$\lg(n!) \approx \lg\left(\left(\frac{n}{e}\right)^n\right) = n\lg n - n\lg e = \Theta(n \log n)$$

5. **Conclusion**: $h = \Omega(n \log n)$.

□

6.4 Summary of Sorting Algorithms

We can now classify sorting algorithms based on this lower bound.

Algorithm	Best Time	Avg Time	Worst Time	Space
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Heap Sort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
Quick Sort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(\lg n)$
Rand. Quick Sort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(\lg n)$

Table 3: Comparison of Sorting Algorithms. Note: Heap Sort and Merge Sort are asymptotically optimal comparison sorts.

Note on Linear Time: The $\Omega(n \log n)$ bound applies only to *comparison* sorts. If we know more about the input (e.g., integers in a small range), we can break this barrier

using algorithms like Counting Sort or Radix Sort (next section).

7 Sorting in Linear Time

7.1 Breaking the Speed Limit

7.1.1 The Comparison Bottleneck

In previous lectures, we established a fundamental limit: **Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time in the worst case.** This is a mathematical certainty derived from the height of the decision tree.

7.1.2 The Way Out

How can we sort faster than $O(n \log n)$? We must change the rules of the game.

- **Stop Comparing**: Do not ask "Is $A[i] < A[j]$?".
- **Start Inspecting**: Use the actual numerical value of the keys to determine their position directly.
- **Constraint**: These algorithms assume the input comes from a restricted domain (e.g., small integers).

7.2 Counting Sort

Counting Sort is the foundational algorithm for linear-time sorting. It relies on the assumption that input elements are integers in the range $\{0, \ldots, k\}$.

7.2.1 The Algorithm logic

The core idea is determining, for each input element x, exactly how many elements are less than or equal to x. If there are 17 elements smaller than x, then x belongs at position 18.

Algorithm 9 Counting Sort

```
1: procedure Counting-Sort(A, B, k)
2:     Let C[0...k] be a new array initialized to 0
3:                                          ▷ Step 1: Frequency Count (Histogram)
4:     for j = 1 to A.length do
5:         C[A[j]] = C[A[j]] + 1
6:     end for
7:                                          ▷ Step 2: Cumulative Sums (Prefix Sums)
8:     for i = 1 to k do
9:         C[i] = C[i] + C[i − 1]
10:    end for
11:                                         ▷ C[i] now contains the number of elements ≤ i
12:                                         ▷ Step 3: Placement (Reverse Traversal)
13:    for j = A.length downto 1 do
14:        B[C[A[j]]] = A[j]                ▷ Place element at its correct sorted position
15:        C[A[j]] = C[A[j]] − 1            ▷ Decrement count for the next instance
16:    end for
17: end procedure
```

7.2.2 Why Traverse Backwards? (Stability)

Notice the loop in Step 3 goes **downto** 1. This is crucial for **Stability**.

- Suppose the input A contains two copies of value 3: 3_a at index 2 and 3_b at index 5.
- The algorithm encounters 3_b first. It places 3_b at position $C[3]$ and decrements $C[3]$.
- Later, it encounters 3_a. It places 3_a at position $C[3]$ (which is now one less).
- Result: 3_b appears after 3_a in the output, preserving their original relative order.

7.2.3 Complexity Analysis

- **Time**: $\Theta(k)$ (init) + $\Theta(n)$ (count) + $\Theta(k)$ (sum) + $\Theta(n)$ (place) = $\Theta(n+k)$.
- **Space**: $\Theta(n+k)$ for arrays B and C.
- **Implication**: If $k = O(n)$ (the range is linear in the number of elements), Counting Sort runs in **linear time** $\Theta(n)$.

7.3 Radix Sort

Counting Sort is fast but impractical for large ranges (e.g., sorting 32-bit integers would require $k = 2^{32}$ memory). Radix Sort solves this by sorting digit-by-digit.

7.3.1 The Algorithm (LSD Approach)

We sort by the Least Significant Digit (LSD) first, then the next digit, and so on, up to the Most Significant Digit (MSD). **Crucial Requirement**: The sort used for each digit must be **stable**.

Algorithm 10 Radix Sort

```
1: procedure RADIX-SORT(A, d)
2:     for i = 1 to d do
3:         Use a stable sort to sort array A on digit i
4:     end for
5: end procedure
```

7.3.2 Why LSD works (Intuition)

Why start from the right?

- After sorting on digit 1, the array is sorted by the last digit.
- When we sort on digit 2, stability ensures that if two numbers have the same 2nd digit, their order determined by the 1st digit is preserved.
- **Inductive Hypothesis**: After sorting on digit i, the array is sorted according to the value of the number formed by the last i digits.

7.3.3 Trace Example

Sorting: $[329, 457, 657, 839, 436, 720, 355]$

1. **Sort on ones digit**: $72\mathbf{0}, 35\mathbf{5}, 43\mathbf{6}, 45\mathbf{7}, 65\mathbf{7}, 32\mathbf{9}, 83\mathbf{9}$ (Note: 457 comes before 657 because it appeared first; 329 before 839).
2. **Sort on tens digit**: $7\mathbf{2}0, 3\mathbf{2}9, 4\mathbf{3}6, 8\mathbf{3}9, 3\mathbf{5}5, 4\mathbf{5}7, 6\mathbf{5}7$
3. **Sort on hundreds digit**: $\mathbf{3}29, \mathbf{3}55, \mathbf{4}36, \mathbf{4}57, \mathbf{6}57, \mathbf{7}20, \mathbf{8}39$ Sorted!

7.4 Advanced Analysis: Breaking the $O(n^3)$ Range

Consider the problem: Sort n integers in the range 0 to $n^3 - 1$.

7.4.1 Attempt 1: Comparison Sort

Standard Merge/Quick Sort takes $\Theta(n \log n)$. Good, but not linear.

7.4.2 Attempt 2: Counting Sort

The range is $k = n^3$. Runtime: $\Theta(n + k) = \Theta(n + n^3) = \Theta(n^3)$. This is terrible.

7.4.3 Attempt 3: Radix Sort (Base 10)

Range $M = n^3$. Number of digits $d = \log_{10} M = 3 \log_{10} n$. Runtime: $\Theta(d \cdot n) = \Theta(n \log n)$. Still not linear.

7.4.4 Attempt 4: Radix Sort (Base n)

Treat the numbers as being written in **base** n.

- The "digits" now range from 0 to $n - 1$. So $k = n$.
- How many digits d?

$$d = \log_n(n^3 - 1) \approx \log_n(n^3) = 3$$

  So we only have 3 "digits".

- **Runtime**:

$$T(n) = \Theta(d(n + k)) = \Theta(3(n + n)) = \Theta(6n) = \Theta(n)$$

Conclusion: By changing the base to n, we can sort integers up to n^k (for any constant k) in linear time.

7.5 Summary Table

Algorithm	Time	Space	Constraints
Counting Sort	$\Theta(n + k)$	$\Theta(n + k)$	Efficient when $k = O(n)$.
Radix Sort	$\Theta(d(n + k))$	$\Theta(n + k)$	Efficient when numbers have fixed d.
Comparison Sorts	$\Omega(n \log n)$	Varies	General purpose.

