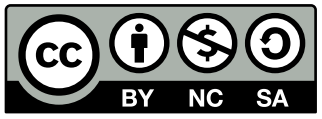
Data Structures and Algorithm Analysis

Kai Chen

December 15, 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 Introduction to Algorithms

1.1.1 Foundations

DEFINITION 1.1 (Algorithm). An **algorithm** is a well-defined computational procedure that takes some input and produces some 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.

We describe algorithms using **pseudocode**. This allows us to focus on the logical structure and ideas rather than syntax issues or idiosyncrasies of specific programming languages.

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.

Remark 1.1. Testing an algorithm on a few instances (e.g., "I tested my algorithm on 3 instances and it worked") is insufficient. Ideally, algorithms should be accompanied by a proof of correctness.

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.*
- ***Logical**: Shifts, comparisons.*
- ***Data movement**: Variable assignments (storing, retrieving).*
- ***Control**: Loops, subroutine/method calls.*

1.3 Insertion Sort

1.3.1 The Algorithm

Insertion Sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. The idea is to build up a sorted sequence by inserting the next element at the right position, similar to sorting a hand of playing cards.

1.3.2 Example Trace

Consider the input sequence: $A = \langle 5, 2, 4, 6, 1, 3 \rangle$.

1. Start with key $= 2$. Array becomes $\langle 2, 5, 4, 6, 1, 3 \rangle$.
2. Key $= 4$. Insert 4 into $\langle 2, 5 \rangle$. Array becomes $\langle 2, 4, 5, 6, 1, 3 \rangle$.
3. Key $= 6$. Insert 6 into $\langle 2, 4, 5 \rangle$. Array remains $\langle 2, 4, 5, 6, 1, 3 \rangle$.
4. Key $= 1$. Insert 1 into $\langle 2, 4, 5, 6 \rangle$. Array becomes $\langle 1, 2, 4, 5, 6, 3 \rangle$.
5. Key $= 3$. Insert 3 into $\langle 1, 2, 4, 5, 6 \rangle$. Final array: $\langle 1, 2, 3, 4, 5, 6 \rangle$.

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
```

1.4 Correctness via Loop Invariants

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

1.4.1 Loop Invariants

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.

1.4.2 Correctness of Insertion Sort

THEOREM 1.2. *The Insertion Sort algorithm (Algorithm 1) is correct.*

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: Before the first iteration ($j = 2$), the subarray is $A[1 \dots 1]$, which consists of the single element $A[1]$. A single element is trivially sorted.

Maintenance: The body of the loop (specifically the **while** loop) moves elements $A[j-1], A[j-2], \dots$ one position to the right until the correct position for $A[j]$ (stored in *key*) is found. $A[j]$ is then inserted. Thus, the new subarray $A[1 \dots j]$ contains the elements originally in $A[1 \dots j]$ but in sorted order.

Termination: The loop terminates when $j = n + 1$. Substituting this into the loop 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

DEFINITION 1.6 (Runtime)**.** The **runtime** of an algorithm on an input instance is the number of elementary operations executed by the RAM model. We focus on the asymptotic growth of runtime with respect to the problem size.

1.5.1 Analysis of Insertion Sort

Let c_i be the cost of executing line i, and let t_j be the number of times the **while** loop test is executed for a given value of j. The total running time $T(n)$ is the sum of the products of costs and execution counts:

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

Best Case Analysis The **best case** occurs when the array is already sorted. In this scenario, $A[i] \leq key$ immediately, so the while loop test is executed only once for each j (i.e., $t_j = 1$).

The running time becomes:

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

This can be expressed as a **linear function** of n: $T(n) = an + b$.

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 $A[1 \dots j-1]$, so $t_j = j$.

Using the summation formulas $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$, the running time becomes:

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

This can be expressed as a **quadratic function** of n: $T(n) = an^2 + bn + c$.

2 Runtime and Asymptotic Notation

2.1 Recap: Runtime of Insertion Sort

In the previous lecture, we derived a messy formula for the runtime of Insertion Sort. Let us simplify this analysis by considering specific cases.

2.1.1 Best, Worst, and Average Cases

- **Best Case**: The array is already sorted. The inner loop terminates immediately. The runtime is a linear function of n:

$$T(n) = an + b \quad \text{(Linear)}$$

- **Worst Case**: The array is reverse sorted. The inner loop runs maximally. The runtime is a quadratic function of n:

$$T(n) = an^2 + bn + c \quad \text{(Quadratic)}$$

- **Average Case**: We assume every permutation of the input is equally likely. For sorting, the average case is often as bad as the worst case (Quadratic).

Remark 2.1. Why focus on the **Worst Case**?

1. It provides a guarantee that the algorithm will never take longer.
2. For some algorithms, the worst case occurs frequently.

2.2 Asymptotic Notation

To compare algorithms effectively, we ignore constant factors and small-order terms, focusing on how the runtime grows as the problem size n becomes very large. This is called **Asymptotic Analysis**.

2.2.1 Definitions

We use Greek letters to describe the bounding behavior of functions.

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

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

Intuition: $f(n)$ grows **as fast as** $g(n)$. $g(n)$ is an asymptotically **tight bound** for $f(n)$.

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

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

Intuition: $f(n)$ grows **at most as fast as** $g(n)$. It provides an **asymptotic upper bound**.

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

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

Intuition: $f(n)$ grows **at least as fast as** $g(n)$. It provides an **asymptotic lower bound**.

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

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

2.2.2 Strict Bounds

DEFINITION 2.4 (Little-o and Little-omega).

- $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. (Strictly slower)
- $f(n) = \omega(g(n))$ if $g(n) = o(f(n))$, i.e., limit is ∞. (Strictly faster)

2.2.3 Analogy to Arithmetic Comparisons

Notation	Arithmetic Analogy	Meaning
$f(n) = O(g(n))$	$f \leq g$	Grows at most as fast as
$f(n) = \Omega(g(n))$	$f \geq g$	Grows at least as fast as
$f(n) = \Theta(g(n))$	$f = g$	Grows as fast as
$f(n) = o(g(n))$	$f < g$	Grows strictly slower than
$f(n) = \omega(g(n))$	$f > g$	Grows strictly faster than

Table 1: Asymptotic Notation Analogy

2.3 Common Runtimes and Rules

2.3.1 Common Growth Classes

Ordered from slowest to fastest growth:

1. $\Theta(1)$: Constant time.
2. $\Theta(\log n)$: Logarithmic time.
3. $\Theta(n)$: Linear time.
4. $\Theta(n \log n)$: Linearithmic time (often seen in good sorting algorithms).
5. $\Theta(n^2)$: Quadratic time.
6. $\Theta(n^3)$: Cubic time.
7. $\Theta(n^k)$: Polynomial time.
8. $\Theta(2^n)$: Exponential time.

Note 2.1. Every polynomial of $\log n$ grows strictly slower than every polynomial of n. Every polynomial of n grows strictly slower than every exponential function 2^n.

2.3.2 Simplification Rules

To make runtime analysis simple, we apply these rules for non-negative functions:

Property 2.2 (Sum Rule)**.** *Slower functions can be ignored.*

$$f(n) + g(n) = \Theta(\max(f(n), g(n)))$$

Example: $2n^2 + 10n = \Theta(n^2)$.

Property 2.3 (Product Rule)**.** *Asymptotic times can be multiplied.*

$$\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$$

2.4 Revisiting Insertion Sort

Using asymptotic notation, we can express the runtime of Insertion Sort more elegantly:

- **Best Case**: The runtime is $\Theta(n)$.
- **Worst Case**: The runtime is $\Theta(n^2)$.

Therefore, for a general input, the runtime of Insertion Sort is:

- $\Omega(n)$: It grows at least as fast as linear (best case).
- $O(n^2)$: It grows at most as fast as quadratic (worst case).

Note: It is **incorrect** to say the runtime is $\Theta(n^2)$ for *all* cases, because the best case is linear. However, it is correct to say the *worst-case* runtime is $\Theta(n^2)$.

3 Divide-and-Conquer

3.1 Design Paradigms

3.1.1 Incremental vs. Divide-and-Conquer

- **Incremental Approach** (e.g., Insertion Sort): We build up a solution incrementally. Having sorted $A[1 \dots j-1]$, we insert $A[j]$ into its proper place to get a sorted subarray $A[1 \dots j]$.
- **Divide-and-Conquer**: This paradigm involves three steps:
  1. **Divide**: Break the problem into smaller subproblems (smaller instances of the original problem).
  2. **Conquer**: Solve these subproblems recursively. (If small enough, solve directly).
  3. **Combine**: Combine the solutions to the subproblems into the solution for the original problem.

3.2 Merge Sort

Merge Sort is a classic sorting algorithm that follows the divide-and-conquer paradigm.

3.2.1 The Algorithm

1. **Divide**: Divide the n-element sequence into two subsequences of $n/2$ elements each.
2. **Conquer**: Sort the two subsequences recursively using Merge Sort. (Base case: sequence of size 1 is trivially sorted).
3. **Combine**: Merge the two sorted subsequences to produce the sorted answer.

Algorithm 2 Merge Sort

