Advanced Algorithms And Parallel Programming

Christian Rossi

Academic Year 2024-2025

Abstract

This course begins with an exploration of randomized algorithms, specifically Las Vegas and Monte Carlo algorithms, and the methods used to analyze them. We will tackle the hiring problem and the generation of random permutations to build a strong foundation. The course will then cover randomized quicksort, examining both worst-case and average-case analyses to provide a comprehensive understanding. Karger-s Min-Cut Algorithm will be studied, along with its faster version developed by Karger and Stein. We will delve into randomized data structures, focusing on skip lists and treaps, to understand their construction and application. Dynamic programming will be a key area, where we will learn about memoization and examine examples such as string matching and Binary Decision Diagrams (BDDs). The course will also introduce amortized analysis, covering dynamic tables, the aggregate method, the accounting method, and the potential method to equip students with robust analytical tools. Additionally, we will touch on approximate programming, providing an overview of this important concept. Finally, the competitive analysis will be explored through self-organizing lists and the move-to-front heuristic.

The second part of the course shifts to the design of parallel algorithms and parallel programming. We will study various parallel patterns, including Map, Reduce, Scan, MapReduce, and Kernel Fusion, to understand their implementation and application. Tools and languages essential for parallel programming, such as Posix Threads, OpenMP, and Message Passing Interface, will be covered, alongside a comparison of these parallel programming technologies. The course will also focus on optimizing and analyzing parallel performance, providing students with the skills needed to enhance and evaluate parallel computing systems. Practical examples of parallel algorithms will be reviewed to solidify understanding and demonstrate real-world applications.

Contents

CHAPTER 1

Algorithms analysis

1.1 Introduction

Definition (*Algorithm*)**.** An algorithm is a well-defined computational procedure that accepts one or more input values and produces one or more output values.

The problem statement outlines the desired relationship between input and output in broad terms, while the algorithm provides a detailed procedure to achieve that relationship. It is essential that an algorithm terminates after a finite number of steps.

1.2 Complexity analysis

The running time of an algorithm varies with the input. Therefore, we often parameterize running time by the input size.

Running time analysis can be categorized into three main types:

- *Worst-case*: here, $T(n)$ represents the maximum time an algorithm takes on any input of size n. This is particularly relevant when time is a critical factor.
- *Average-case*: in this case $T(n)$ reflects the expected time of the algorithm across all inputs of size n. It requires assumptions about the statistical distribution of inputs.
- *Best-case*: this scenario highlights a slow algorithm that performs well on specific inputs.

To establish a general measure of complexity, we focus on a machine-independent evaluation. This framework is called asymptotic analysis.

As the input length n increases, algorithms with lower complexity will outperform those with higher complexities. However, asymptotically slower algorithms should not be dismissed, as real-world design often requires a careful balance of various engineering objectives.

In mathematical terms, we define the complexitybound as:

$$\Theta\left(g(n)\right) = f(n)$$

Here, $f(n)$ satisfies the existence of positive constants c_1, c_2, and n_0 such that:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \forall n \geq n_0$$

In engineering practice, we typically ignore lower-order terms and constants.

Example:
Consider the following expression:

$$3n^3 + 90n^2 - 5n + 6046$$

The corresponding theta notation is:

$$\Theta(n^3)$$

Given $c > 0$ and $n_0 > 0$, we can define other bounds notations:

Bound type	**Notation**	**Condition**
Upper bound	$\mathcal{O}(g(n)) = f(n)$	$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$
Lower bound	$\Omega(g(n)) = f(n)$	$0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0$
Strict upper bound	$o(g(n)) = f(n)$	$0 \leq f(n) < cg(n) \quad \forall n \geq n_0$
Strict lower bound	$\omega(g(n)) = f(n)$	$0 \leq cg(n) < f(n) \quad \forall n \geq n_0$

Example:
For the expression $2n^2$:

$$2n^2 \in O(n^3)$$

For the expression $\sqrt{n}$:

$$\sqrt{n} \in \Omega(\ln(n))$$

From this, we can redefine the average bound as:

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

1.2.1 Sorting problem

The sorting problem involves taking an array of numbers $\langle a_1, a_2, \dots, a_n \rangle$ and returning the permutation of the input $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Example:
Given an array:

$$\langle 8, 2, 4, 9, 3, 6 \rangle$$

The sorted version will be:

$$\langle 2, 3, 4, 6, 8, 9 \rangle$$

Algorithm 1 Insertion sort

for $j = 2$ **to** n **do**
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ **and** $A[i] > key$ **do**
        $A[i+1] = A[i]$
        $i = i - 1$
    end while
    $A[i+1] = key$
end for

The complexities for the insertion sort are:

Case	Complexity	Notes
Worst	$T(n) = \Theta(n^2)$	Input in reverse order
Average	$T(n) = \Theta(n^2)$	All permutations equally likely
Best	$T(n) = \Theta(n)$	Already sorted

In conclusion, while this algorithm performs well for small n, it becomes inefficient for larger input sizes.

A recursive solution for the sorting problem could be implemented with the merge sort.

Algorithm 2 Merge sort

if $n = 1$ **then**
    return $A[n]$
end if
Recursively sort the two half lists $A\left[1 \dots \left\lceil \frac{n}{2} \right\rceil\right]$ and $A\left[\left\lceil \frac{n}{2} \right\rceil + 1 \dots n\right]$
Merge $\left(A\left[1 \dots \left\lceil \frac{n}{2} \right\rceil\right], A\left[\left\lceil \frac{n}{2} \right\rceil + 1 \dots n\right]\right)$

The merge operation makes this algorithm recursive. To analyze its complexity, we consider the following components:

- When the array has only one element, the complexity is constant: $\Theta(1)$.
- The recursive sorting of the two halves contributes a total cost of $2T\left(\frac{n}{2}\right)$.
- The merging of the two sorted lists requires linear time to check all elements, yielding a complexity of $\Theta(n)$.

Thus, the overall complexity for merge sort can be expressed as:

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

For sufficiently small n, the base case $\Theta(1)$ can be omitted if it does not affect the asymptotic solution. The solution for the recurrence equation is:

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

1.3 Recurrences

To determine the complexity a recurrent algorithm, we need to solve the equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$$

To solve this recurrence we may use three different tecniques:

1. Recursion tree.
2. Substitution method.
3. Masther method.

1.3.1 Recursion tree

In the recursion tree we expand nodes until we reach the base case.

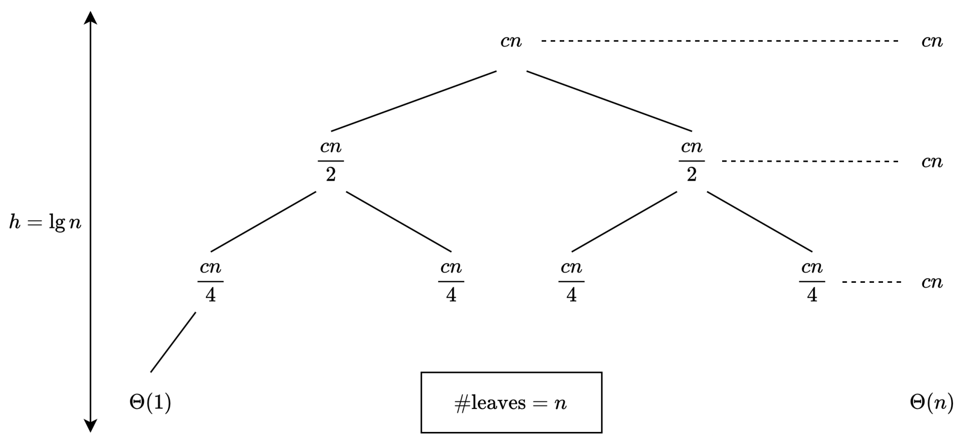


Figure 1.1: Partial recursion tree for merge sort algorithm

The depth of the tree is $h = \log n$, and the total number of leaves is n. Thus, the complexity can be computed as:

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

The merge sort outperforms insertion sort in the worst case, but in practice merge sort generally surpasses insertion sort for $n > 30$.

1.3.2 Substitution method

The substitution method is a general technique for solving recursive complexity equations. The steps are as follows:

1. Guess the form of the solution based on preliminary analysis of the algorithm.
2. Verify the guess by induction.
3. Solve for any constants involved.

Example:
Consider the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Assuming the base case $T(1) = \Theta(1)$, we can apply the substitution method:

1. Guess a solution of $\mathcal{O}(n^3)$, so we assume $T(k) \leq ck^3$ for $k < n$.
2. Verify by induction that $T(n) \leq cn^3$.

This approach, while effective, may not always be straightforward.

1.3.3 Master method

To simplify the analysis, we can use the master method, applicable to recurrences of the form:

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

Here, $a \geq 1$, $b > 1$, and $f(n)$ is asymptotically positive. While less general than the substitution method, it is more straightforward.