Table 4: Comparison of Linear-Time vs General Sorts

8 Elementary Data Structures

8.1 Foundations: The Limitations of Arrays

8.1.1 Data Structures vs. Raw Memory

A **Data Structure** is not merely a container; it is a strategic way of organizing a **finite dynamic set** of elements to optimize specific operations. Elements usually consist of:

- **Key**: The identifier (used for sorting/searching).
- **Satellite Data**: The payload.
- **Attributes**: Meta-data maintained by the structure (e.g., `next`, `top`, `head`).

8.1.2 The Array Bottleneck

Why do we need anything other than arrays? Arrays map indices to memory addresses directly ($O(1)$ access), but they are **rigid**.

- **Unsorted Array**:
  - **Insertion**: $O(1)$ (append to end).
  - **Search**: $\Theta(n)$ (linear scan).
- **Sorted Array**:
  - **Search**: $O(\log n)$ (Binary Search).
  - **Insertion/Deletion**: $\Theta(n)$. **Why?** Because elements are stored contiguously in memory. Inserting x at index i requires shifting elements $i \dots n$ one position to the right.

Conclusion: Arrays cannot simultaneously offer fast search **and** fast dynamic updates.

8.2 Stacks (LIFO)

8.2.1 Philosophy

A **Stack** enforces a **Last-In, First-Out (LIFO)** policy. By restricting access **strictly** to the "top" element, we eliminate the need for shifting. Thus, operations become $O(1)$.

8.2.2 Array Implementation

We can simulate a stack using an array S and a pointer $S.top$.

- **Empty**: $S.top = 0$.
- **Push**: Increment *top*, then write. (Check for Overflow).
- **Pop**: Read, then decrement *top*. (Check for Underflow).

Algorithm 11 Stack Operations

```
1: procedure PUSH(S, x)
2:     if S.top == S.size then
3:         Error "Overflow"
4:     else
5:         S.top = S.top + 1
6:         S[S.top] = x
7:     end if
8: end procedure

9: procedure POP(S)
10:    if STACK-EMPTY(S) then
11:        Error "Underflow"
12:    else
13:        S.top = S.top − 1
14:        Return S[S.top + 1]
15:    end if
16: end procedure
```

8.2.3 Algorithmic Applications

Stacks are the natural data structure for processing **nested** or **recursive** structures.

1. **Bracket Balance Checking**: Problem: Is '[()][()]' valid?

- **Algorithm**: Iterate through the string.

- If **Opening** (‘(‘, ‘[‘, ‘{‘): **Push** onto stack.
- If **Closing** (‘)‘, ‘]‘, ‘}‘): **Pop**. Check if the popped element matches the current closing bracket.
- **Result**: Valid iff stack is empty at the end and no mismatches occurred.

2. Postfix Expression Evaluation: Infix: $(9+3)*(4*2) \rightarrow$ Postfix: 9 3 + 4 2 * *

- **Algorithm**:
  - Operand $(9, 3)$: **Push**.
  - Operator $(+)$: **Pop** b (3), **Pop** a (9). Compute $a+b$. **Push** result (12).
- **Advantage**: Eliminates need for parentheses and operator precedence rules.

8.3 Queues (FIFO)

8.3.1 Philosophy

A **Queue** enforces a **First-In, First-Out (FIFO)** policy. It models fairness (lines, buffers).

- **Enqueue**: Add to **Tail**.
- **Dequeue**: Remove from **Head**.

8.3.2 Circular Buffer Implementation

To implement a queue in a fixed array without shifting elements after every dequeue, we use **modular arithmetic** to wrap indices around.

- $Q.head$: Index of the element to dequeue.
- $Q.tail$: Index of the **next empty slot**.
- **Full Condition**: $(Q.tail + 1) == Q.head$ (conceptually).

Runtime: $O(1)$ for both.

Algorithm 12 Queue Operations

```
1: procedure Enqueue(Q, x)
2:     Q[Q.tail] = x
3:     if Q.tail == Q.size then Q.tail = 1                ▷ Wrap around
4:     else Q.tail = Q.tail + 1
5:     end if
6: end procedure

7: procedure Dequeue(Q)
8:     x = Q[Q.head]
9:     if Q.head == Q.size then Q.head = 1                ▷ Wrap around
10:    else Q.head = Q.head + 1
11:    end if
12:    Return x
13: end procedure
```

8.4 Priority Queues

Stacks and Queues are determined by *time*. **Priority Queues** are determined by *importance* (Key).

- **Operations**: `Insert`, `Extract-Max`.
- **Implementation**:
  - Sorted Array: Extract-Max is $O(1)$, but Insert is $O(n)$.
  - **Heap**: Both operations are $O(\log n)$. (Covered in detail in Heapsort lecture).

8.5 Linked Lists

8.5.1 Breaking the Contiguity Constraint

Linked Lists decouple logical order from physical memory order.

- **Pros**: Dynamic size; $O(1)$ insertion/deletion (if location is known).
- **Cons**: No random access; $\Theta(n)$ search; Extra memory for pointers.

8.5.2 Operations Analysis

1. Searching: Must scan sequentially from `head`.

$$T(n) = \Theta(n)$$

2. Insertion (at front):

- New node points to old head.
- Old head's `prev` points to new node.
- Update `head` pointer.

$$T(n) = O(1)$$

3. Deletion: This operation highlights the difference between "Knowing the Key" and "Knowing the Node".

- **Case A: Given Key** k: We must first `Search` for k ($\Theta(n)$), then delete. Total: $\Theta(n)$.
- **Case B: Given Pointer** x: We simply rewire the pointers of $x.prev$ and $x.next$ to bypass x.

$$x.prev.next = x.next$$

$$x.next.prev = x.prev$$

Total: $O(1)$.

8.6 Summary of Complexities

Data Structure	Insert	Delete	Search	Constraint
Array (Unsorted)	$O(1)$	$O(1)^*$	$\Theta(n)$	Static Size
Array (Sorted)	$\Theta(n)$	$\Theta(n)$	$O(\log n)$	Static Size
Stack	$O(1)$	$O(1)$	-	LIFO Only
Queue	$O(1)$	$O(1)$	-	FIFO Only
Linked List	$O(1)$	$O(1)^{**}$	$\Theta(n)$	Sequential Access

Table 5: *Delete by swapping with last element. **Delete given pointer to node.

9 Binary Search Trees

9.1 Introduction to Trees

9.1.1 Definitions and Terminology

DEFINITION 9.1 (Binary Tree)**.** A **binary tree** is a structure defined recursively. It is a finite set of nodes that is either:

- Empty (contains no nodes).
- Consists of a **root** node and two disjoint binary trees called the **left subtree** and the **right subtree**.

Key terminology:

- **Leaf**: A node with no children (left and right subtrees are empty).
- **Height of a node**: The length (number of edges) of the longest simple path from the node to a leaf.
- **Height of a tree**: The height of the root.
- **Depth of a node**: The length of the path from the root to the node.
- **Full Binary Tree**: A binary tree in which every node is either a leaf or has exactly two children.