```
1: procedure MergeSort(A, p, r)
2:     if p < r then
3:         q = ⌊(p + r)/2⌋                          ▷ Divide
4:         MergeSort(A, p, q)                       ▷ Conquer Left
5:         MergeSort(A, q + 1, r)                   ▷ Conquer Right
6:         Merge(A, p, q, r)                        ▷ Combine
7:     end if
8: end procedure
```

3.2.2 The Merge Procedure

The key component is the 'Merge(A, p, q, r)' procedure, which merges two sorted subarrays $A[p \dots q]$ and $A[q+1 \dots r]$ into a single sorted subarray $A[p \dots r]$.

Idea: Copy the two subarrays into temporary arrays L and R. Then, iterate through L and R, picking the smallest element at each step and placing it back into A.

Runtime of Merge: $\Theta(n)$, where $n = r - p + 1$. It iterates through the elements linearly.

3.3 Analysis of Merge Sort

3.3.1 Recurrence Relation

Let $T(n)$ be the time for Merge Sort to sort n elements.

- **Divide**: Computing the middle index takes $\Theta(1)$.
- **Conquer**: We solve two subproblems of size $n/2$, taking $2T(n/2)$.
- **Combine**: The Merge procedure takes $\Theta(n)$.

Thus, the recurrence for Merge Sort 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 Solving the Recurrence

We can solve this using the **Master Theorem** (see below) or a Recursion Tree. Using a Recursion Tree:

- Level 0: Cost cn
- Level 1: Two nodes, each costs $cn/2 \to$ Total cn
- Level i: 2^i nodes, each costs $cn/2^i \to$ Total cn
- Height of tree: $\lg n$

Total cost = (Cost per level) $\times$ (Number of levels) $\approx cn \times \lg n = \Theta(n \lg n)$.

THEOREM 3.1. *The runtime of Merge Sort is* $\Theta(n \lg n)$.

Remark 3.1. Comparison with Insertion Sort:

- Merge Sort: $\Theta(n \lg n)$ in all cases (Best, Worst, Average).
- Insertion Sort: $\Theta(n^2)$ worst case, but $\Theta(n)$ best case.
- **Space Complexity**: Merge Sort is **not** in-place; it requires $\Theta(n)$ auxiliary space for the 'Merge' step. Insertion Sort is in-place ($O(1)$ extra space).

3.4 The Master Theorem

The Master Theorem provides a "cookbook" method for solving recurrences of the form:

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

where $a \geq 1$, $b > 1$ are constants, and $f(n)$ is an asymptotically positive function. Here, $n^{\log_b a}$ is called the **watershed function**.

THEOREM 3.2 (Master Theorem). *Compare* $f(n)$ *with* $n^{\log_b a}$*:*

1. ***Case 1**: If* $f(n) = O(n^{\log_b a - \epsilon})$ *for some constant* $\epsilon > 0$*, then* $T(n) = \Theta(n^{\log_b a})$*. (Watershed dominates polynomially).*
2. ***Case 2**: If* $f(n) = \Theta(n^{\log_b a} \lg^k n)$ *for some* $k \geq 0$*, then* $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$*. (Typically* $k = 0$*, so* $f(n) \approx n^{\log_b a}$*, then* $T(n) = \Theta(n^{\log_b a} \lg n)$*).*
3. ***Case 3**: If* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *for some constant* $\epsilon > 0$*, and if* $af(n/b) \leq cf(n)$

for some constant $c < 1$ *and sufficiently large* n *(Regularity Condition), then* $T(n) = \Theta(f(n))$*. (Driving function dominates).*

Example 3.1 (Applying Master Theorem to Merge Sort). $T(n) = 2T(n/2) + \Theta(n)$.

- $a = 2, b = 2, f(n) = \Theta(n)$.
- Watershed: $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.
- Compare $f(n)$ and watershed: $f(n) = \Theta(n)$ matches watershed (Case 2 with $k = 0$).
- Solution: $T(n) = \Theta(n^{\log_2 2} \lg^{0+1} n) = \Theta(n \lg n)$.

4 Heapsort

4.1 Heaps

4.1.1 Concept

A **Heap** is a specialized tree-based data structure that satisfies the heap property. It can be visualized as a nearly complete binary tree.

DEFINITION 4.1 (Heap Property)**.** There are two kinds of heaps:

- **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.
- **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.

4.1.2 Array Representation

A heap can be stored efficiently in an array A. For a node at index i:

- $\text{Parent}(i) = \lfloor i/2 \rfloor$
- $\text{Left}(i) = 2i$
- $\text{Right}(i) = 2i + 1$

The attribute A.heap-size represents the number of elements in the heap stored within array A. Note that $0 \leq A.\text{heap-size} \leq A.\text{length}$.

4.2 Maintaining the Heap Property

4.2.1 Max-Heapify

The procedure `Max-Heapify` is used to maintain the max-heap property. It assumes that the binary trees rooted at $\text{Left}(i)$ and $\text{Right}(i)$ are max-heaps, but $A[i]$ might be smaller than its children, violating the property. The idea is to let the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Algorithm 3 Max-Heapify

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

Runtime: The running time of `Max-Heapify` on a node of height h is $O(h)$. Since the height of a heap of n elements is $\Theta(\lg n)$, the time complexity is $O(\lg n)$.

4.3 Building a Heap

We can use `Max-Heapify` in a bottom-up manner to convert an array $A[1 \ldots n]$ into a max-heap. The elements in the subarray $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are all leaves of the tree, so they are essentially 1-element heaps to begin with.

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
```