To apply the master method, compare $f(n)$ with $n^{\log_b a}$. There are three possible outcomes:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then
$$T(n) = \Theta(n^{\log_b a})$$
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then:
$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $f(n)$ satisfies the regularity condition $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some constant $0 < c < 1$, then:
$$T(n) = \Theta(f(n))$$

Example:
Let's analyze the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

In this case, we have $a = 4$ and $b = 2$, which gives us:

$$n^{\log_b a} = n^2 \qquad f(n) = n$$

Here, we find ourselves in the first case of the master theorem, where $f(n) = \mathcal{O}(n^{2-\varepsilon})$ for $\varepsilon = 1$. Thus, the solution is:

$$T(n) = \Theta(n^2)$$

Now consider the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

Again, we have $a = 4$ and $b = 2$, leading to:

$$n^{\log_b a} = n^2 \qquad f(n) = n^2$$

In this scenario, we are in the second case of the theorem, where $f(n) = \Theta(n^2 \log^k n)$ for $k = 0$. Therefore, the solution is:

$$T(n) = \Theta(n^2 \log n)$$

Next, consider:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

With $a = 4$ and $b = 2$, we find:

$$n^{\log_b a} = n^2 \qquad f(n) = n^3$$

Here, we fall into the third case of the theorem, where $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$. Thus, the solution is:

$$T(n) = \Theta(n^3)$$

Finally, consider the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$$

Again, we have $a = 4$ and $b = 2$ yielding:

$$n^{\log_b a} = n^2 \qquad f(n) = \frac{n^2}{\log n}$$

In this case, the master method does not apply. Specifically, for any constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\log n)$, indicating that the conditions for the theorem are not satisfied.

CHAPTER 2

Advanced algorithms

2.1 Divide and conquer algorithms

The divide and conquer design paradigm consists of three key steps:

1. Divide the problem into smaller sub-problems.
2. Conquer the sub-problems by solving them recursively.
3. Combine the solutions of the sub-problems.

This approach enables us to tackle larger problems by breaking them down into smaller, more manageable pieces, often resulting in faster overall solutions.

The divide step is typically constant, as it involves splitting an array into two equal parts. The time required for the conquer step depends on the specific algorithm being analyzed. Similarly, the combine step can either be constant or require additional time, again depending on the algorithm.

Merge sort The merge sort algorithm, previously discussed, follows these steps:

- *Divide*: the array is split into two sub-arrays.
- *Conquer*: each of the two sub-arrays is sorted recursively.
- *Combine*: the two sorted sub-arrays are merged in linear time.

The recursive expression for the complexity of merge sort can be expressed as follows:

$$T(n) = \underbrace{2}_{\text{\#subproblems}} \underbrace{T\left(\frac{n}{2}\right)}_{\text{subproblem size}} + \underbrace{\Theta(n)}_{\text{work dividing and combining}}$$

2.1.1 Binary search

The binary search problem involves locating an element within a sorted array. This can be efficiently solved using the divide and conquer approach, outlined as follows:

1. *Divide*: check the middle element of the array.
2. *Conquer*: recursively search within one of the sub-arrays.
3. *Combine*: if the element is found, return its index in the array.

In this scenario, we only have one sub-problem, which is the new sub-array, and its length is half that of the original array. Both the divide and combine steps have a constant complexity.

Thus, the final expression for the complexity is:

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

By applying the master method, we find a final complexity of:

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

2.1.2 Power of a number

The problem at hand is to compute the value of a^n, where $n \in \mathbb{N}$. The naive approach involves multiplying a by itself n times, resulting in a total complexity of $\Theta(n)$.

We can also use a divide and conquer algorithm to solve this problem by dividing the exponent by two, as follows:

$$a^n = \begin{cases} a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} & \text{if } n \text{ is even} \\ a^{\frac{n-1}{2}} \cdot a^{\frac{n-1}{2}} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

In this approach, both the divide and combine phases have a constant complexity, as they involve a single division and a single multiplication, respectively. Each iteration reduces the problem size by half, and we solve one sub-problem (with two equal parts).

Thus, the recurrence relation for the complexity is:

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

By applying the master method, we find a final complexity of:

$$\Theta(\log n)$$

2.1.3 Matrix multiplication

Matrix multiplication involves taking two matrices A and B as input and producing a resulting matrix C, which is their product. Each element of the matrix C is computed as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

The standard algorithm for matrix multiplication is outlined below:

Algorithm 3 Standard matrix multiplication

```
for i = 1 to n do
    for j = 1 to n do
        c_ij = 0
        for k = 1 to n do
            c_ij = c_ij + a_ik b_kj
        end for
    end for
end for
```

The complexity of this algorithm, due to the three nested loops, is $\Theta(n^3)$.

Divide and conquer For the divide and conquer approach, we divide the original $n \times n$ matrix into four $\frac{n}{2} \times \frac{n}{2}$ sumatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

This requires solving the following system:

$$\begin{cases} r = ae + bg \\ s = af + bh \\ t = ce + dg \\ u = cf + dh \end{cases}$$

This results in a total of eight multiplications and four additions of the submatrices. The recursive part of the algorithm involves the matrix multiplications. The time complexity can be expressed as $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$. Using the master method, we find that the total complexity remains $\Theta(n^3)$.

Strassen To improve efficiency, Strassen proposed a method that reduces the number of multiplications from eight to seven matrices. This approach requires seven multiplications and a total of eighteen additions and subtractions.

The divide and conquer steps are as follows:

1. *Divide*: partition matrices A and B into $\frac{n}{2} \times \frac{n}{2}$ submatrices and formulate terms for multiplication using addition and subtraction.

2. *Conquer*: recursively perform seven multiplications of $\frac{n}{2} \times \frac{n}{2}$ submatrices.

3. *Combine*: construct matrix C using additions and subtractions on the $\frac{n}{2} \times \frac{n}{2}$ sumbatrices.

The recurrence relation for the complexity is: $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$ By solving this recurrence with the master method, we obtain a complexity of:

$$\Theta\left(n^{\log_2 7}\right) \approx \Theta\left(n^{2.81}\right)$$

Although 2.81 may not seem significantly smaller than 3, the impact of this reduction in the exponent is substantial in terms of running time. In practice, Strassen's algorithm outperforms the standard algorithm for $n \geq 32$.

The best theoretical complexity achieved so far is $\Theta\left(n^{2.37}\right)$, although this remains of theoretical interest, as no practical algorithm currently achieves this efficiency.

2.1.4 VLSI layout

The problem involves embedding a complete binary tree with n leaves into a grid while minimizing the area used.

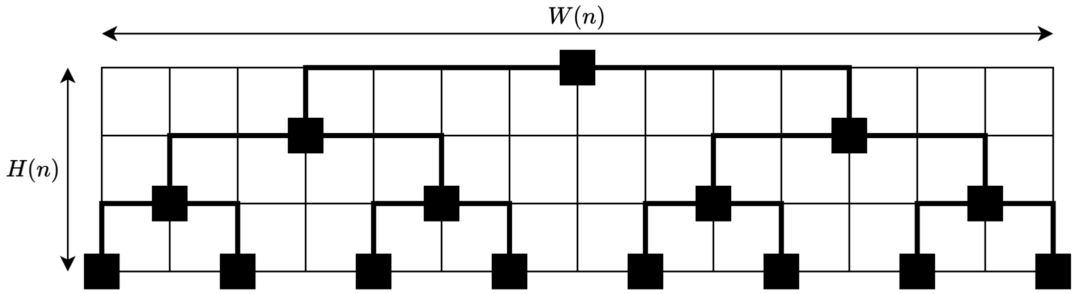


Figure 2.1: VLSI layout problem

For a complete binary tree, the height is given by:

$$H(n) = H\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\log_2 n)$$

The width is expressed as:

$$W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1) = \Theta(n)$$

Thus, the total area of the grid required is:

$$\text{Area} = H(n) \cdot W(n) = \Theta(n \log_2 n)$$

H-tree An alternative solution to this problem is to use an h-tree instead of a binary tree.

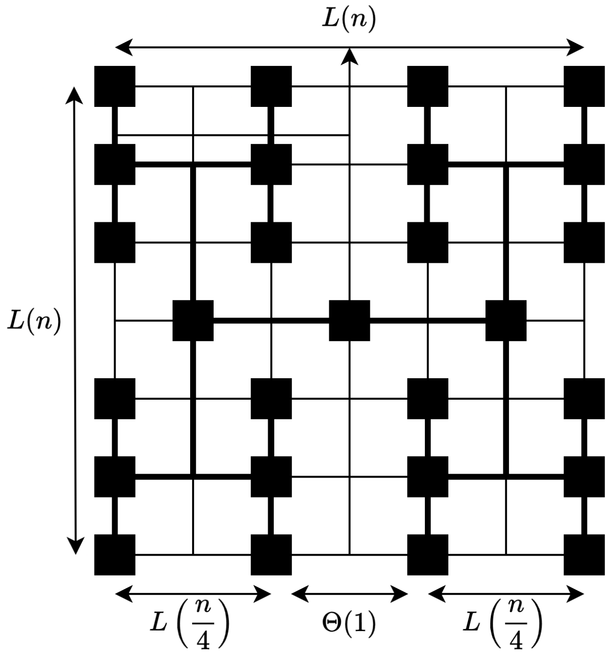


Figure 2.2: VLSI layout problem

For the h-tree, the length is given by:

$$L(n) = 2L\left(\frac{n}{4}\right) + \Theta(1) = \Theta(\sqrt{n})$$

Consequently, the total area required for the h-tree is computed as:

$$\text{Area} = L(n)^2 = \Theta(n)$$

2.2 Dynamic Programming algorithms

Dynamic Programming is a problem-solving technique first introduced in the 1940s by Richard Bellman. It focuses on finding optimal solutions by breaking problems down into overlapping subproblems, solving each once, and storing the results to avoid redundant calculations. The term dynamic highlights the approach's adaptability to the time-varying nature of certain problems, while programming originally referred to developing an optimized sequence of decisions, akin to a military plan or schedule for logistics or training.