9.1.2 Inductive Proofs on Trees

Because trees are defined recursively, they are well-suited for proofs by induction.

THEOREM 9.1. *A binary tree of height h has at most 2^h leaves.*

Proof. We proceed by induction on the height h.

- **Base Case ($h = 0$)**: The tree consists of a single root node. It has 1 leaf. Since $2^0 = 1$, the base case holds.
- **Inductive Hypothesis**: Assume that for any binary tree of height $h - 1$, the number of leaves is at most 2^{h-1}.

- **Inductive Step**: Consider a tree T of height h. Its root has two subtrees (left and right), each with height at most $h-1$. Let $L(T)$ be the number of leaves. $L(T) = L(T_{left}) + L(T_{right})$. By the hypothesis, $L(T_{left}) \leq 2^{h-1}$ and $L(T_{right}) \leq 2^{h-1}$. Thus, $L(T) \leq 2^{h-1} + 2^{h-1} = 2 \cdot 2^{h-1} = 2^h$.

□

9.2 Binary Search Trees (BST)

DEFINITION 9.2 (Binary Search Tree Property)**.** Let x be a node in a binary search tree.

- If y is a node in the **left** subtree of x, then $y.key \leq x.key$.
- If y is a node in the **right** subtree of x, then $y.key \geq x.key$.

This property allows us to print all keys in sorted order using a simple recursive algorithm.

9.2.1 Tree Walks

Algorithm 13 Inorder Tree Walk

```
1: procedure INORDER-WALK(x)
2:     if x ≠ NIL then
3:         INORDER-WALK(x.left)
4:         Print x.key
5:         INORDER-WALK(x.right)
6:     end if
7: end procedure
```

THEOREM 9.2. *If x is the root of an n-node subtree, the call* `Inorder-Walk(x)` *takes $\Theta(n)$ time.*

Proof. We can prove this using the **Accounting Method**. Assign a constant cost c to each node. The procedure visits each node exactly once (when printing), and traverses each edge exactly twice (once going down, once returning up). Since the number of edges is $n-1$, the total work is proportional to the number of nodes. Thus, $T(n) = \Theta(n)$. □

9.3 Query Operations

Operations on a BST generally depend on the height h of the tree, taking $O(h)$ time.

9.3.1 Search

To find a node with key k, we trace a path from the root.

Algorithm 14 Tree Search

```
1: procedure TREE-SEARCH(x, k)
2:     if x == NIL or k == x.key then
3:         return x
4:     end if
5:     if k < x.key then
6:         return TREE-SEARCH(x.left, k)
7:     else
8:         return TREE-SEARCH(x.right, k)
9:     end if
10: end procedure
```

9.3.2 Minimum and Maximum

Due to the BST property:

- The **minimum** element is found by following `left` pointers until a node with no left child is reached.
- The **maximum** element is found by following `right` pointers until a node with no right child is reached.

Runtime: $O(h)$.

9.3.3 Successor

The **successor** of a node x is the node with the smallest key greater than $x.key$.

- **Case 1**: If the right subtree of x is non-empty, the successor is the **minimum** node in the right subtree.
- **Case 2**: If the right subtree is empty, the successor is the lowest ancestor of x whose left child is also an ancestor of x (i.e., we go up until we turn right).

Algorithm 15 Tree Successor

```
1: procedure TREE-SUCCESSOR(x)
2:     if x.right ≠ NIL then
3:         return TREE-MINIMUM(x.right)
4:     end if
5:     y = x.p
6:     while y ≠ NIL and x == y.right do        ▷ Go up while we are a right child
7:         x = y
8:         y = y.p
9:     end while
10:    return y
11: end procedure
```

9.4 Modifying Operations

9.4.1 Insertion

To insert a new value z, we trace a path from the root downwards, similar to `Tree-Search`. We maintain a "trailing pointer" y to keep track of the parent. When we hit `NIL`, we attach z to y.

Algorithm 16 Tree Insert

```
1: procedure TREE-INSERT(T, z)
2:     y = NIL
3:     x = T.root
4:     while x ≠ NIL do
5:         y = x
6:         if z.key < x.key then x = x.left
7:         else x = x.right
8:         end if
9:     end while
10:    z.p = y
11:    if y == NIL then T.root = z              ▷ Tree was empty
12:    else if z.key < y.key then y.left = z
13:    else y.right = z
14:    end if
15: end procedure
```

Runtime: $O(h)$.

9.4.2 Deletion

Deleting a node z is the most complex operation. We handle three cases:

1. **No children**: z is a leaf. Simply remove it.
2. **One child**: Splice z out. Replace z with its child.
3. **Two children**: Find z's successor y. y must lie in z's right subtree and has no left child. We replace z's content with y's content and splice y out.

We use a subroutine `Transplant` to replace one subtree rooted at u with another rooted at v.

Algorithm 17 Transplant

```
1: procedure TRANSPLANT(T, u, v)
2:     if u.p == NIL then
3:         T.root = v
4:     else if u == u.p.left then
5:         u.p.left = v
6:     else
7:         u.p.right = v
8:     end if
9:     if v ≠ NIL then
10:        v.p = u.p
11:    end if
12: end procedure
```

Algorithm 18 Tree Delete

```
1: procedure TREE-DELETE(T, z)
2:     if z.left == NIL then                    ▷ Case 1 or Case 2 (no left child)
3:         TRANSPLANT(T, z, z.right)
4:     else if z.right == NIL then              ▷ Case 2 (no right child)
5:         TRANSPLANT(T, z, z.left)
6:     else                                     ▷ Case 3: Two children
7:         y = TREE-MINIMUM(z.right)            ▷ Find successor
8:         if y.p ≠ z then                      ▷ If successor is not immediate child
9:             TRANSPLANT(T, y, y.right)        ▷ Replace y with its right child
10:            y.right = z.right
11:            y.right.p = y
12:        end if
13:        TRANSPLANT(T, z, y)                  ▷ Replace z with y
14:        y.left = z.left
15:        y.left.p = y
16:    end if
17: end procedure
```

Runtime: $O(h)$ because finding the successor and handling pointers takes time proportional to height.

9.5 Performance Analysis

The runtime of BST operations is determined by the height h.

- **Worst Case**: If we insert elements in sorted (or reverse sorted) order, the tree becomes a linear chain.

$$h = n \implies \text{Runtime} = \Theta(n)$$

- **Best/Average Case**: In a randomly built BST, the expected height is logarithmic.

$$h = O(\lg n) \implies \text{Runtime} = O(\lg n)$$

This instability motivates the need for **Balanced Binary Search Trees** (like AVL trees or Red-Black trees), which guarantee $h = O(\lg n)$.

10 AVL Trees

10.1 Motivation

Standard Binary Search Trees (BSTs) support operations (Search, Insert, Delete, Min, Max) in $O(h)$ time. However, in the worst case (e.g., inserting sorted data), the height h can degrade to $O(n)$, making operations linear. To guarantee $O(\lg n)$ performance, we need to enforce a balance condition. **AVL Trees** (Adelson-Velsky and Landis, 1962) are the first self-balancing BSTs invented.