Correctness: The loop invariant is that at the start of each iteration of the **for** loop, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.

Runtime: A simple upper bound is $O(n \lg n)$ because we call `Max-Heapify` $O(n)$ times. However, a tighter analysis shows that the runtime is actually linear, i.e., $O(n)$. This is

because most nodes have small heights, and the time for `Max-Heapify` depends on height.

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

4.4 The Heapsort Algorithm

The Heapsort algorithm proceeds as follows:

1. Build a max-heap from the input array.
2. The maximum element is at the root $A[1]$.
3. Swap $A[1]$ with the last element $A[n]$. Now the max element is at the correct final position.
4. Discard node n from the heap (by decrementing heap-size).
5. The new root might violate the max-heap property, so call `Max-Heapify(A, 1)` to fix it.
6. Repeat the process for the remaining heap of size $n - 1$ down to 2.

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
```

4.5 Analysis of Heapsort

4.5.1 Runtime Complexity

- `Build-Max-Heap` takes $O(n)$.
- The **for** loop executes $n - 1$ times.
- Each call to `Max-Heapify` takes $O(\lg n)$.

Thus, the total running time is $O(n) + (n - 1)O(\lg n) = O(n \lg n)$.

4.5.2 Comparison

- **Time Complexity**: $O(n \lg n)$ in all cases (Best, Average, Worst). This is better than the worst-case of Insertion Sort ($O(n^2)$) and matches Merge Sort.
- **Space Complexity**: Heapsort sorts **in place**, meaning it uses only $O(1)$ auxiliary memory. This is an advantage over Merge Sort, which requires $O(n)$ auxiliary space.

5 Quicksort

5.1 Overview

Quicksort is a popular sorting algorithm that is fast in practice, despite having a worst-case running time of $\Theta(n^2)$. It follows the divide-and-conquer paradigm.

- **Divide**: Pick an element as the **pivot**. Partition the array so that all elements smaller than the pivot are to its left, and all elements larger are to its right. The pivot is moved to its final sorted position.
- **Conquer**: Recursively sort the two subarrays (elements smaller than pivot and elements larger than pivot).
- **Combine**: No work is needed; the array is already sorted after the recursion.

5.2 The Algorithm

The core of Quicksort is the `Partition` procedure. Unlike Merge Sort, which does the heavy lifting in the "Combine" step (Merge), Quicksort does it in the "Divide" step (Partition).

Algorithm 6 Quicksort

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

5.2.1 Partitioning

The `Partition` procedure rearranges the subarray $A[p \dots r]$ in place. It selects the last element $A[r]$ as the pivot. It maintains four regions in the array during execution:

1. $A[p \dots i]$: Elements $\leq$ pivot.

2. $A[i+1 \dots j-1]$: Elements $>$ pivot.
3. $A[j \dots r-1]$: Unrestricted (not yet scanned).
4. $A[r]$: The pivot.

Algorithm 7 Partition