2.2.1 Longest Common Subsequence

Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, the objective of the Longest Common Subsequence (LCS) problem is to find the longest sequence of elements that appears in both x and y in the same relative order. A subsequence is derived by deleting some (or none) of the elements from a sequence without changing the order of the remaining elements. Thus, the LCS is the longest such sequence common to both x and y.

2.2.1.1 Naive algorithm

A naive approach to solving the LCS problem involves generating all possible subsequences of x and checking each one to see if it is also a subsequence of y. The steps for this brute-force algorithm are as follows:

1. *Generate all subsequences of* x: for a sequence of length m, there are 2^m possible subsequences. Each subsequence corresponds to a unique bit vector of length $m$where each bit indicates whether the corresponding element in x is included in that subsequence.

2. *Check each subsequence in* y: for each subsequence of x, verify if it also appears as a subsequence in y. This can be done by iterating over y and confirming that the elements of the subsequence appear in the same order within y.

3. *Track the LCS*: while iterating through all possible subsequences, keep a record of the longest one that is a subsequence of both x and y. Once all subsequences have been checked, the LCS found is the solution.

The naive approach is highly inefficient due to its exponential time complexity. With m elements in x, there are 2^m possible subsequences, as each element has the option to be included or excluded. Checking if a subsequence of x is also a subsequence of y takes $\mathcal{O}(n)$ time. Therefore, the total time complexity for the naive algorithm is:

$$T(n) = \mathcal{O}(n2^m)$$

2.2.1.2 Recursive algorithm

To solve the LCS problem using a recursive approach, we break it down into two main steps:

1. *Compute the length of the LCS*: define a recursive function that calculates the length of the LCS between prefixes of two sequences, x and y.

2. *Extend to contruct the LCS itself*: moodify the function to also record subsequence characters, allowing reconstruction of the LCS.

We define a two-dimensional array $c[i, j] = |\text{LCS}(x[1 \dots i], y[1 \dots j])|$, where each cell $c[i, j]$ represents the length of the LCS between the prefixes $x[1 \dots i]$ and $y[1 \dots j]$. The value at $c[m, n]$ then gives the length of the LCS for the entire sequences x and y.

Theorem 2.2.1. *The recursive relation for $c[i, j]$ is given by:*

$$\begin{cases} c[i-1, j-1] + 1 & \textit{if } x[i] = y[j] \\ \max\{c[i-1, j], c[i, j-1]\} & \textit{otherwise} \end{cases}$$

This recursive approach leverages the optimal substructure property of the LCS, meaning that the optimal solution to the LCS problem for sequences x and y depends on optimal solutions to subproblems involving prefixes of these sequences.

Definition (*Optimal substructure*)**.** An optimal solution to a problem instance includes optimal solutions to its subproblems.

Algorithm 4 Recursive LCS

```
function LCS(x, y, i, j)
    if x[i] = y[j] then
        c[i, j] = LCS(x, y, i − 1, j − 1) + 1
    else
        c[i, j] = max{LCS(x, y, i − 1, j), LCS(x, y, i, j − 1)}
    end if
    return c[i, j]
end function
```

In the worst-case scenario, when $x[i] \neq y[j]$, the algorithm recursively evaluates two subproblems for each pair (i, j), leading to an exponential time complexity of $\mathcal{O}(2^{m+n})$. Since many subproblems are recalculated multiple times, this recursive approach can be highly inefficient without further optimization.

Memoization To eliminate redundant calculations, we use memoization, storing results of previously solved subproblems in a table. When the algorithm encounters a subproblem it has solved before, it retrieves the stored result instead of recalculating it.

Definition (*Memoization*)**.** Memoization is a technique where, after computing the solution to a subproblem, we store it in a table so that future calls can retrieve the result directly, avoiding redundant work.

Algorithm 5 Memoized recursive LCS

```
procedure LCS(x, y, i, j)
    if c[i, j] = null then
        if x[i] = y[j] then
            c[i, j] = LCS(x, y, i − 1, j − 1) + 1
        else
            c[i, j] = max[LCS(x, y, i − 1, j),LCS(x, y, i, j − 1) ]
        end if
    end if
end procedure
```

By storing each subproblem's solution only once, the memoization approach reduces the time complexity to $\Theta(mn)$, where m and n are the lengths of sequences x and y. This is because we compute each of the mn subproblems only once. he space complexity is also $\Theta(mn)$, as we store each subproblem result in a table.

2.2.1.3 Dynamic Programming

The Dynamic Programming approach to solving the LCS problem avoids recursion by systematically building the solution from the smallest subproblems up to the full problem. This bottom-up approach is efficient in both time and space, eliminating the overhead of recursive calls.

The steps involved in the DP solution are as follows:

1. *Construct the table*: create a table c where $c[i, j]$ holds the length of the LCS for prefixes $x[1 \dots i]$ and $y[1 \dots j]$. Initialize $c[0, 0]$ and fill the table up to $c[m, n]$ based on the recurrence relation used in the recursive solution.

2. *Reconstruct the LCS*: after computing $c[m, n]$, use the table to backtrack from $c[m, n]$ to $c[0, 0]$, tracing the characters that contribute to the LCS.

Algorithm 6 Dynamic Programming LCS

```
procedure LCS(x, y)
    Initialize c[0 . . . m, 0 . . . n] to 0
    for i = 1 to m do
        for j = 1 to n do
            if x[i] = y[j] then
                c[i, j] = c[i − 1, j − 1] + 1
            else
                c[i, j] = max(c[i − 1, j], c[i, j − 1])
            end if
        end for
    end for
    return c[m, n]
end procedure
```

This approach has a time complexity of $\mathcal{O}(mn)$ as each entry of the $m \times n$ table is filled once. The space complexity is also $\mathcal{O}(mn)$, since we store each subproblem result in the table. This

Dynamic Programming method is both efficient and avoids redundant calculations, making it well-suited for large inputs.

2.2.2 Binary Decision Diagram

Binary Decision Diagrams (BDDs) are a compact data structure for representing Boolean functions. They improve on traditional approaches by storing evaluated sub-cases in memory, allowing for efficient retrieval and manipulation of Boolean expressions. BDDs are particularly valuable because they can be made canonical, which provides a unique representation for each Boolean function with a fixed variable ordering, enabling easier comparison and optimization.

One of the key benefits of BDDs is their efficiency in performing Boolean operations, such as conjunction, disjunction, and negation. However, the size of a BDD heavily depends on the chosen variable ordering; optimal ordering can significantly reduce the size and complexity of the BDD.

In a BDD, a Boolean function is represented as a Directed Acyclic Graph (DAG) with: single root node representing the function's entry point and two terminal nodes labeled 0 and 1, representing the function's output values. Each internal node is associated with a variable and has exactly two children, representing the function's behavior when that variable is assigned 1 or 0.

Ordered BDD An Ordered BDD (OBDD) applies only the ordering constraint, requiring that variables appear in a specific sequence along any path but without enforcing reduction.

Reduced Ordered BDD An ROBDD is a special type of BDD that is both reduced and ordered, resulting in a compact, canonical representation for Boolean functions. An ROBDD is derived from a Shannon co-factoring tree with two main modifications:

- *Reduction*: this process eliminates redundancy by:
  - Removing nodes with identical children.
  - Merging nodes with identical subgraphs, so that each unique subgraph appears only once in the diagram.
- *Ordering*: variables are processed in a fixed, consistent order along every path from the root to a terminal node. This ordering ensures that each path encounters variables in a predefined sequence, contributing to the canonical nature of ROBDDs.

The canonical structure of an ROBDD allows straightforward identification of the function's onset (the set of variable assignments that make the function evaluate to 1). This can be achieved by tracing all paths from the root to the 1-terminal node. Each such path represents a set of variable assignments that satisfy the Boolean function.

2.2.2.1 Implementation

The efficient implementation of BDDs relies on specific data structures that prevent redundancy, streamline Boolean operations, and optimize memory usage. The key components are:

- *Unique table*: ensures that each node in the BDD is unique, preventing duplication. This is typically implemented as a hash table, where the properties of each node serve as a key mapping to a unique existing or newly created node.

- *Computed table*: stores the results of previously computed operations to speed up repeated calculations. This table, which can be implemented as a hash table, indexes operations in the form (f, g, h), representing arguments passed to the ITE operator. If a result for (f, g, h) exists in the table, it can be retrieved without recomputation.