10.2 Definition and Properties

10.2.1 AVL Property

DEFINITION 10.1 (AVL Tree)**.** An **AVL tree** is a binary search tree that satisfies the **AVL Property**: For every node v in the tree, the heights of its left and right subtrees differ by at most 1.

DEFINITION 10.2 (Balance Factor)**.** The **balance factor** of a node v, denoted $bal(v)$, is defined as the height of the left subtree minus the height of the right subtree:

$$bal(v) = height(v.left) - height(v.right)$$

In an AVL tree, for every node v, $bal(v) \in \{-1, 0, 1\}$.

10.2.2 Height Analysis

THEOREM 10.1. *The height h of an AVL tree with n nodes is $O(\lg n)$. More specifically, $h < 1.44 \lg(n + 2)$.*

Proof. Let $N(h)$ be the minimum number of nodes in an AVL tree of height h.

- $N(0) = 1$ (Root only, height 0).
- $N(1) = 2$ (Root + 1 child).

- For general h, the root must have two subtrees. To minimize nodes while maintaining height h, one subtree must have height $h-1$ and the other $h-2$.

$$N(h) = 1 + N(h-1) + N(h-2)$$

This recurrence is strictly greater than the Fibonacci sequence ($F_h \approx \phi^h/\sqrt{5}$ where $\phi \approx 1.618$). Solving the recurrence implies $n > \phi^h$, thus $h < \log_\phi n \approx 1.44 \lg n$. □

10.3 Rotations

Rotations are the fundamental operations to rebalance a tree without violating the BST property (inorder traversal remains unchanged). They take $O(1)$ time.

10.3.1 Right Rotation

Used when the left subtree is too heavy.

Algorithm 19 Right Rotation

```
1: procedure RIGHT-ROTATE(T, y)
2:     x = y.left
3:     y.left = x.right                ▷ Turn x's right subtree into y's left subtree
4:     if x.right ≠ NIL then x.right.p = y
5:     end if
6:     x.p = y.p                       ▷ Link x to y's parent
7:     if y.p == NIL then T.root = x
8:     else if y == y.p.right then y.p.right = x
9:     else y.p.left = x
10:    end if
11:    x.right = y                     ▷ Put y on x's right
12:    y.p = x
13:    Update heights of x and y
14: end procedure
```

10.3.2 Left Rotation

Used when the right subtree is too heavy. Symmetric to Right Rotation.

10.4 Insertion

Goal: Insert a node z and restore the AVL property if violated. **Steps**:

1. Perform a standard BST insertion.
2. Update the height of ancestors of z.
3. Check for imbalance: If a node v has $|bal(v)| > 1$, perform rotations.

Let v be the lowest unbalanced node (the first one encountered moving up from z). Let x be the child of v in the direction of the imbalance. There are 4 cases:

10.4.1 Case 1: Left-Left (LL)

- **Condition**: v is heavy on the left ($bal(v) = +2$) AND x is heavy on the left ($bal(x) = +1$ or 0).
- **Fix**: **Right-Rotate**(v).

10.4.2 Case 2: Right-Right (RR)

- **Condition**: v is heavy on the right ($bal(v) = -2$) AND x is heavy on the right ($bal(x) = -1$ or 0).
- **Fix**: **Left-Rotate**(v).

10.4.3 Case 3: Left-Right (LR)

- **Condition**: v is heavy on the left ($bal(v) = +2$) BUT x is heavy on the right ($bal(x) = -1$).
- **Fix**: Double Rotation.
  1. **Left-Rotate**(x) (Converts structure to LL case).
  2. **Right-Rotate**(v).

10.4.4 Case 4: Right-Left (RL)

- **Condition**: v is heavy on the right ($bal(v) = -2$) BUT x is heavy on the left ($bal(x) = +1$).
- **Fix**: Double Rotation.
  1. **Right-Rotate(x)** (Converts structure to RR case).
  2. **Left-Rotate(v)**.

Runtime: Insertion takes $O(\lg n)$ for the search. Rebalancing requires retracing the path ($O(\lg n)$) and at most **one** or **two** rotations (since fixing the lowest imbalance restores the global height). Total: $O(\lg n)$.

10.5 Deletion

Goal: Delete node z and restore AVL property. **Steps**:

1. Perform standard BST deletion.
2. Retrace the path from the parent of the deleted node (or spliced node) up to the root.
3. At each node, update height and check balance.
4. If unbalanced, apply rotations (same 4 cases as insertion).

10.5.1 Propagation of Imbalance

Unlike insertion, where one rotation fixes the tree, a rotation during deletion might reduce the height of the subtree, causing the *parent* to become unbalanced. Thus, rebalancing may propagate all the way up to the root, requiring $O(\lg n)$ rotations.

Runtime: $O(\lg n)$.

10.6 Summary

Operation	**BST (Worst Case)**	**AVL (Worst Case)**
Search	$O(n)$	$O(\lg n)$
Insert	$O(n)$	$O(\lg n)$
Delete	$O(n)$	$O(\lg n)$

Table 6: Comparison of BST and AVL complexities

11 Dynamic Programming

11.1 The Paradigm Shift

11.1.1 Motivation: Divide-and-Conquer vs. Dynamic Programming

Divide-and-Conquer algorithms (like MergeSort or QuickSort) rely on splitting a problem into *disjoint* subproblems.

- **Disjoint**: The subproblems do not share any common history. Solving the left half of an array tells you nothing about the right half.
- **Efficiency**: Because they are disjoint, we just solve each once and combine them.

However, many optimization problems break down into **Overlapping Subproblems**.

- If we use standard recursion, we end up solving the same small subproblems millions of times.
- **Dynamic Programming (DP)** is the technique of "Store, Don't Recompute". We trade space (memory to store results) for time (CPU cycles).

11.2 Motivating Example: Fibonacci Numbers

The Fibonacci sequence is defined as:

$$F_0 = 0, F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

11.2.1 The Naive Recursive Approach

Algorithm 20 Naive-Fib(n)

```
1: procedure FIB(n)
2:     if n ≤ 1 then return n
3:     else return FIB(n − 1) + FIB(n − 2)
4:     end if
5: end procedure
```

Analysis: The recursion tree grows exponentially. To compute $F(5)$, we compute $F(3)$ twice. To compute $F(6)$, we compute $F(4)$ twice and $F(3)$ three times.

$$T(n) \approx \phi^n \approx 1.618^n \quad \text{(Exponential Explosion)}$$

11.2.2 The DP Approach (Memoization)

We use a table to cache results. This is called **Memoization** (Top-Down strategy).

- Before computing, check the table.
- After computing, save to the table.

Algorithm 21 Memoized-Fib(n)

```
Let memo[0...n] be array initialized to ∞
procedure FIB-MEMO(n)
    if memo[n] ≠ ∞ then return memo[n]          ▷ Cache Hit
    end if
    if n ≤ 1 then f = n
    else f = FIB-MEMO(n − 1) + FIB-MEMO(n − 2)
    end if
    memo[n] = f                                 ▷ Cache Miss: Store result return f
end procedure
```

Analysis: Since each $F(i)$ for $0 \leq i \leq n$ is computed exactly once, the time complexity drops to:

$$T(n) = \Theta(n) \quad \text{(Linear Time)}$$

11.3 Case Study: Rod Cutting Problem

11.3.1 Problem Definition

We are given a steel rod of length n and a price table p_i for rod pieces of length i. We want to cut the rod into pieces to maximize total revenue r_n.