```
1: procedure PARTITION(A, p, r)
2:     x = A[r]                                   ▷ Pivot
3:     i = p − 1
4:     for j = p to r − 1 do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             Exchange A[i] with A[j]
8:         end if
9:     end for
10:    Exchange A[i + 1] with A[r]                ▷ Put pivot in correct place
11:    return i + 1
12: end procedure
```

Correctness: The loop invariant is: At the start of each iteration of the loop, for any index k:

- If $p \leq k \leq i$, then $A[k] \leq x$.
- If $i+1 \leq k \leq j-1$, then $A[k] > x$.

Upon termination, the pivot is placed at index $i+1$, satisfying the partition property.

Runtime of Partition: The procedure makes a single pass over the array $A[p \dots r]$, performing constant work per element. Thus, the runtime is $\Theta(n)$, where $n = r - p + 1$.

5.3 Performance Analysis

The running time of Quicksort depends heavily on how balanced the partitioning is.

5.3.1 Worst-case Partitioning

The worst case occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements. This happens, for example, when the input array is already sorted or reverse sorted. The recurrence is:

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

Solving this yields $T(n) = \Theta(n^2)$.

5.3.2 Best-case Partitioning

The best case occurs when the partition splits the problem into two roughly equal sizes, similar to Merge Sort. The recurrence is:

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

Solving this yields $T(n) = \Theta(n \lg n)$.

5.3.3 Average-case Partitioning

Suppose the split is always proportional, e.g., 9-to-1. The recurrence becomes:

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$

The recursion tree still has logarithmic depth ($\log_{10/9} n$), and the cost at each level is at most cn. Thus, the runtime remains $O(n \lg n)$.

5.3.4 Rigorous Average-Case Analysis

If we assume all input permutations are equally likely, or if we use a randomized pivot selection, the average runtime is $O(n \lg n)$. The recurrence for the average case considers the pivot landing with equal probability at any rank $q \in \{1, \ldots, n\}$:

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

Solving this using the substitution method proves that the expected runtime is $O(n \lg n)$.

5.4 Randomized Quicksort

To avoid the $\Theta(n^2)$ worst-case behavior on specific inputs (like sorted arrays), we can introduce randomization. Instead of always picking $A[r]$ as the pivot, we randomly choose an element from $A[p \dots r]$, swap it with $A[r]$, and then partition.

Algorithm 8 Randomized Partition

```
1: procedure RANDOMIZED-PARTITION(A, p, r)
2:     i = RANDOM(p, r)
3:     Exchange A[r] with A[i]
4:     return PARTITION(A, p, r)
5: end procedure
```

This ensures that the expected running time is $O(n \lg n)$ regardless of the input distribution. The worst case is determined only by the random number generator, not the input data.

6 Randomisation & Lower Bounds

6.1 Randomised Algorithms

In the previous lecture, we saw that QuickSort has a worst-case runtime of $\Theta(n^2)$ when the pivot is chosen poorly (e.g., always the last element on a sorted array). To address this, we can introduce randomness into the algorithm design.

6.1.1 Randomised QuickSort

Instead of always picking a fixed element (like $A[r]$) as the pivot, we choose the pivot **uniformly at random** from the subarray $A[p \dots r]$.

Algorithm 9 Randomised Partition

```
1: procedure RANDOMISED-PARTITION(A, p, r)
2:     i = RANDOM(p, r)                ▷ Pick i ∈ [p, r] uniformly at random
3:     Exchange A[r] with A[i]
4:     return PARTITION(A, p, r)
5: end procedure
```

Algorithm 10 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
```

Remark 6.1. **Performance**:

- There is **no worst-case input** for Randomised QuickSort.
- Because the pivot is chosen randomly, every split is equally likely regardless of the input patterns (e.g., sorted or reverse sorted arrays).
- The runtime is a random variable, but the expected runtime is good for *all* inputs.

6.2 Analysis of Randomised QuickSort

For randomised algorithms, we analyze the **expected running time** $E[T(n)]$. We use the number of comparisons as a proxy for the runtime, since other operations are proportional to the number of comparisons.

6.2.1 Expectation and Linearity

DEFINITION 6.1 (Expectation)**.** The expectation of a discrete random variable X is given by:

$$E[X] = \sum_x x \cdot \Pr(X = x)$$

Property 6.1 (Linearity of Expectation)**.** *For any random variables X_1, X_2:*

$$E[X_1 + X_2] = E[X_1] + E[X_2]$$

This holds even if the variables are dependent.

6.2.2 Expected Number of Comparisons

Let X be the total number of comparisons performed by Randomised QuickSort. We want to show that $E[X] = O(n \lg n)$.

Let $z_1 < z_2 < \cdots < z_n$ be the elements of the array A sorted in increasing order. Define indicator random variables $X_{i,j}$ for $i < j$:

$$X_{i,j} = \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \\ 0 & \text{otherwise} \end{cases}$$

The total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$. By linearity of expectation:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(z_i \text{ is compared to } z_j)$$

Probability of Comparison: Elements z_i and z_j are compared if and only if the first element chosen as a pivot from the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ is either z_i or z_j.

- If a pivot x is chosen such that $z_i < x < z_j$, then z_i and z_j are separated into different subarrays and will never be compared.
- The set Z_{ij} has $j - i + 1$ elements.
- Since pivots are chosen uniformly at random, any element in Z_{ij} is equally likely to be the first one chosen.

Thus:

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

Summation:

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

Let $k = j - i$. As j goes from $i + 1$ to n, k goes from 1 to $n - i$.

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

THEOREM 6.2. *The expected running time of Randomised QuickSort is $O(n \lg n)$.*

6.3 Lower Bounds for Sorting

We have seen several algorithms (Merge Sort, Heap Sort, Quick Sort) that run in $O(n \lg n)$. Can we sort faster?

6.3.1 Comparison Sorts

DEFINITION 6.2 (Comparison Sort). A **comparison sort** is a sorting algorithm that determines the sorted order of elements only by comparing them (e.g., testing $a_i < a_j$). Examples include Insertion Sort, Merge Sort, Heap Sort, and Quick Sort.