ITE operator The ITE operator is central to BDD operations, as it can implement any binary Boolean function. Mathematically, the ITE operator is defined as:

$$\text{ite}(f, g, h) = fg + f\bar{h}$$

Before adding a new node (v, g, h) is added to the BDD, the unique table is checked to avoid duplication. If a matching node exists, its pointer is reused; otherwise, a new node is created and added to the unique table.

Algorithm 7 Recursive ITE

```
function ITE(f, g, h)
    if f = 1 then
        return g
    end if
    if f = 0 then
        return h
    end if
    if g = h then
        return g
    end if
    if p = HashLookupComputedTable(f, g, h) then
        return p
    end if
    v = TopVariable(f, g, h)
    f_n = ITE(f_v, g_v, h_v)
    g_n = ITE(f_v̄, g_v̄, h_v̄)
    if f_n = g_n then
        return g_n
    end if
    if !(p = HashLookupComputedTable(v, f_n, g_n)) then
        p = CreateNode(v, f_n, g_n)
        Insert p into the unique table
    end if
    key = HashKey(f, g, h)
    InsertComputedTable(p, key)
    return p
end function
```

Complemented edges Complemented edges optimize memory and operation efficiency by allowing the BDD to represent negated functions without additional nodes. This technique reduces storage needs and accelerates NOT and ITE operations. To maintain the canonical form of the BDD, certain edge equivalences are managed, ensuring consistent representation across nodes.

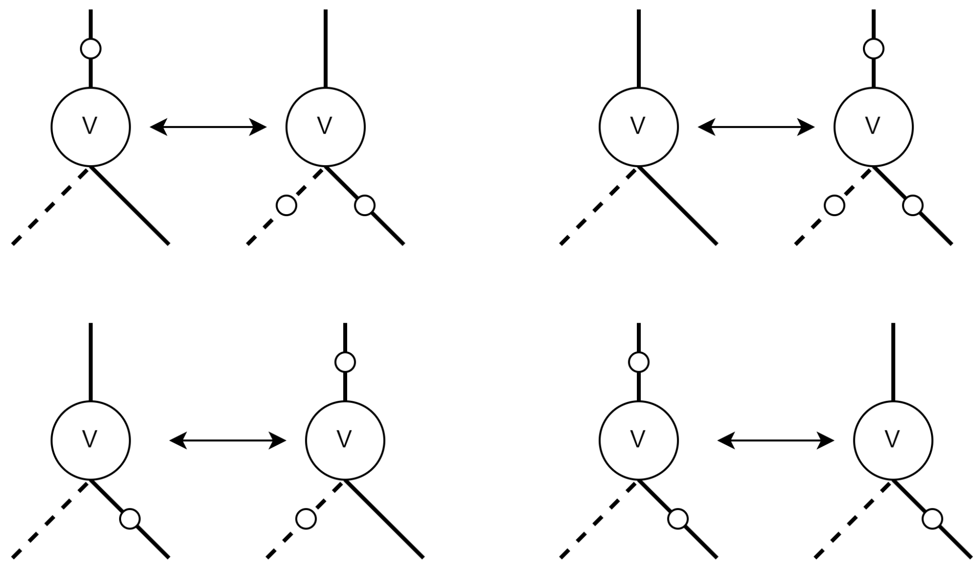


Figure 2.3: Edge equivalence

Optimization To maximize matches in the computed table, conventions are applied, such as selecting the argument with the smallest top variable (or smallest pointer in case of a tie) as the first argument. Caching mechanisms—including local caching, operation (specific caching, and shared caching) also contribute to efficient BDD operations.

2.2.2.2 Garbage collection

Effective garbage collection is essential for managing memory usage in BDDs, as unreferenced nodes consume resources and degrade performance. Garbage collection in BDDs typically involves:

1. *Reference counting*: each node tracks the number of references to it, allowing immediate deallocation when the reference count drops to zero.

2. *Mark-and-sweep*: this process involves marking nodes with references during a traversal and then sweeping through to delete unmarked nodes in a second pass.

Garbage collection timing is important to optimize resource usage without excessive overhead. Nodes can be deallocated on demand, at specific intervals, or after predefined operations. Since computed tables do not track references, they are cleared separately during the garbage collection process to maintain efficiency.

2.3 Randomized algorithms

Probabilistic analysis in algorithms assumes that the algorithm itself is deterministic; for a given fixed input, it will produce the same output and follow the same sequence of operations every time it runs. This analysis model considers a probability distribution over the possible inputs, evaluating the algorithm's performance based on this distribution. While this can provide useful insights into average-case behavior, it has limitations. or instance, certain specific inputs may lead to particularly poor performance, and if the assumed distribution does not accurately represent real-world inputs, the analysis may yield a misleading or overly optimistic view of the algorithm's expected efficiency.

In contrast, randomized algorithms incorporate randomness into their execution process, introducing variability in their behavior even when given a fixed input. Due to this inherent randomness, a randomized algorithm may produce different results or follow different execution paths on the same input in separate runs. Generally, randomized algorithms are designed to perform well with high probability across any input, though there remains a small probability of failure on any given run.

2.3.1 Taxonomy

	Las Vegas	**Monte Carlo**
Randomness effect	Running time	Running time Solution correctness
Efficiency (polynomial bound)	Expected running time	Worst-case running time

Monte Carlo algorithms are classified further based on their error probabilities. A Monte Carlo algorithm with two-sided error has a nonzero probability of error for both possible outputs. In contrast, a one-sided error algorithm guarantees correctness for at least one of the outputs, meaning it has zero error probability for that output.

2.4 Minimum cut problem

Let $G = (V, E)$ be a connected, undirected graph, where $n = |V|$ and $m = |E|$ represent the number of vertices and edges, respectively. For any subset $S \subset V$, the set $\delta(S) = \{(u, v) \in E \mid u \in S, v \in S'\}$ defines a cut, separating the vertices in S from those in $S' = V \setminus S$. The minimum cut problem seeks to identify a cut with the fewest edges connecting S and S', effectively partitioning the graph with minimal separation.

2.4.1 Naive algorithm

A traditional approach to solving the minimum cut problem is to compute $n - 1$ minimum source-target cuts, one for each possible pair of vertices. A source-target cut partitions the graph into two disjoint sets such that one subset contains a designated source vertex s and the other contains a designated target vertex t.

The size of the minimum source-target cut is equivalent to the maximum flow between s and t. The most efficient algorithm known for the maximum flow problem has a time complexity of:

$$T(n) = \mathcal{O}\left(nm \log\left(\frac{n^2}{m}\right)\right)$$

Here, n is the number of vertices and m is the number of edges.

2.4.2 Karger's algorithm

Karger introduced a randomized algorithm that avoids explicit maximum flow calculations by using edge contraction to iteratively simplify the graph. This contraction process preserves the minimum cut with high probability, resulting in a more efficient approach.

Edge contraction involves merging two vertices connected by an edge, $e = (u, v)$, into a single new vertex w. The contraction replaces all edges incident to u or v with edges incident to w, while removing any self-loops created by this merging process.

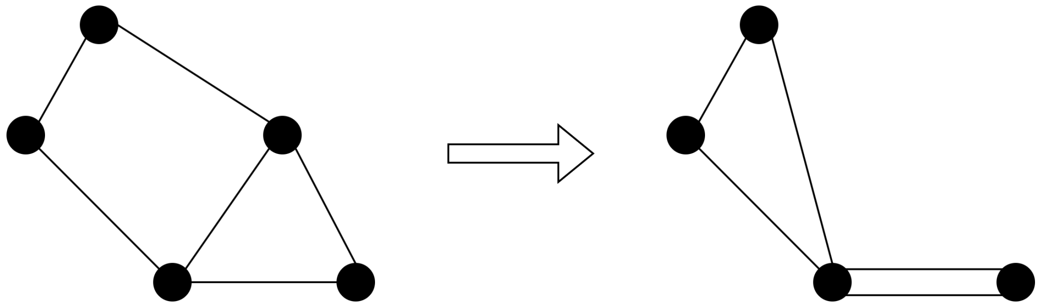


Figure 2.4: Edge contraction

Definition (*Edge contraction*)**.** For a multigraph $G = (V, E)$ without self-loops, contracting an edge $e = \{u, v\} \in E$, denoted $G \setminus e$, results in:

1. Replacing vertices u and v with a new vertex w.
2. Redirecting all edges incident to u and v to w.
3. Removing any self-loops involving w.

After contraction, the graph $G \setminus e$ remains a multigraph. Importantly, contracting (u, v) does not affect cuts where u and v are both in the same set.

Algorithm Karger's algorithm for the minimum cut works as follows:

1. Select an edge uniformly at random and contract its endpoints.
2. Repeat the contraction process until only two vertices remain.

The two remaining vertices form a partition (S, S') of the original graph, where the edges connecting S and S' defines the cut $\delta(S)$ in G.