Mathematical Formulation: We can cut a piece of length i ($1 \leq i \leq n$) off the left end, and then optimally solve the problem for the remaining length $n - i$.

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

(Base case: $r_0 = 0$).

11.3.2 Approach 1: Top-Down with Memoization

We stick to the recursive logic but add a "memory" (array r).

- **Check**: Before computing, is $r[n] \geq 0$? If yes, return it.
- **Save**: After computing max q, save $r[n] = q$.

Algorithm 22 Memoized-Cut-Rod(p, n)

```
1: procedure MEMOIZED-CUT-ROD(p, n)
2:     Let r[0...n] be new array initialized to −∞
3:     return MEMOIZED-CUT-ROD-AUX(p, n, r)
4: end procedure
                                        ▷ Visual separator between procedures
5: procedure MEMOIZED-CUT-ROD-AUX(p, n, r)
6:     if r[n] ≥ 0 then
7:         return r[n]                                  ▷ Return cached value
8:     end if
9:     if n == 0 then
10:        q = 0
11:    else
12:        q = −∞
13:        for i = 1 to n do                    ▷ Try every possible first cut
14:            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
15:        end for
16:    end if
17:    r[n] = q                              ▷ Store optimal value for length n
18:    return q
19: end procedure
```

11.3.3 Approach 2: Bottom-Up (Tabulation)

Recursion has overhead (stack depth). We can eliminate recursion by filling the table in order of dependency: from smallest (0) to largest (n).

Runtime: The nested loop structure gives a clear arithmetic sum:

$$T(n) = \sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \Theta(n^2)$$

Algorithm 23 Bottom-Up-Cut-Rod(p, n)

```
1: procedure BOTTOM-UP-CUT-ROD(p, n)
2:     Let r[0...n] be new array
3:     r[0] = 0
4:     for j = 1 to n do                                   ▷ Solve for rod length j
5:         q = −∞
6:         for i = 1 to j do                               ▷ Try cut length i
7:                                  ▷ Use previously computed solution for remainder j-i
8:             q = max(q, p[i] + r[j − i])
9:         end for
10:        r[j] = q
11:    end forreturn r[n]
12: end procedure
```

11.4 Reconstructing the Solution

Merely knowing the max revenue (r_n) is often insufficient; we need the list of cut lengths. To do this, we store the choice that yielded the optimal value. We extend the algorithm to maintain an array $s[j]$, which stores the **optimal first cut length** for a rod of length j.

Algorithm 24 Extended-Bottom-Up-Cut-Rod(p, n)

```
1: procedure EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2:     Let r[0...n] and s[0...n] be new arrays
3:     r[0] = 0
4:     for j = 1 to n do
5:         q = −∞
6:         for i = 1 to j do
7:             if q < p[i] + r[j − i] then
8:                 q = p[i] + r[j − i]
9:                 s[j] = i                                ▷ Record the best cut i
10:            end if
11:        end for
12:        r[j] = q
13:    end forreturn r and s
14: end procedure
```

11.4.1 Printing the Cuts

We do not need recursion to print the solution. We simply trace the s array.

Algorithm 25 Print-Cut-Rod-Solution(p, n)

```
1: procedure Print-Cut-Rod-Solution(p, n)
2:     (r, s) = Extended-Bottom-Up-Cut-Rod(p, n)
3:     while n > 0 do
4:         Print s[n]                          ▷ Print first cut
5:         n = n − s[n]                        ▷ Move to remaining length
6:     end while
7: end procedure
```

11.5 Theoretical Foundations

When should we use DP? Two key properties must hold:

11.5.1 Optimal Substructure

A problem exhibits optimal substructure if **an optimal solution to the problem contains within it optimal solutions to subproblems**.

- **Example**: Shortest Path. If $A \to B \to C$ is the shortest path from A to C, then $A \to B$ must be the shortest path from A to B.
- **Non-Example**: Longest Simple Path. Longest path $q \to r \to t$ does NOT imply $q \to r$ is the longest simple path.

This property justifies the recurrence relation: $r_n = \max(p_i + r_{n-i})$.

11.5.2 Overlapping Subproblems

The recursive algorithm visits the **same** problem instances repeatedly.

- This distinguishes DP from Divide-and-Conquer.
- **Merge Sort**: Computes merge on disjoint arrays.
- **Rod Cutting**: To solve for length 4, we need length 3, 2, 1, 0. To solve for length 3, we need 2, 1, 0. Subproblem "length 2" is reused multiple times.

11.6 Summary Comparison

Feature	**Divide and Conquer**	**Dynamic Programming**
Subproblems	Independent (Disjoint)	Overlapping
Approach	Recursive	Recursive (Memoized) or Iterative (Tabulation)
Efficiency	Splits problem size (e.g., $n/2$)	Reuses solutions (Polynomial time)
Examples	Merge Sort, Quick Sort	Rod Cutting, Fibonacci, LCS

Table 7: Paradigm Comparison

12 Greedy Algorithms

12.1 The Philosophy of Greed

12.1.1 Core Concept

A **Greedy Algorithm** builds a solution piece by piece, always choosing the next piece that offers the most immediate benefit.

- **Motto**: "Take what you can get now!"
- **Strategy**: At each step, make a **locally optimal** choice in the hope that these choices will lead to a **globally optimal** solution.
- **Benefit**: Usually very efficient (often $O(n \log n)$ or $O(n)$).
- **Risk**: It doesn't always work. It never looks back to correct previous mistakes.

12.1.2 Comparison with Dynamic Programming

- **Dynamic Programming**: "I need to know the solution to all subproblems before I make a decision." (Bottom-Up or Recursion). It considers *all* possibilities.
- **Greedy**: "I will make a decision right now based on current information, and then I will solve the remaining subproblem." (Top-Down). It commits to a choice *before* solving the subproblem.

12.2 Case Study: Activity Selection Problem

12.2.1 The Problem

We have a set of activities $S = \{a_1, a_2, \ldots, a_n\}$. Each activity a_i has a start time s_i and a finish time f_i.

- **Constraint**: Two activities are compatible if their time intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- **Goal**: Select the **maximum number** of mutually compatible activities.

12.2.2 The Greedy Strategy

How do we choose the "best" activity to pick first?

1. *Shortest duration?* No, a short task might happen right in the middle, blocking two other tasks.
2. *Earliest start time?* No, a task starting at 1 AM might last until midnight, blocking everything else.
3. **Earliest Finish Time**: Yes!

Intuition: By picking the activity that finishes **soonest**, we leave the **maximum amount of remaining time** for other activities.