6.3.2 Decision Trees

We can view any comparison sort algorithm as a **Decision Tree**.

- **Internal Nodes**: Comparisons $a_i : a_j$.
- **Branches**: Outcomes of comparisons ($\leq$ or $>$).
- **Leaves**: Permutations of the input (possible sorted orders).
- **Execution**: A path from the root to a leaf. The length of the path is the number of comparisons made.

6.3.3 The Lower Bound Theorem

THEOREM 6.3 (Lower Bound for Comparison Sorts)**.** *Any comparison sort algorithm requires* $\Omega(n \lg n)$ *comparisons in the worst case.*

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

1. There are $n!$ possible permutations of n distinct elements. A correct sorting algorithm must be able to produce any of these as output. Thus, the tree must have at least $n!$ leaves: $L \geq n!$.
2. A binary tree of height h has at most 2^h leaves. Thus:

$$2^h \geq L \geq n!$$

3. Taking logarithms:

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

4. Using Stirling's approximation ($n! \approx (n/e)^n$):

$$\lg(n!) = \Theta(n \lg n)$$

5. Therefore, the worst-case number of comparisons (height of the tree) is $h = \Omega(n \lg n)$.

□

Corollary 6.4. *Heap Sort and Merge Sort are* ***asymptotically optimal*** *comparison sorts because their upper bounds* ($O(n \lg n)$) *match the lower bound* ($\Omega(n \lg n)$).

6.4 Summary of Sorting Algorithms

Algorithm	**Avg Time**	**Worst Time**	**Space**	**Stable?**
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Merge Sort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$	Yes
Heap Sort	$O(n \lg n)$	$O(n \lg n)$	$O(1)$	No
Quick Sort	$O(n \lg n)$	$O(n^2)$	$O(\lg n)$	No
Rand. Quick Sort	$O(n \lg n)$	$O(n^2)$	$O(\lg n)$	No

Table 2: Comparison of Sorting Algorithms

7 Sorting in Linear Time

7.1 Beyond Comparison Sorts

We established a lower bound of $\Omega(n \lg n)$ for any comparison sort. However, if we know more about the input elements (e.g., they are integers in a specific range), we can sort them faster, even in linear time $\Theta(n)$. This is achieved by algorithms that do not rely solely on element comparisons.

7.2 Counting Sort

Assumption: The input consists of integers in the range $\{0, \ldots, k\}$.

Idea: For each element x, count the number of elements less than or equal to x. This information allows us to place x directly into its correct position in the output array.

Algorithm 11 Counting Sort

```
1:  procedure COUNTING-SORT(A, B, k)
2:      Let C[0...k] be a new array
3:      for i = 0 to k do
4:          C[i] = 0                                  ▷ Initialize counter array
5:      end for
6:      for j = 1 to A.length do
7:          C[A[j]] = C[A[j]] + 1                     ▷ Count occurrences of each element
8:      end for
9:      for i = 1 to k do
10:         C[i] = C[i] + C[i − 1]                    ▷ Compute running sum: #elements ≤ i
11:     end for
12:     for j = A.length downto 1 do
13:         B[C[A[j]]] = A[j]                         ▷ Place element in sorted position
14:         C[A[j]] = C[A[j]] − 1                     ▷ Decrement count for stability
15:     end for
16: end procedure
```

7.2.1 Analysis

- **Input**: $A[1 \ldots n]$ (input array), $B[1 \ldots n]$ (output array), k (max value).
- **Time Complexity**:

- Initializing C: $\Theta(k)$.
- Counting occurrences: $\Theta(n)$.
- Computing running sums: $\Theta(k)$.
- Placing elements: $\Theta(n)$.

Total Time: $\Theta(n+k)$. If $k = O(n)$, the runtime is $\Theta(n)$.

- **Space Complexity**: $\Theta(n+k)$ for arrays B and C.
- **Stability**: Counting Sort is **stable**. The loop in line 8 iterates **downto** 1, ensuring that elements with the same value appear in the output in the same relative order as they did in the input.

7.3 Radix Sort

Counting Sort is efficient when k is small. If k is large (e.g., polynomial in n), we can use **Radix Sort**.

Idea: Sort the numbers digit by digit, starting from the least significant digit (LSD) to the most significant digit (MSD). Crucially, the sort used for each digit must be **stable**.

Algorithm 12 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
```

7.3.1 Example

Sort: 329, 457, 657, 839, 436, 720, 355

1. Sort by 1st digit: 720, 355, 436, 457, 657, 329, 839
2. Sort by 2nd digit: 720, 329, 436, 839, 355, 457, 657
3. Sort by 3rd digit: 329, 355, 436, 457, 657, 720, 839 (Sorted!)

7.3.2 Analysis

- **Correctness**: Follows from the stability of the intermediate sort. By induction: after sorting on digit i, the array is sorted with respect to the last i digits.
- **Time Complexity**: Given n numbers with d digits, where each digit can take up to k values. Total Time: $\Theta(d(n+k))$ if using Counting Sort for each digit. If d is constant and $k = O(n)$, the runtime is $\Theta(n)$.

7.3.3 Applications

Consider sorting n integers in the range 0 to $n^3 - 1$.

- Comparison Sort: $\Theta(n \lg n)$.
- Counting Sort: $k = n^3$, so $\Theta(n + n^3) = \Theta(n^3)$. Too slow.
- Radix Sort: Treat numbers as base-n.
  - Then each "digit" ranges from 0 to $n - 1$, so $k = n$.
  - The number of digits $d = \log_n(n^3) = 3$.
  - Total Time: $\Theta(3(n + n)) = \Theta(n)$. Linear time!

8 Elementary Data Structures

8.1 Introduction

Data structures are methods for representing finite dynamic sets of elements that can be stored and retrieved. Each element typically contains a **key** (used to identify the element), **satellite data** (payload), and identifying **attributes** (pointers, etc.).

8.1.1 Dynamic Set Operations

Operations on dynamic sets are generally categorized into queries and modifying operations:

- **Search**(S, k): Returns a pointer to an element $x \in S$ with key k, or NIL.
- **Insert**(S, x): Adds element x to S.
- **Delete**(S, x): Removes element x from S (given a pointer to x).
- **Minimum**(S), **Maximum**(S): Returns element with smallest/largest key.
- **Successor**(S, x), **Predecessor**(S, x): Returns the next larger/smaller element.

8.2 Stacks

DEFINITION 8.1 (Stack). A **stack** is a dynamic set that follows the **Last-In, First-Out (LIFO)** policy. The element deleted is always the one that was most recently inserted.

8.2.1 Operations

Stacks are typically implemented using an array S with an attribute $S.top$.

- **Push**: Inserts an element.
- **Pop**: Deletes and returns the top element.
- **Stack-Empty**: Checks if the stack is empty.

Remark 8.1. All stack operations take $O(1)$ time.

Algorithm 13 Stack Operations