Lemma 2.4.1. *For a minimum cut $\delta(S)$ in the graph $G = (V, E)$, Karger's algorithm produces this minimum cut with probability at least:*

$$\Pr(\text{minimum cut}) \geq \frac{1}{\binom{n}{2}}$$

To increase the success probability, we repeat Karger's algorithm $l \cdot \binom{n}{2}$ times. The probability that at least one run will successfully produce the minimum cut is:

$$\Pr(\text{one success}) \geq 1 - e^{l}$$

Setting $l = c \log n$ reduces the error probability to less than:

$$\Pr(\text{error}) \leq \frac{1}{n^c}$$

Complexity One run of Karger's algorithm takes $\mathcal{O}(n^2)$ time. By repeating the algorithm $\mathcal{O}(n^2 \log n)$ times, we obtain a randomized algorithm with total time complexity:

$$T(n) = \mathcal{O}(n^4 \log n)$$

2.4.3 Karger and Stein algorithm

Karger and Stein refined Karger's original minimum cut algorithm to improve efficiency by enhancing the edge contraction process. The core insight lies in understanding the telescoping product that emerges when calculating the probability of preserving edges in the minimum cut set $\delta(S)$ during contractions.

In the early stages, it's unlikely that an edge from the minimum cut set is contracted. However, as the graph reduces in size, the probability of contracting such an edge increases. By focusing on the probability that a fixed minimum cut $\delta(S)$ survives contraction to a subgraph with l vertices, we find that:

$$\Pr(\text{cut survives}) = \frac{\binom{l}{2}}{\binom{n}{2}}$$

Setting $l = \frac{n}{\sqrt{2}}$ ensures a survival probability of at least $\frac{1}{2}$. This suggests that, on average, running two trials of the algorithm should be sufficient to find the minimum cut with high probability.

Algorithm The Karger-Stein algorithm proceeds as follows for a multigraph G with at least six vertices:

1. Run the edge contraction algorithm on $\frac{n}{\sqrt{2}} + 1$ vertices.
2. Recur on the resulting contracted graph.
3. Repeat these steps twice, then return the smaller of the two cuts found.

Notably, setting the recursion threshold to six vertices affects only the constant factor of the runtime, without impacting the asymptotic complexity.

Algorithm 8 Karger and Stein

```
function CONTRACT(G = (V, E), t)
    while |V| > t do
        Choose e ∉ E uniformly at random
        G = G \ e
    end while
    return G
end function

function FASTMINCUT(G = (V, E))
    if |V| < 6 then
        return mincut(V)
    else
        t = ⌈1 + |V|/√2⌉
        G_1 = CONTRACT(G, t)
        G_2 = CONTRACT(G, t)
        return min{FASTMINCUT(G_1), FASTMINCUT(G_2)}
    end if
end function
```

Complexity The recurrence relation for the running time of the Karger-Stein algorithm is:

$$T(n) = T\left(\frac{n}{\sqrt{2}}\right) + \Theta(n^2)$$

Which solves to a complexity of $\mathcal{O}(n^2 \log n)$.

The algorithm's success probability at each recursive step is at least $\geq \frac{1}{2}$. To increase the probability of finding the minimum cut, we repeat the algorithm multiple times. The probability of success is:

$$\Pr(\text{success}) = \Omega\left(\frac{1}{\log n}\right)$$

Thus, to ensure the algorithm succeeds with high probability we need to run the algorithm $\mathcal{O}(\log^2 n)$ times. Thus, the total time complexity of the Karger-Stein algorithm is:

$$T(n) = \mathcal{O}(n^2 \log^3 n)$$

Corollary 2.4.1.1. *Any graph has at most $\mathcal{O}(n^2)$ distinct minimum cuts.*

2.5 Sorting problem

The sorting problem is a fundamental computational task in which a collection of elements is arranged in a specific order, typically ascending or descending. Given an unsorted list or array, the goal is to rearrange the elements to follow a predefined sequence based on a chosen criterion, such as numerical or lexicographical order.

2.5.1 Quicksort

Quicksort, introduced by Hoare in 1962, is a highly efficient, in-place, divide-and-conquer sorting algorithm known for its practical performance across a variety of applications. By partitioning data around a pivot element, Quicksort can achieve efficient sorting with minimal extra storage, making it one of the most widely used sorting algorithms.

The Quicksort algorithm works as follows:

1. *Divide*: select a pivot element from the array, then partition the array into two subarrays. Elements less than or equal to the pivot form the left subarray. Elements greater than or equal to the pivot form the right subarray.
2. *Conquer*: recursively apply Quicksort to each of the two subarrays.
3. *Combine*: since the subarrays are sorted in place, no additional merging is needed.

The efficiency of Quicksort depends on the partitioning step, which operates in $\mathcal{O}(n)$ time.

Algorithm 9 Quicksort

```
function PARTITION(A, p, q)
    x = A[p]
    i = p
    for j = p + 1 to q do
        if A[j] ≤ x then
            i = i + 1
            exchange A[i] and A[j]
        end if
    end for
    exchange A[p] and A[i]
    return i
end function

procedure QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure
```

The performance of Quicksort varies based on the choice of pivot and the input data. Here are the primary cases:

- *Worst-case*: the pivot always ends up at one of the ends of the array, resulting in highly unbalanced partitions. This scenario, often due to already sorted or reverse-sorted data when a poor pivot is chosen, yields a time complexity of $\Theta(n^2)$.
- *Average case*: the pivot splits the array into reasonably balanced parts. This is achieved with a random or median pivot selection, leading to a time complexity of $\Theta(n \log n)$, which is efficient for large datasets.

- *Best case*: the pivot consistently splits the array into two equal halves, minimizing the depth of recursive calls. This optimal scenario also results in a time complexity of $\Theta(n \log n)$.

2.5.2 Randomized Quicksort

Randomized Quicksort is an improved variant of the classic Quicksort algorithm that selects a pivot randomly from the array, significantly reducing the chance of worst-case performance. By ensuring the pivot choice is independent of the input structure, Randomized Quicksort achieves efficient performance on average for various inputs.

The randomized selection of the pivot minimizes the probability of consistently poor partitions, where the pivot might otherwise split the array in highly unbalanced ways. With a randomized pivot, the expected time complexity becomes $\Theta(n \log n)$, independent of any initial ordering of the data.

Analysis Let X denote the running time of Randomized Quicksort on an input of size n, assuming that each pivot selection is independent and uniformly random. Define an indicator variable X_k for the event that a partition results in a split of k elements on one side and $n-k-1$ elements on the other:

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k \mid (n-k-1) \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

Since any element can be chosen as the pivot with equal probability, the expected value of X_k is:

$$\mathbb{E}[X_k] = \Pr(X_k = 1) = \frac{1}{n}$$

Thus, the expected running time $\mathbb{E}[T(n)]$ can be written in terms of the recursive costs of partitioning:

$$\mathbb{E}[T(n)] = \mathbb{E}\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

This simplifies to:

$$\mathbb{E}[T(n)] = \frac{2}{n}\sum_{k=1}^{n} \mathbb{E}[T(k)] + \Theta(n)$$

For sufficiently large $n \geq 2$, this recursive relation leads to:

$$\mathbb{E}[X] \leq \frac{2}{n}\sum_{k=1}^{n} ak \log k + \Theta(n) \leq an \log n$$

Thus, for a sufficiently large constant a, the $\Theta(n)$ term is dominated, leading to an overall time complexity of:

$$T(n) = \mathcal{O}(n \log n)$$

Practical performance In practice, Randomized Quicksort often outperforms Merge Sort, typically running at least twice as fast due to its efficient in-place operations and reduced memory usage. With careful code tuning and optimized implementations, Quicksort's performance can be further enhanced. Its contiguous memory access patterns make it highly cache-friendly, and it handles virtual memory effectively.

2.5.3 Comparison sort analysis

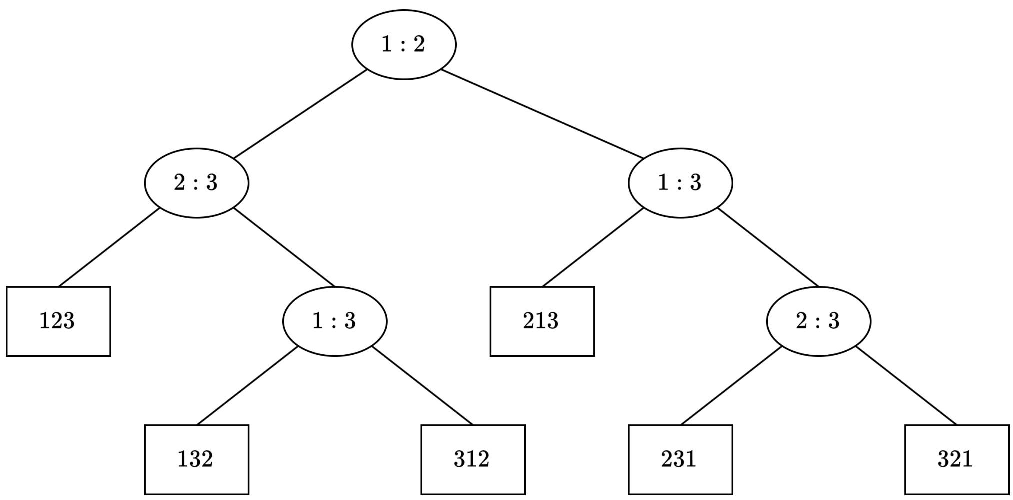
All the sorting algorithms discussed so far are comparison sorts, where element ordering is determined by comparing pairs of elements. The best worst-case time complexity for these algorithms is $\mathcal{O}(n \log n)$.