Algorithm 26 Greedy-Activity-Selector(s, f)

```
1: procedure GREEDY-ACTIVITY-SELECTOR(s, f)
2:     Sort activities by finish time: f_1 ≤ f_2 ≤ ··· ≤ f_n
3:     A = {a_1}                          ▷ Always select the first activity
4:     k = 1                              ▷ Index of the last selected activity
5:     for m = 2 to n do
6:         if s_m ≥ f_k then              ▷ If activity m starts after k finishes
7:             A = A ∪ {a_m}
8:             k = m
9:         end if
10:    end forreturn A
11: end procedure
```

12.2.3 Runtime Analysis

- Sorting: $O(n \log n)$.
- Linear Scan: $\Theta(n)$.
- **Total**: $\Theta(n \log n)$.

12.3 Theoretical Foundations

To prove a Greedy Algorithm works, the problem must exhibit two key properties:

12.3.1 Greedy Choice Property

We can assemble a globally optimal solution by making a locally optimal (greedy) choice.

- **Implication**: We don't need to consider all options (like DP). We just prove that the greedy choice is *always* part of *some* optimal solution.

12.3.2 Optimal Substructure

An optimal solution to the problem contains within it optimal solutions to subproblems.

- If we pick activity a_1, the problem reduces to finding optimal activities in the remaining time after a_1 finishes.

12.4 The Tale of Two Knapsacks

This is the classic example showing when Greedy works and when it fails.

12.4.1 Problem Definitions

A thief robbing a store finds n items. The i-th item has value v_i and weight w_i. The knapsack capacity is W.

- **0-1 Knapsack**: You must take the item whole or leave it ($x_i \in \{0, 1\}$). (e.g., Gold ingots).
- **Fractional Knapsack**: You can take any fraction of an item ($0 \leq x_i \leq 1$). (e.g., Gold dust).

12.4.2 The Greedy Strategy: Value Density

The intuitive greedy choice is to pick items with the highest **value per unit weight** (v_i/w_i).

12.4.3 Analysis: Why Greedy Fails for 0-1 Knapsack

Consider Capacity $W = 50$.

- Item A: Weight 10, Value \$60 (\$6/lb)
- Item B: Weight 20, Value \$100 (\$5/lb)
- Item C: Weight 30, Value \$120 (\$4/lb)

Greedy Choice: 1. Pick Item A (Highest density). Remaining Capacity: 40. 2. Next best is Item B. Remaining Capacity: 20. 3. Item C (Weight 30) doesn't fit. **Total Value**: \$160.

Optimal Solution: Skip Item A. Pick Item B and Item C. Total Weight: $20+30 = 50$. **Total Value**: $\$100 + \$120 = \$220$.

Conclusion: Greedy failed because picking A "polluted" the capacity, preventing us from taking the heavier but valuable combination of B+C. 0-1 Knapsack requires **Dynamic Programming**.

12.4.4 Analysis: Why Greedy Works for Fractional Knapsack

Using the same items: 1. Pick Item A (10 lbs). Value \$60. Rem Cap: 40. 2. Pick Item B (20 lbs). Value \$100. Rem Cap: 20. 3. Pick 2/3 of Item C (20 lbs). Value $(2/3) \times 120 = \$80$. **Total Value**: \$240.

Conclusion: Because we can fill the "gaps" with fractions, the "Value Density" strategy guarantees optimality.

12.5 5. Summary Comparison

Feature	**Dynamic Programming**	**Greedy Algorithms**
Choice Logic	Dependent on subproblems (Look ahead)	Independent, Local (Look now)
Structure	Bottom-Up or Top-Down	Top-Down
Efficiency	Slower (often $O(n^2)$ or pseudo-polynomial)	Very Fast (often $O(n \log n)$)
Correctness	Guarantees Optimal	Hard to prove (requires Greedy Proper
Example	0-1 Knapsack, LCS, Rod Cutting	Fractional Knapsack, Activity Selectio

Table 8: DP vs. Greedy

13 Elementary Graph Algorithms

13.1 Graph Representations

A graph $G = (V, E)$ can be represented in two standard ways: adjacency lists and adjacency matrices.

13.1.1 Adjacency Lists

An array Adj of $|V|$ lists, one for each vertex. The list $Adj[u]$ contains all vertices v such that $(u, v) \in E$.

- **Space Complexity**: $\Theta(V + E)$.
- **Pros**: Compact for sparse graphs.
- **Cons**: Checking if an edge (u, v) exists takes $O(deg(u))$ time.

13.1.2 Adjacency Matrix

A $|V| \times |V|$ matrix A where $A_{ij} = 1$ if $(i, j) \in E$, and 0 otherwise.

- **Space Complexity**: $\Theta(V^2)$.
- **Pros**: Checking edge existence takes $O(1)$.
- **Cons**: Wasteful for sparse graphs.

13.2 Breadth-First Search (BFS)

BFS explores a graph by visiting all neighbors of a node before moving to the next level of neighbors. It computes the shortest path distance (in terms of number of edges) from a source vertex s to all reachable vertices.

13.2.1 Algorithm

BFS uses a **queue** to manage the frontier of exploration. Vertices are colored to track their status:

- **White**: Undiscovered.
- **Gray**: Discovered but not fully processed (in the queue).
- **Black**: Fully processed (dequeued).

Algorithm 27 Breadth-First Search

```
1: procedure BFS(G, s)
2:     for each vertex u ∈ G.V − {s} do
3:         u.color = WHITE
4:         u.d = ∞
5:         u.π = NIL
6:     end for
7:     s.color = GRAY
8:     s.d = 0
9:     s.π = NIL
10:    Q = ∅
11:    ENQUEUE(Q, s)
12:    while Q ≠ ∅ do
13:        u = DEQUEUE(Q)
14:        for each v ∈ G.Adj[u] do
15:            if v.color == WHITE then
16:                v.color = GRAY
17:                v.d = u.d + 1
18:                v.π = u
19:                ENQUEUE(Q, v)
20:            end if
21:        end for
22:        u.color = BLACK
23:    end while
24: end procedure
```

13.2.2 Analysis

- **Time Complexity**:
  - Each vertex is enqueued and dequeued at most once: $O(V)$.
  - The adjacency list of each vertex is scanned exactly once: $\sum_{u \in V} |Adj[u]| = \Theta(E)$.

  Total runtime: $O(V + E)$.
- **Shortest Paths**: Upon termination, $v.d = \delta(s, v)$ (shortest path distance) for all reachable v.
- **BFS Tree**: The predecessor pointers π define a breadth-first tree rooted at s.

13.3 Depth-First Search (DFS)

DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all edges from v have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.

13.3.1 Algorithm

DFS uses recursion (implicit stack). It records two timestamps for each vertex:

- $v.d$: Discovery time (vertex becomes Gray).
- $v.f$: Finish time (vertex becomes Black).