```
1: procedure Stack-Empty(S)
2:     if S.top == 0 then return TRUE
3:     else return FALSE
4:     end if
5: end procedure

6: procedure Push(S, x)
7:     if S.top == S.size then Error"overflow"
8:     else
9:         S.top = S.top + 1
10:        S[S.top] = x
11:    end if
12: end procedure

13: procedure Pop(S)
14:    if Stack-Empty(S) then Error"underflow"
15:    else
16:        S.top = S.top − 1
17:        return S[S.top + 1]
18:    end if
19: end procedure
```

8.2.2 Applications

1. **Bracket Balance Checking**: Ensure parentheses, braces, and brackets are properly nested. Push opening brackets; pop and match on closing brackets.
2. **Postfix Expression Evaluation**: Evaluate arithmetic expressions like 9 3+4 2∗∗7+. Push operands; pop operands and push result when an operator is encountered.

8.3 Queues

DEFINITION 8.2 (Queue). A **queue** is a dynamic set that follows the **First-In, First-Out (FIFO)** policy. The element deleted is always the one that has been in the set for the longest time.

8.3.1 Operations

Queues can be implemented using an array Q treated as a circular buffer ("wrapped around"). It maintains $Q.head$ (index of the first element) and $Q.tail$ (index where the next element will be inserted).

- **Enqueue**: Inserts an element at the tail.
- **Dequeue**: Removes and returns the element at the head.

Algorithm 14 Queue Operations

```
1: procedure ENQUEUE(Q, x)
2:     Q[Q.tail] = x
3:     if Q.tail == Q.size then Q.tail = 1
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
10:    else Q.head = Q.head + 1
11:    end if
12:    return x
13: end procedure
```

Remark 8.2. All queue operations take $O(1)$ time.

8.4 Linked Lists

DEFINITION 8.3 (Linked List)**.** A **linked list** is a data structure in which objects are arranged in a linear order. Unlike an array, the order is determined by a pointer in each object.

Each element x has:

- **key**: The data.
- **next**: Pointer to the next element.
- **prev**: Pointer to the previous element (in a doubly linked list).

8.4.1 Comparison with Arrays

- **Disadvantages of Arrays**: Fixed size (resizing is costly), inserting/deleting in the middle requires shifting elements ($\Theta(n)$).
- **Advantages of Linked Lists**: Dynamic size, efficient insertion/deletion if the position is known.

8.4.2 Operations

Searching: To find an element with key k, we must traverse the list linearly. **Runtime**:

Algorithm 15 List Search

```
1: procedure LIST-SEARCH(L, k)
2:     x = L.head
3:     while x ≠ NIL and x.key ≠ k do
4:         x = x.next
5:     end while
6:     return x
7: end procedure
```

$\Theta(n)$ in the worst case.

Insertion: Inserting an element x at the front of the list (or after a known node) involves only pointer updates. **Runtime**: $O(1)$.

Algorithm 16 List Insert (Prepend)

```
1: procedure LIST-PREPEND(L, x)
2:     x.next = L.head
3:     x.prev = NIL
4:     if L.head ≠ NIL then
5:         L.head.prev = x
6:     end if
7:     L.head = x
8: end procedure
```

Deletion: If we are given a pointer to the element x to be deleted, we can splice it out by updating pointers. **Runtime**: $O(1)$ if the element pointer is known. If only the key is known, we must search first, taking $\Theta(n)$.

Algorithm 17 List Delete

```
1: procedure List-Delete(L, x)
2:     if x.prev ≠ NIL then
3:         x.prev.next = x.next
4:     else
5:         L.head = x.next
6:     end if
7:     if x.next ≠ NIL then
8:         x.next.prev = x.prev
9:     end if
10: end procedure
```

8.5 Summary of Elementary Structures

Structure	Insertion	Deletion	Search
Stack	$O(1)$ (Push)	$O(1)$ (Pop)	N/A
Queue	$O(1)$ (Enqueue)	$O(1)$ (Dequeue)	N/A
Linked List	$O(1)$ (at head)	$O(1)$ (given ptr)	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(n)$	$O(\lg n)$
Priority Queue (Heap)	$O(\lg n)$	$O(\lg n)$	N/A

Table 3: Complexity Comparison of Elementary Data Structures

9 Binary Search Trees

9.1 Introduction to Trees

9.1.1 Definitions

DEFINITION 9.1 (Binary Tree)**.** A **binary tree** is a finite set of nodes that is either empty or consists of a **root** and two disjoint binary trees called the **left subtree** and the **right subtree**.

Key terminology:

- **Leaf**: A node with no children.
- **Height of a node**: Length of the longest simple path from the node to a leaf.
- **Height of a tree**: Height of the root.
- **Depth of a node**: Length of the path from the root to the node.
- **Full binary tree**: Every node is either a leaf or has exactly two children.

9.1.2 Inductive Proofs on Trees

Recursive definitions allow for inductive proofs.

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

Proof. **Base case**: Height 0, 1 leaf ($2^0 = 1$). True. **Inductive step**: Assume true for height $h-1$. A tree of height h has a root and two subtrees of height at most $h-1$. Total leaves $\leq 2^{h-1} + 2^{h-1} = 2^h$. □

9.2 Binary Search Trees (BST)

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

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

A BST is represented by a linked data structure where each node has attributes: ‘key‘, ‘left‘, ‘right‘, and ‘p‘ (parent).

[Image of a binary search tree]

9.3 Operations on BSTs

All dynamic set operations can be supported by a BST. The time complexity for most operations is $O(h)$, where h is the height of the tree.

9.3.1 Searching

Algorithm 18 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
```

Runtime: $O(h)$.