To understand this lower bound, consider sorting an array $\langle a_1, a_2, \ldots, a_n \rangle$. We can represent each sequence of comparisons made by a sorting algorithm as a path in a decision tree:

- Each internal node in this tree represents a comparison between two elements a_i and a_j.
- The left child of a node represents the branch taken if $a_i \leq a_j$, while the right child represents the branch if $a_i > a_j$.

Each leaf node in the decision tree corresponds to a unique, final sorted order (or permutation) of the array. Therefore, this tree models all possible sequences of comparisons that could occur for different input configurations.

Example:
Consider sorting the array $\langle 9, 4, 6 \rangle$. A decision tree for this array might look as follows, showing different comparison paths:



The height of the decision tree represents the worst-case number of comparisons needed to sort the array, which corresponds to the algorithm's worst-case running time. Since there are $n!$ possible ways to order n distinct elements, the decision tree must have at least $n!$ leaves to account for every possible permutation.

For a binary tree of height h, the maximum number of leaves is 2^h, so we must have:

$$2^h \geq n!$$

Taking the logarithm of both sides and applying Stirling's approximation, $n! \approx \left(\frac{n}{e}\right)^n$, we get:

$$\log n! \geq n \log n - n \log e$$

Thus, the minimum height $H(n)$ of the decision tree is:

$$H(n) = \Omega(n \log n)$$

This proves that any comparison-based sorting algorithm has a worst-case time complexity of $\Omega(n \log n)$.

Theorem 2.5.1. *Any decision tree that can sort n elements must have height $\Omega(n \log n)$.*

Corollary 2.5.1.1. *Heapsort and Merge Sort achieve this asymptotic lower bound, making them optimal comparison-based sorting algorithms.*

2.5.4 Counting sort

Counting sort is a non-comparison-based sorting algorithm that efficiently organizes elements by leveraging their value range rather than making direct comparisons between them. This makes it particularly useful for sorting arrays with integer elements drawn from a small range of possible values.

Algorithm Counting sort takes as input an array $A[n]$, where each element $A[j] \in \{1, \ldots, k\}$, and returns a sorted array $B[n]$. The algorithm also requires an auxiliary array $C[k]$ for counting the frequency of each element in $A[n]$.

The Counting Sort algorithm works in the following steps:

1. *Initialize the counting array*: initialize an array C of size k, where each element is initially set to zero. This array will store the frequency of each element in A.

2. *Count the occurrences*: traverse the input array A, and for each element A[j], increment the corresponding position in array C.

3. *Compute the prefix sum*: update array C by converting it to a prefix sum array. This allows the algorithm to determine the final position of each element in the sorted output.

4. *Build the sorted array*: iterate through the input array A in reverse order, and place each element at its correct position in the output array B, using the values in C for positioning. As elements are placed into B, decrement their corresponding counts in C.

Algorithm 10 Counting Sort

for $i = 1$ **to** k **do** ▷ Set all elements of array C to zero.
    $C[i] = 0$
end for ▷ Complexity $\Theta(k)$
for $j = 1$ **to** n **do** ▷ Count how many times each element appears in the input array A
    $C[A[j]] = C[A[j]] + 1$
end for ▷ Complexity $\Theta(n)$
for $i = 2$ **to** k **do** ▷ Compute the prefix sum for each element in C
    $C[i] = C[i] + C[i - 1]$
end for ▷ Complexity $\Theta(k)$
for $j = n$ **to** 1 **do** ▷ Place elements into output array B and reduce counters in C
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$
end for ▷ Complexity $\Theta(n)$

Example:
Let's consider an example where we sort the array $A = \langle 4, 1, 3, 4, 3 \rangle$, with $k = 4$.

1. *Initial state*: initialize the array C to all zeros: $C = \langle 0, 0, 0, 0 \rangle$.

2. *Count elements*: count the occurrences of each element in A, resulting in: $C = \langle 1, 0, 2, 2 \rangle$.
3. *Compute the prefix sum*: compute the prefix sum over C: $C = \langle 1, 1, 3, 5 \rangle$.
4. *Place elements into output array*: starting from the last element of A, place elements into their correct position in B using the cumulative counts in C, and decrement the counts as we go. The final result will be: $B = \langle 1, 3, 3, 4, 4 \rangle$ and $C = \langle 0, 1, 1, 3 \rangle$.

The time complexity of Counting Sort is the sum of the complexities of each of the four loops

$$T(n) = \mathcal{O}(n) + \mathcal{O}(k) + \mathcal{O}(n) + \mathcal{O}(k) = \mathcal{O}(n + k)$$

If $k = \mathcal{O}(n)$, then counting sort runs in linear time:

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

Definition (*Stable sorting algorithm*)**.** A sorting algorithm is called stable if it preserves the relative order of equal elements from the input array.

Property 2.5.1. Counting Sort is a stable sort.

2.5.5 Radix sort

Radix sort is a non-comparative sorting algorithm that sorts numbers digit by digit, starting from the least significant digit to the most significant digit. It can be highly efficient for large datasets when certain conditions are met.

The idea behind radix sort is to process each digit of the numbers, from the least significant to the most significant, sorting them progressively by each digit's place value. This approach avoids direct comparisons between elements, making it suitable for sorting large datasets when combined with a stable auxiliary sorting algorithm.

Assume the numbers have already been sorted by the least significant $t - 1$ digits. Now, sort the numbers based on the t-th digit. If two numbers differ in the t-th digit, they will be correctly ordered after the pass. If they are identical in the t-th , they retain their relative order due to the stability of the auxiliary sort.

Algorithms The teps for radix sort are:

- *Sort by least significant digit*: sort the numbers based on the least significant digit, using a stable sorting algorithm like counting sort.
- *Iterate through remaining digits*: repeat the sorting process for each more significant digit, ensuring that the relative order of numbers is maintained between passes.
- *Final order*: after all digits have been processed, the numbers are fully sorted.

Analysis To analyze the efficiency of radix sort, we assume that counting sort is used as the stable sorting method for each digit. Suppose we are sorting n integers, where each integer is represented by b bits. Each integer can be thought of as having $\frac{b}{r}$ digits, where each digit is based on 2^r possible values. Each pass of counting sort processes n elements and sorts them based on a single digit, requiring $\Theta(n + 2^r)$ time. Since there are $\frac{b}{r}$ passes (each pass sorting based on one digit), the overall time complexity of radix sort is:

$$T(n) = \Theta\left(\frac{b}{r}(n + 2^r)\right)$$

Optimization To optimize radix sort, we aim to minimize the total running time. Increasing r, the number of bits used for each digit, reduces the number of passes $\frac{b}{r}$, but it also increases the cost of processing each digit, as 2^r grows exponentially.

For optimal efficiency, we want to avoid letting 2^r exceed n, since this would lead to unnecessary overhead. For efficiency, we should avoid letting $2^r > n$.

The optimal choice for r is typically $r = \log n$, as it balances the number of passes and the digit processing cost. Thus, the overall time complexity becomes:

$$T(n) = \Theta\left(\frac{bn}{\log n}\right)$$

Considerations In practice, radix sort is particularly efficient for large datasets, especially when the number of digits is relatively small compared to the number of elements. It is simple to implement and does not require comparisons between elements. However, radix sort has poorer cache locality and memory access patterns compared to algorithms like Quicksort, which can negatively affect performance for smaller datasets or systems with limited memory bandwidth.

2.6 Selection problem

The selection problem involves finding the element of a specified rank in a set of n distinct numbers. Given an integer i where $1 \leq i \leq n$, the task is to return the element that is larger than exactly $i - 1$ other elements in the set. We can have three extreme cases: minimum element ($i = 1$), maximum element ($i = n$), or median element.

2.6.1 Naive algorithm

A straightforward approach to solving the selection problem is to first sort the array and then return the i-th smallest element from the sorted array. The worst-case running time for this approach is dominated by the sorting step, which takes $\Theta(n \log n)$ time. Selecting the i-th element from the sorted array is a constant-time operation, resulting in a total complexity of:

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

While this solution is simple, it is not the most efficient, as it relies on sorting the entire array, even though only one element is ultimately needed.

However, there are more efficient algorithms that can solve the selection problem in linear time. Two popular approaches are:

- *Quickselect*: this algorithm is based on the partitioning method of Quicksort, but instead of recursively sorting both sides of the partition, it only recurses on the side that contains the desired element. Quickselect has an average-case time complexity of $\mathcal{O}(n)$.
- *Median of medians*: this more sophisticated approach uses a median of medians strategy to ensure that each partition step reduces the problem size by a constant fraction. The median of medians algorithm has a worst-case time complexity of $\mathcal{O}(n)$, making it more predictable than Quickselect in terms of performance.