Algorithm 28 Depth-First Search

```
1: procedure DFS(G)
2:     for each vertex u ∈ G.V do
3:         u.color = WHITE
4:         u.π = NIL
5:     end for
6:     time = 0
7:     for each vertex u ∈ G.V do
8:         if u.color == WHITE then
9:             DFS-Visit(G, u)
10:        end if
11:    end for
12: end procedure

13: procedure DFS-Visit(G, u)
14:    time = time + 1
15:    u.d = time
16:    u.color = GRAY
17:    for each v ∈ G.Adj[u] do
18:        if v.color == WHITE then
19:            v.π = u
20:            DFS-Visit(G, v)
21:        end if
22:    end for
23:    u.color = BLACK
24:    time = time + 1
25:    u.f = time
26: end procedure
```

13.3.2 Analysis

- **Time Complexity**:
  - Initialization loops: $\Theta(V)$.
  - `DFS-Visit` is called exactly once for each vertex.
  - The loop over $Adj[u]$ executes $|Adj[u]|$ times. Total cost for edge exploration is $\Theta(E)$.

  Total runtime: $\Theta(V + E)$.
- **Parenthesis Theorem**: For any two vertices u and v, exactly one of the following holds:
  - Intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint.
  - Interval $[u.d, u.f]$ is contained within $[v.d, v.f]$ (v is an ancestor of u).
  - Interval $[v.d, v.f]$ is contained within $[u.d, u.f]$ (u is an ancestor of v).

13.3.3 Edge Classification

DFS can classify edges (u, v) based on the color of v when the edge is explored:

1. **Tree Edge**: v is WHITE. (Edge in the DFS forest).
2. **Back Edge**: v is GRAY. (Edge to an ancestor). Indicates a cycle.
3. **Forward Edge**: v is BLACK and $u.d < v.d$. (Edge to a descendant).
4. **Cross Edge**: v is BLACK and $v.d < u.d$. (All other edges).

13.4 Applications of DFS

13.4.1 Topological Sort

A topological sort of a DAG (Directed Acyclic Graph) is a linear ordering of vertices such that for every edge (u, v), u appears before v.

- **Algorithm**: Run DFS. As each vertex is finished (blackened), insert it onto the front of a linked list.
- **Runtime**: $\Theta(V + E)$.

13.4.2 Strongly Connected Components (SCC)

A strongly connected component of a directed graph is a maximal set of vertices $C \subseteq V$ such that for every pair $u, v \in C$, u is reachable from v and v is reachable from u.

- **Algorithm (Kosaraju's)**:
  1. Call DFS on G to compute finish times $u.f$.
  2. Compute G^T (transpose of G).
  3. Call DFS on G^T, but in the main loop consider vertices in order of decreasing $u.f$.
  4. Output the vertices of each tree in the DFS forest of step 3 as a separate SCC.
- **Runtime**: $\Theta(V + E)$.

14 Depth First Search & Applications

14.1 Review of DFS

Depth-First Search (DFS) explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. It uses colors (White, Gray, Black) and timestamps ($v.d$ for discovery, $v.f$ for finish) to track progress.

Algorithm 29 Depth-First Search

```
1: procedure DFS(G)
2:     for each vertex u ∈ G.V do
3:         u.color = WHITE
4:         u.π = NIL
5:     end for
6:     time = 0
7:     for each vertex u ∈ G.V do
8:         if u.color == WHITE then
9:             DFS-Visit(G, u)
10:        end if
11:    end for
12: end procedure
```

Algorithm 30 DFS-Visit

```
1: procedure DFS-Visit(G, u)
2:     time = time + 1
3:     u.d = time
4:     u.color = GRAY
5:     for each v ∈ G.Adj[u] do
6:         if v.color == WHITE then
7:             v.π = u
8:             DFS-Visit(G, v)
9:         end if
10:    end for
11:    u.color = BLACK
12:    time = time + 1
13:    u.f = time
14: end procedure
```

Runtime: $\Theta(V + E)$.

14.2 Properties of DFS

14.2.1 Parenthesis Structure

For any two vertices u and v, exactly one of the following holds:

1. The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint (neither is a descendant of the other).
2. The interval $[u.d, u.f]$ is contained entirely within $[v.d, v.f]$ (u is a descendant of v).
3. The interval $[v.d, v.f]$ is contained entirely within $[u.d, u.f]$ (v is a descendant of u).

14.2.2 White-Path Theorem

THEOREM 14.1. *In a depth-first forest of a graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.*

14.2.3 Edge Classification

DFS classifies edges (u, v) based on the color of v when the edge is explored:

- **Tree Edge**: v is WHITE.
- **Back Edge**: v is GRAY. (Implies a cycle).
- **Forward Edge**: v is BLACK and $u.d < v.d$.
- **Cross Edge**: v is BLACK and $u.d > v.d$.

Note: In an undirected graph, every edge is either a tree edge or a back edge.

14.3 Applications

14.3.1 Cycle Detection

THEOREM 14.2. *A graph G contains a cycle if and only if a DFS yields at least one back edge.*

Proof. ($\Leftarrow$) If (u, v) is a back edge, then v is an ancestor of u in the DFS tree. Thus, there is a tree path from v to u, and the edge (u, v) completes the cycle. ($\Rightarrow$) If G has a cycle C, let v be the first vertex in C discovered by DFS. Let (u, v) be the preceding edge in C. At time $v.d$, the vertices of C form a white path from v to u. By the White-Path Theorem, u becomes a descendant of v. Thus, (u, v) is a back edge. □

14.3.2 Topological Sort

A topological sort of a DAG (Directed Acyclic Graph) is a linear ordering of vertices such that for every edge (u, v), u appears before v.

Algorithm 31 Topological Sort

```
1: procedure TOPOLOGICAL-SORT(G)
2:     Call DFS(G) to compute finishing times v.f for each vertex v
3:     As each vertex is finished, insert it onto the front of a linked list
4:     return the linked list of vertices
5: end procedure
```

Correctness: If there is an edge (u, v), then $v.f < u.f$.

- When (u, v) is explored, v cannot be GRAY (otherwise it's a back edge, implying a cycle).
- If v is WHITE, it becomes a descendant of u, so $v.f < u.f$.
- If v is BLACK, it is already finished, so $v.f < u.d < u.f$.

Runtime: $\Theta(V + E)$.

14.3.3 Strongly Connected Components (SCC)

An SCC is a maximal set of vertices $C \subseteq V$ such that for every pair $u, v \in C$, $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Runtime: $\Theta(V + E)$.

- DFS on G: $\Theta(V + E)$.
- Compute G^T: $\Theta(V + E)$.
- DFS on G^T: $\Theta(V + E)$.

Algorithm 32 SCC Algorithm (Kosaraju)

1: **procedure** Strongly-Connected-Components(G)
2: Call DFS(G) to compute finishing times $u.f$
3: Compute G^T (transpose of G)
4: Call DFS(G^T), but in the main loop consider vertices in order of decreasing $u.f$ (from step 1)
5: Output the vertices of each tree in the DFS forest of step 3 as a separate SCC
6: **end procedure**

Remark 14.1. This algorithm works because components in the component graph (which is a DAG) are effectively topologically sorted by finish times. Processing nodes in decreasing order of finish times in G^T isolates the SCCs one by one (sink components in the component graph of G become source components in G^T).