9.3.2 Minimum and Maximum

- **Minimum**: Follow ‘left‘ pointers until NIL.
- **Maximum**: Follow ‘right‘ pointers until NIL.

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 right subtree is non-empty, successor is the minimum in the right subtree.

- Case 2: If right subtree is empty, go up the tree until we encounter a node that is the left child of its parent.

Runtime: $O(h)$.

9.3.4 Insertion

To insert value z, trace a path downward from the root, moving left or right depending on comparisons with $z.key$, until a NIL is reached. Replace NIL with z.

Algorithm 19 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
12:    else if z.key < y.key then y.left = z
13:    else y.right = z
14:    end if
15: end procedure
```

Runtime: $O(h)$.

9.3.5 Deletion

Deleting node z involves three cases:

1. z has no children: Simply remove z.
2. z has one child: Replace z with its child.
3. z has two children: Find z's successor y (which lies in z's right subtree and has no left child). Replace z with y.

We use a helper subroutine `Transplant(T, u, v)` to replace subtree rooted at u with subtree rooted at v.

Runtime: $O(h)$.

Algorithm 20 Transplant

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

9.4 Tree Walks

We can traverse a BST to output keys.

- **Inorder Tree Walk**: Prints keys in sorted order. (Left, Root, Right).
- **Preorder**: Root, Left, Right.
- **Postorder**: Left, Right, Root.

Algorithm 21 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. *An inorder tree walk of an n-node BST takes $\Theta(n)$ time.*

9.5 Worst Case Analysis

The height h of a BST determines the runtime of basic operations.

- **Best case**: The tree is balanced, $h = O(\lg n)$.
- **Worst case**: The tree degenerates into a linear chain (e.g., inserting sorted data), $h = \Theta(n)$.

Thus, basic operations take $O(n)$ in the worst case. To guarantee $O(\lg n)$, we need **balanced** search trees (e.g., Red-Black Trees, AVL Trees).

10 AVL Trees

10.1 Self-Balancing Binary Search Trees

Ordinary BSTs can degenerate into linear chains (height $\Theta(n)$). To guarantee $O(\lg n)$ runtime for operations, we need to keep the tree balanced. AVL trees are a class of self-balancing BSTs.

10.2 Definition and Properties

DEFINITION 10.1 (AVL Tree)**.** An **AVL tree** is a binary search tree that satisfies the **AVL property**: For every node v, the heights of the 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:

$$bal(v) = h(T_L) - h(T_R)$$

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

THEOREM 10.1. *The height of an AVL tree with n nodes is $O(\lg n)$. Specifically, $h \approx 1.44 \lg n$.*

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

- $A(0) = 1$
- $A(1) = 2$
- $A(h) = 1 + A(h-1) + A(h-2)$ (Root + min nodes in subtrees of heights $h-1$ and $h-2$).

This recurrence is related to Fibonacci numbers: $A(h) = Fib(h+2) - 1$. Since $Fib(h) \approx \phi^h$, n grows exponentially with h, implying h grows logarithmically with n. □

10.3 Operations

10.3.1 Searching

Searching in an AVL tree is identical to searching in a standard BST. Since $h = O(\lg n)$, search takes $O(\lg n)$.

10.3.2 Rotations

Rotations are local operations that change the tree structure while preserving the BST property. They are used to rebalance the tree.

- **Left Rotation**: Moves a node down to the left and its right child up.
- **Right Rotation**: Moves a node down to the right and its left child up.

10.3.3 Insertion

Insert the node as in a standard BST. Then, trace the path back up to the root, updating balance factors. If a node v becomes unbalanced (balance factor becomes $+2$ or -2), perform rotations to fix it.

Let v be the lowest unbalanced node.

1. **Left-Left Case**: Left child x is heavy on the left. Fix with **Right Rotation** at v.
2. **Right-Right Case**: Right child x is heavy on the right. Fix with **Left Rotation** at v.
3. **Left-Right Case**: Left child x is heavy on the right. Fix with **Left Rotation** at x, then **Right Rotation** at v (Double Rotation).
4. **Right-Left Case**: Right child x is heavy on the left. Fix with **Right Rotation** at x, then **Left Rotation** at v (Double Rotation).

Runtime: $O(\lg n)$ to insert and trace back. Rotations take $O(1)$. Total $O(\lg n)$.

10.3.4 Deletion

Delete the node as in a standard BST. Then, trace the path back up to the root. Unlike insertion, fixing one imbalance might create another imbalance higher up. Thus, we may need $O(h)$ rotations. **Runtime**: $O(\lg n)$.

10.4 Summary

- **Height**: Always $O(\lg n)$.
- **Operations**: Search, Insert, Delete, Min, Max, Successor all take $O(\lg n)$.
- **Overhead**: Need to store height or balance factor at each node. Rotations add some constant overhead to updates.

11 Dynamic Programming

11.1 Introduction

Dynamic Programming (DP) is a general algorithm design paradigm used to solve optimization problems. It is applicable when the solution can be viewed as the result of a sequence of decisions.

Core Idea:

- Break down a problem into smaller subproblems.
- Solve these subproblems and save their solutions in a table (avoid recomputing).
- Solve subproblems of increasing size until the original problem is solved.

11.2 Fibonacci Sequence: A Motivating Example

The Fibonacci numbers are defined as:

$$Fib(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{if } n > 1 \end{cases}$$

A naive recursive implementation computes the same values many times. The runtime $T(n)$ grows exponentially: $T(n) = \Omega((\frac{1+\sqrt{5}}{2})^n)$.

DP Approach (Bottom-Up): Compute values from $Fib(0)$ up to $Fib(n)$ and store them in a table. **Runtime**: $O(n)$.

11.3 Elements of Dynamic Programming

1. **Optimal Substructure**: An optimal solution to the problem contains within it optimal solutions to subproblems.
2. **Overlapping Subproblems**: The space of subproblems is small, and a recursive algorithm visits the same subproblems repeatedly.

11.4 Rod Cutting Problem

Problem: Given a steel rod of length n and a table of prices p_i for rods of length $i = 1, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

11.4.1 Optimal Substructure

To maximize revenue for a rod of length n, we can make a first cut of length i ($1 \leq i \leq n$) and then optimally cut the remaining $n - i$ length. This leads to the **Bellman Equation**:

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

with base case $r_0 = 0$.

11.4.2 Algorithms

Naive Recursive Solution Directly implements the Bellman equation. **Runtime**: $T(n) = 2^n$ (Exponential).

Bottom-Up DP Sort subproblems by size and solve smaller ones first. **Runtime**:

Algorithm 22 Bottom-Up Rod Cutting