2.6.2 Minmax

To determine the minimum or maximum of a set of n elements, an optimal approach requires exactly $n-1$ comparisons.

Algorithm 11 Minimum and maximum

```
function MINIMUM(A)
    min = A[1]
    for i = 2 to length(A) do
        if min > A[i] then
            min = A[i]
        end if
    end for
    return min
end function

function MAXIMUM(A)
    max = A[1]
    for i = 2 to length(A) do
        if max < A[i] then
            max = A[i]
        end if
    end for
    return max
end function
```

This algorithm performs exactly $n-1$ comparisons, making it optimal for finding either the minimum or maximum in a set.

If both the minimum and maximum are needed, a naive approach would be to execute two passes over the array resulting in $2n-2$ comparisons. However, a more efficient approach allows both the minimum and maximum to be found in fewer than $3\left\lfloor \frac{n}{2} \right\rfloor$ comparisons.

The optimized approach works by comparing elements in pairs and adjusting the minimum and maximum accordingly. This reduces the number of total comparisons by approximately 25%.

2.6.3 Quickselect

Quickselect is an efficient, divide-and-conquer algorithm designed to find the i-th smallest element in an unsorted array with an expected time complexity of $\mathcal{O}(n)$. The algorithm builds on the principles of randomized quicksort by using a pivot to partition the array, but it only recurses on the side that contains the desired element, reducing unnecessary work.

Algorithm 12 Quickselect

```
function RAND-SELECT(A, p, q, i)
    if p = q then
        return A[p]
    end if
    i = RAND-PARTITION(A, p, q)
    k = r − p + 1                                   ▷ k = rank(A[r])
    if i = k then
        return A[r]
    end if
    if i < k then
        return RAND-SELECT(A, p, r − 1, i)
    else
        return RAND-SELECT(A, r + 1, q, i − k)
    end if
end function
```

The running time of Quickselect depends on the quality of the partitioning achieved by the random pivot. This results in the following cases:

- *Best case*: if each partition splits the array evenly, the recurrence relation becomes:

  $$T(n) = T\left(\frac{9}{10}n\right) + \Theta(n) = \Theta(n)$$

  Solving this recurrence yields $\Theta(n)$, meaning that Quickselect runs in linear time when the partition is balanced.

- *Worst case*: if each partition results in only one element on one side and the rest on the other, the recurrence relation is:

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

  This gives $\Theta(n^2)$, leading to quadratic time complexity in the worst case, although this is rare in practice due to random pivot selection.

Analysis The analysis involves defining the expected running time $\mathbb{E}[T(n)]$ and taking into account the probability distribution of possible splits. Let X_k be an indicator variable for whether the partition creates a $k \mid (n-k-1)$ split:

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k \mid n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

The expected running time is then:

$$T(n) = \sum_{k=0}^{n-1} X_k \left(T\left(\max\{k, n-k-1\}\right) + \Theta(n)\right)$$

Taking expectations, we get:

$$\mathbb{E}[T(n)] = \mathbb{E}\left[\sum_{k=0}^{n-1} X_k \left(T\left(\max\{k, n-k-1\}\right) + \Theta(n)\right)\right]$$

By applying bounds and the principle of linearity of expectation, it can be shown that:

$$\mathbb{E}[T(n)] \leq cn$$

For a constant $c > 0$, confirming that Quickselect achieves $\Theta(n)$ expected time complexity.

Practical performance Quickselect is highly efficient in practice, often outperforming deterministic selection algorithms due to its linear average-case time. The worst-case $\Theta(n^2)$ performance is rare, especially if a good pivot strategy or random selection is used. Its efficiency makes it a popular choice in scenarios where the expected linear time is sufficient for robust performance.

2.6.4 Median of medians

The Median of Medians algorithm is a deterministic selection algorithm that guarantees worst-case linear time complexity for selecting the i-th smallest element in an unsorted array. Unlike randomized algorithms, this method avoids the risk of quadratic behavior and provides a reliable worst-case performance.

Algorithm 13 Median of medians

function SELECT(i, n)
  Divide the array in 5 elements groups (each with the median) ▷ Complexity $\Theta(n)$
  Recursively select the median x of the groups medians as pivot ▷ Complexity $T\left(\frac{n}{5}\right)$
  Partition around the pivot x, and let $k = \text{rank}(x)$ ▷ Complexity $\Theta(n)$
  if $i = k$ **then** ▷ Complexity $T\left(\frac{3n}{4}\right)$
    return x
  else if $i < k$ **then**
    SELECT(i-th smallest element in the lower part, n)
  else
    SELECT($(i-k)$-th smallest element in the upper part, n)
  end if
end function

Analysis To understand why the algorithm is efficient, note that choosing x as the median of medians ensures a reasonably balanced partition. Specifically, at least half of the group medians are guaranteed to be less than or equal to x. Since each group has five elements, at least $\left\lfloor \frac{n}{10} \right\rfloor$ elements in A are smaller than x, and at least $\left\lfloor \frac{n}{10} \right\rfloor$ elements are larger. Therefore, each partition discards at least $\left\lfloor \frac{3n}{10} \right\rfloor$ elements, ensuring significant progress with each recursive step. The recurrence relation for the algorithm's time complexity is given by:

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

This relation can be solved to yield $T(n) = \Theta(n)$, proving that the algorithm runs in linear time.

Practical considerations While the median of medians algorithm offers a strong theoretical guarantee of linear time, its practical efficiency is often hindered by relatively high constant factors in its complexity. Here are a few key points about its real-world performance:

- *Work per level*: at each level of recursion, the work done is a fraction of the previous level. Although the algorithm remains linear, the constant factors are substantial due to the overhead of partitioning and the recursive calculation of medians.
- *Comparisons*: altough median of medians is optimal in the worst case, it is often outperformed in practice by the randomized Quickselect algorithm. Quickselect, with its lower constant factors, tends to be faster on average, even though its worst-case complexity is $\Theta(n^2)$.

The Median of Medians algorithm is particularly valuable in applications where a strong worst-case guarantee is essential, ensuring linear time regardless of input characteristics. However, in practical scenarios with large datasets, the randomized Quickselect algorithm is often preferred for its faster average performance, despite the possibility of quadratic worst-case behavior.

2.7 Primality problem

The primality problem involves determining whether a given integer $n \geq 2$ is a prime number.

Definition (*Prime number*)**.** An integer $p \geq 2$ is called prime if and only if it has no positive divisors other than 1 and itself.

2.7.1 Naive algorithm

The simplest way to test if a number n is prime is to check if it has any divisors other than 1 and itself. Since any factor of n greater than $\sqrt{n}$ would have a corresponding factor smaller than $\sqrt{n}$, we only need to check divisibility up to $\sqrt{n}$.

Algorithm 14 Naive primality test

if $n = 2$ **then**
    return true
end if
if n is even **then**
    return false
end if
for $i = 1$ *to* $\sqrt{\frac{n}{2}}$ **do**
    if $2i + i$ divides n **then**
        return false
    end if
end for
return true

The time complexity of this naive algorithm is $\mathcal{O}(\sqrt{n})$.

2.7.2 Fermat primality test

To improve efficiency, we can use a probabilistic primality test based on Fermat's Little Theorem, which states:

Theorem 2.7.1 (Fermat). *If p is a prime number and a and integer a is an integer such that $0 < a < p$, then $a^{p-1} \mod p = 1$.*

This theorem leads to a simple test for primality. If n is prime, $a^{n-1} \mod n = 1$ for some randomly chosen a. However, if n is composite, it may still satisfy this condition for certain a, in which case it is called a pseudoprime to base a.

Algorithm 15 Fermat's primality test

```
if 2^{n-1} mod n = 1 then
    n is possibly prime
else
    n is composite
end if
```

The Fermat test runs in $\mathcal{O}(\log^2 n)$ using modular exponentiation. However, it can mistakenly classify some composite numbers as prime (false positives).

2.7.3 Carmichael primality test

Definition (*Carmichael number*). A composite number $n \geq 2$ is a Carmichael number if, for every integer a coprime to n, it holds that $a^{n-1} \mod n = 1$

Algorithm 16 Carmichael primality test

```
Randomly choose a ∈ [2, n − 1]
if a^{n-1} mod n = 1 then
    n is possibly prime
else
    n is composite
end if
```

2.7.4 Miller-Rabin primality test

The Miller-Rabin test improves on Carmichael's test by checking additional properties that only hold for prime numbers. Specifically, it looks for non-trivial square roots of $1 \mod n$.

Definition (*Non-trivial square root*). An number a is a non-trivial square root of $1 \mod n$ if:

$$a^2 \mod n = 1 \qquad a \neq 1 \qquad a \neq n - 1$$

The Miller-Rabin test randomly selects bases and tests whether they exhibit properties consistent with a prime modulus.