15 Minimum Spanning Trees and Shortest Paths

15.1 Minimum Spanning Trees (MST)

15.1.1 Problem Definition

Given a connected, undirected graph $G = (V, E)$ where each edge $(u, v) \in E$ has a weight $w(u, v) \geq 0$. We want to find a subset of edges $T \subseteq E$ that connects all vertices and minimizes the total weight:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

Since T connects all vertices and has minimum weight, it must be acyclic. Thus, T is a tree, called a **Spanning Tree**.

15.1.2 Generic Greedy Approach

We grow a set of edges A, maintaining the loop invariant that A is a subset of some minimum spanning tree. At each step, we add a "safe edge" (u, v) to A such that $A \cup \{(u, v)\}$ is still a subset of an MST.

DEFINITION 15.1 (Cut). A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V.

DEFINITION 15.2 (Cross). An edge $(u, v) \in E$ **crosses** the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$.

DEFINITION 15.3 (Light Edge). An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

THEOREM 15.1 (Cut Property / Safe Edge Theorem). *Let A be a subset of some MST $(S, V - S)$ be any cut that respects A (i.e., no edge in A crosses the cut). Let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.*

15.2 Kruskal's Algorithm

Idea: Kruskal's algorithm builds the MST by finding safe edges from the lightest to the heaviest. It treats the algorithm as managing a forest of trees. Initially, each vertex is its own tree.

1. Sort all edges in non-decreasing order of their weights.
2. Iterate through the sorted edges. For each edge (u, v), if u and v are in different trees (sets), add (u, v) to the forest and combine the two trees.
3. Repeat until all vertices are in the same tree.

15.2.1 Disjoint Set Data Structure

To implement Kruskal's efficiently, we use a **Disjoint Set Union (DSU)** data structure supporting:

- `Make-Set(v)`: Creates a new set containing only v.
- `Find-Set(v)`: Returns a representative element of the set containing v.
- `Union(u, v)`: Merges the sets containing u and v.

Algorithm 33 Kruskal's Algorithm

```
1: procedure MST-Kruskal(G, w)
2:     A = ∅
3:     for each vertex v ∈ G.V do
4:         Make-Set(v)
5:     end for
6:     Sort the edges of G.E into nondecreasing order by weight w
7:     for each edge (u, v) taken from the sorted list do
8:         if Find-Set(u) ≠ Find-Set(v) then
9:             A = A ∪ {(u, v)}
10:            Union(u, v)
11:        end if
12:    end for
13:    return A
14: end procedure
```

15.2.2 Analysis

- Initializing sets: $O(V)$.

- Sorting edges: $O(E \lg E)$.
- Disjoint-set operations: $O(E\alpha(V))$, where α is the inverse Ackermann function (extremely slow-growing, effectively constant).

Total runtime: $O(E \lg E)$. Since $E < V^2$, $\lg E = O(\lg V)$, so the time is dominated by sorting: $O(E \lg V)$.

15.3 Prim's Algorithm

Idea: Prim's algorithm works like Dijkstra's algorithm. It grows a single tree T from an arbitrary root r. At each step, it adds the lightest edge connecting a vertex in T to a vertex outside T.

- The vertices not yet in the tree reside in a **Min-Priority Queue** Q.
- The key of a vertex v, $v.key$, is the minimum weight of any edge connecting v to a vertex in the tree.
- $v.\pi$ is the parent of v in the tree.

Algorithm 34 Prim's Algorithm

```
1:  procedure MST-Prim(G, w, r)
2:      for each u ∈ G.V do
3:          u.key = ∞
4:          u.π = NIL
5:      end for
6:      r.key = 0
7:      Q = G.V                                  ▷ Build Min-Priority Queue
8:      while Q ≠ ∅ do
9:          u = Extract-Min(Q)
10:         for each v ∈ G.Adj[u] do
11:             if v ∈ Q and w(u, v) < v.key then
12:                 v.π = u
13:                 v.key = w(u, v)
14:                 Decrease-Key(Q, v, w(u, v))
15:             end if
16:         end for
17:     end while
18: end procedure
```

15.3.1 Analysis

The runtime depends on the implementation of the priority queue:

- **Binary Heap**:
  - `Build-Heap`: $O(V)$.
  - `Extract-Min`: $O(\lg V)$, called V times. Total $O(V \lg V)$.
  - `Decrease-Key`: $O(\lg V)$, called at most E times. Total $O(E \lg V)$.

  Total: $O(E \lg V)$.
- **Fibonacci Heap**:
  - `Extract-Min`: $O(\lg V)$ amortized.
  - `Decrease-Key`: $O(1)$ amortized.

  Total: $O(E + V \lg V)$.

15.4 Single-Source Shortest Paths

Problem: Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, find the shortest path from a source s to all other vertices v. The weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. The shortest-path weight $\delta(u, v)$ is $\min\{w(p) : u \rightsquigarrow v\}$.

15.4.1 Relaxation

The algorithms rely on the technique of **relaxation**. For each vertex v, we maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v.

15.5 Dijkstra's Algorithm

Assumption: Edge weights are non-negative ($w(u, v) \geq 0$).

Idea: Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights have been determined. It repeatedly selects the vertex $u \in V - S$ with the

Algorithm 35 Initialization and Relaxation

```
1: procedure INITIALIZE-SINGLE-SOURCE(G, s)
2:     for each vertex v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end procedure

8: procedure RELAX(u, v, w)
9:     if v.d > u.d + w(u, v) then
10:        v.d = u.d + w(u, v)
11:        v.π = u
12:    end if
13: end procedure
```

minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

Algorithm 36 Dijkstra's Algorithm

```
1: procedure DIJKSTRA(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     S = ∅
4:     Q = G.V                          ▷ Min-priority queue keyed by d values
5:     while Q ≠ ∅ do
6:         u = EXTRACT-MIN(Q)
7:         S = S ∪ {u}
8:         for each vertex v ∈ G.Adj[u] do
9:             RELAX(u, v, w)           ▷ Implicitly performs Decrease-Key if v ∈ Q
10:        end for
11:    end while
12: end procedure
```

15.5.1 Correctness

THEOREM 15.2. *Dijkstra's algorithm, run on a weighted, directed graph G with non-negative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.*

Proof. By invariant: At the start of each iteration of the **while** loop, $v.d = \delta(s, v)$ for each vertex $v \in S$. When a vertex u is added to S, its shortest path is already found. This relies on the fact that since weights are non-negative, taking a path through a vertex with a larger d value cannot result in a shorter path to u. □

15.5.2 Analysis

- **Using Binary Heap**: Each vertex extracted once: $O(V \lg V)$. Each edge relaxed once: $O(E \lg V)$. Total: $O(E \lg V)$.
- **Using Fibonacci Heap**: $O(V \lg V + E)$.

15.6 Summary of Algorithms

Algorithm	**Problem**	**Constraint**	**Runtime (Binary Heap)**
Kruskal	MST	Undirected	$O(E \lg V)$
Prim	MST	Undirected	$O(E \lg V)$
Dijkstra	Shortest Path	Non-negative weights	$O(E \lg V)$
Bellman-Ford	Shortest Path	No negative cycles	$O(VE)$

Table 9: Comparison of Graph Algorithms

Index