```
1:  procedure BOTTOM-UP-CUT-ROD(p, n)
2:      Let r[0...n] be a new array
3:      r[0] = 0
4:      for j = 1 to n do                          ▷ Solve for length j
5:          q = −∞
6:          for i = 1 to j do                      ▷ Try all first cuts i
7:              q = max(q, p[i] + r[j − i])
8:          end for
9:          r[j] = q
10:     end for
11:     return r[n]
12: end procedure
```

$\Theta(n^2)$ (due to nested loops).

Top-Down with Memoization Write the recursive procedure but check a table before computing. If the value is known, return it; otherwise, compute and save it. **Runtime**: $\Theta(n^2)$.

11.4.3 Reconstructing the Solution

The above algorithms return the max revenue but not the cuts. To find the cuts, we store the optimal choice $s[j]$ (the cut size i that gave the max) for each subproblem size j.

Algorithm 23 Extended Bottom-Up Rod Cutting

```
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                          ▷ Store optimal first cut length
10:            end if
11:        end for
12:        r[j] = q
13:    end for
14:    return r and s
15: end procedure
```

Algorithm 24 Print Solution

```
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]
5:         n = n − s[n]
6:     end while
7: end procedure
```

12 Greedy Algorithms

12.1 Introduction

A **Greedy Algorithm** makes locally optimal choices at each step with the hope of finding a globally optimal solution.

- **Pros**: Simple, efficient (often linear or $O(n \lg n)$).
- **Cons**: Does not always work (depends on the problem structure).

12.2 Activity Selection Problem

Problem: Given a set of activities $S = \{a_1, \ldots, a_n\}$ with start times s_i and finish times f_i, select a maximum-size subset of mutually compatible activities. Two activities are compatible if their intervals $[s_i, f_i)$ do not overlap.

12.2.1 Greedy Strategy

Sort activities by finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

1. Select the first activity a_1 (earliest finish time).
2. Remove all incompatible activities (those starting before a_1 finishes).
3. Repeat for the remaining activities.

Algorithm 25 Greedy Activity Selector

```
1: procedure GREEDY-ACTIVITY-SELECTOR(s, f)
2:     n = s.length
3:     A = {a_1}
4:     k = 1
5:     for m = 2 to n do
6:         if s[m] ≥ f[k] then                ▷ If compatible
7:             A = A ∪ {a_m}
8:             k = m
9:         end if
10:    end for
11:    return A
12: end procedure
```

12.2.2 Correctness

We must prove that the greedy choice (picking the activity with the earliest finish time) is safe.

THEOREM 12.1. *Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.*

Proof. Let A_k be a maximum-size subset of compatible activities. Let a_j be the first activity in A_k. Since a_m has the earliest finish time, $f_m \leq f_j$. If $a_j = a_m$, we are done. If not, replace a_j with a_m in A_k. The new set $A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}$ is still compatible (since $f_m \leq f_j$) and has the same size. Thus, there exists an optimal solution containing a_m. □

12.3 Knapsack Problems

12.3.1 0-1 Knapsack Problem

Problem: A thief has a knapsack with capacity W. There are n items, item i has value v_i and weight w_i. For each item, the thief can either take it (1) or leave it (0). Maximize total value $\sum v_i x_i$ subject to $\sum w_i x_i \leq W$.

Greedy Strategy Fails: Picking items with highest value-to-weight ratio v_i/w_i does NOT work. **Example**: $W = 50$. Items: A (60, 10), B (100, 20), C (120, 30).

- Ratios: A=6, B=5, C=4.
- Greedy picks A (weight 10, value 60). Remaining capacity 40. Picks B (weight 20, value 100). Remaining 20. Cannot pick C. Total value = 160.
- Optimal: Pick B and C. Total weight 50, Total value 220.

Solution: Use Dynamic Programming.

12.3.2 Fractional Knapsack Problem

Problem: Same as above, but the thief can take fractions of items (e.g., $x_i \in [0, 1]$).

Greedy Strategy Works:

1. Compute value-to-weight ratio v_i/w_i for all items.
2. Sort items by decreasing ratio.
3. Take as much as possible of the best item. If the knapsack fills up, take a fraction of the current item.

Runtime: $O(n \lg n)$ due to sorting.

12.4 Summary

- **Greedy Choice Property**: A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- **Optimal Substructure**: An optimal solution to the problem contains within it optimal solutions to subproblems.
- Greedy algorithms are generally faster than Dynamic Programming but harder to prove correct.

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 26 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 27 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 Lecture 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 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
```

Algorithm 29 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 30 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 31 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).

Index