Algorithm 17 Miller-Rabin primality test

```
function POWER(a, p, n)
    if p = 0 then                                   ▷ Compute a^p mod n
        return 1
    end if
    x = POWER(a, p/2, n)
    res = (x · x) % n
    if res = 1 and x ≠ 1 and x ≠ n − 1 then         ▷ check x^2 mod n = 1 and x ≠ 1, n − 1
        isProbablyPrime = false
    end if
    if p % 2 = 1 then
        res = (a · res) % n
    end if
    return res
end function

function PRIMALITYTEST(n)
    a = RANDOM(2, n − 1)
    isProbablyPrime = true
    result = POWER(a, n − 1, n)
    if res ≠ 1 or !isProbablyPrime then
        return false
    else
        return true
    end if
end function
```

Each iteration of the Miller-Rabin test has a low probability of incorrectly identifying a composite number as prime, and repeating the test k times reduces this probability to $\left(\frac{1}{4}\right)^k$.

Theorem 2.7.2. *If p is prime and $0 < a < p$, the only solutions to $a^2 \mod p = 1$ are $a = 1$ and $a = p - 1$.*

Theorem 2.7.3. *If n is composite, the Miller-Rabin test incorrectly classifies n as prime with probability at most is at most $\frac{1}{4}$.*

The Miller-Rabin test runs in $\mathcal{O}(\log^2 n)$ time, making it efficient and reliable for large numbers. It is commonly used in practice for cryptographic applications where probabilistic primality testing is acceptable.

2.8 Dictionary problem

A dictionary is a collection of elements, each associated with a unique search key. The goal is to maintain the set efficiently while supporting operations such as insertions and deletions.

Operation	Description
Search(x, S)	Check if $x \in S$.
Insert(x, S)	Insert x into S if it is not already present
Delete(x, S)	Remove x from S if it exists
Minimum(S)	Return the smallest key in S
Maximum(S)	Return the largest key in S
List(S)	Output the elements of S in increasing order of keys
Union(S_1, S_2)	Merge two sets S_1 and S_2, maintaining the order such that for every $x_1 \in S_1$ and $x_2 \in S_2$, $x_1 < x_2$
Split(S, x, S_1, S_2)	Split S into two sets S_1 and S_2, where all elements in S_1 are $\leq x$ and all elements in S_2 are $> x$

The basic strutures complexity for the main operations are the following:

	Search	Delete	Insert
Unordered array	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Ordered array	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Trees	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

2.8.1 Trees

Binary Search Tree A Binary Search Tree (BST) is a binary tree where each internal node stores an item (k, e) representing a key k and associated element e. The structure of the tree satisfies the property that:

- Keys in the left subtree of any node $v \leq k$.
- Keys in the right subtree of $v > k$.

The drawback of the standard BST is that an unbalanced sequence of insertions may degrade it into a linear structure, resulting in poor performance for searches, inserts, and deletes.

AVL An AVL tree is a self-balancing BST where the heights of the two child subtrees of any node differ by at most one. Rotations ensure that the height of the tree remains logarithmic. While AVL trees guarantee fast lookups and updates, they can be more complex to implement due to the need for maintaining balance factors.

Splay Splay trees are another type of self-adjusting BST. The key idea is the splay operation, which moves a node accessed via a search or update to the root through rotations. This ensures that frequently accessed nodes stay near the root, while infrequently accessed nodes do not contribute much to the overall cost.

2.8.2 Treaps

Treaps are a type of randomized binary search tree that provide efficient time bounds for operations while requiring minimal balance maintenance. Unlike traditional balanced search trees, treaps rely on randomized priorities to achieve a balanced structure, leading to simplicity and efficiency in implementation. The structure of a treap resembles the one obtained if elements were inserted in the order of randomly assigned priorities.

Definition (*Treap*)**.** A is a binary search tree where each node contains an element x with a unique key $\text{key}(x) \in U$ and an associated random priority $\text{prio}(x) \in \mathbb{R}$.

Property 2.8.1 (Search tree)**.** For any node x, all elements y in the left subtree satisfy $\text{key}(y) < \text{key}(x)$, and all elements y in the right subtree satisfy $\text{key}(y) > \text{key}(x)$.

Property 2.8.2 (Heap)**.** For any pair of nodes x and y, if y is a child of x, then $\text{prio}(y) > \text{prio}(x)$.

Lemma 2.8.1. *Given n elements with keys $key(x_i)$ and priorities $prio(x_i)$, there exists a unique treap that satisfies both the search tree property and the heap property.*

Thus, the structure of the treap is entirely determined by the insertion order of elements based on their priorities.

2.8.2.1 Search

Searching in a treap follows the same process as in binary search trees: starting from the root, the search path is determined by comparing the search key with the keys of nodes along the path.

Algorithm 18 Search

$v = root$
while $v \neq \text{null}$ **do**
    if $\text{key}(v) = k$ **then**
        return v ▷ Element found
    end if
    if $\text{key}(v) < k$ **then**
        $v = \text{RIGHTCHILD}(v)$
    end if
    if $\text{key}(v) > k$ **then**
        $v = \text{LEFTCHILD}(v)$
    end if
end while
return null ▷ Element not found

The expected time complexity of a search depends on the depth of the path traversed. For a treap with n elements, the expected depth is $\mathcal{O}(\log n)$ due to the randomized priorities.

Definition (*Harmonic number*)**.** The n-th harmonic number is defined as:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \mathcal{O}(1)$$

Let T be a treap with elements $x_1, \dots, x_n$, and let x_m be the element we are searching for.

Lemma 2.8.2 (Succesful search)**.** *The expected number of nodes on the path to x_m is given by:*

$$H_m + H_{n-m+1} - 1$$

Let m represent the number of keys smaller than the search key k.

Lemma 2.8.3 (Unsuccesful search)**.** *The expected number of nodes on the path during an unsuccessful search is:*

$$H_m + H_{n-m}$$

2.8.2.2 Insertion and deletion

Insertion and deletion operations in treaps involve rotating nodes to maintain the heap property.

Algorithm 19 Insert

```
Choose prio(x)
Search for the position of x in the tree
Insert x as a leaf
while prio(parent(x)) > prio(x) do        ▷ Restore the heap property
    if x is left child then
        RotateRight(parent(x))
    else
        RotateLeft(parent(x))
    end if
end while
```

Algorithm 20 Delete

```
Find x in the tree
while x is not a leaf do
    u = child with smaller priority
    if u is left child then
        RotateRight(x)
    else
        RotateLeft(x)
    end if
end while
Delete x
```

These operations maintain both the search tree property and heap property.

Lemma 2.8.4. *The expected running time of the insert and delete operations is $\mathcal{O}(\log n)$, with an expected 2 rotations per operation.*

2.8.2.3 Split and union

Split To split treap T by key k:

1. Insert a new element x with $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$.
2. Insert x into T.
3. Delete x; the left and right subtrees of x become T_1 and T_2, respectively.

Union To merge two treaps T_1 and T_2:

1. Select a key k such that $\text{key}(x_1) < k < \text{key}(x_2)$ for all $x_1 \in T_1$ and $x_2 \in T_2$.
2. Create a new node x with $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$.
3. Set T_1 and T_2 as the left and right subtrees of x, respectively.
4. Delete x from the resulting tree.

Lemma 2.8.5. *The expected time complexity of both union and split operations is $\mathcal{O}(\log n)$.*

2.8.2.4 Implementation

In treaps, priorities are random values drawn from $[0, 1)$, ensuring that tree balancing remains probabilistic rather than explicit. If two nodes have equal priorities, tie-breaking is achieved by appending uniformly random bits to the priorities until a difference is found. This preserves randomness and ensures that the heap property is maintained.

2.8.3 Skip lists

Skip lists, introduced by William Pugh in 1989, are a randomized, dynamic data structure that maintains a sorted set of elements with efficient average-case operations for search, insertion, and deletion. They offer a probabilistic time complexity of $\mathcal{O}(\log n)$ for these operations, making them simple to implement yet powerful for dynamic sets where performance is expected rather than strictly guaranteed.

Skip lists improve upon the basic sorted linked list by adding additional linked lists layered above the main list. The main list connects all elements, like a standard linked list, while each higher level connects increasingly sparse subsets of elements, allowing faster traversal by skipping over parts of the lower lists.

2.8.3.1 Search

To search for an element in a skip list:

1. Begin at the highest level list.
2. Traverse each level by moving right until the target is either found or overshot.
3. If overshot, drop down to the next level and repeat the process.
4. Continue this process until reaching the bottom level, where the target element is either located or confirmed absent.

The higher levels of the skip list serve as express lanes, allowing large jumps, while the lower levels provide finer granularity in search.

Analysis With two levels in the skip list, the search cost is approximately:

$$T(n) = |L_1| + \frac{|L_2|}{|L_1|}$$

This is minimized when:

$$T(n) = |L_1|^2 = |L_2| = n \implies |L_1| = \sqrt{n}$$

Resulting in $T(n) = 2\sqrt{n}$. Generalizing this to k levels gives a cost of $k\sqrt[k]{n}$ With $\log n$ levels, the cost becomes:

$$T(n) = \mathcal{O}(\log n)$$

This efficient layout mimics a balanced binary tree, enabling skip lists to support rapid searching in practice.

2.8.3.2 Insertion

To insert a new element x:

1. Search for x's position in the bottom list.
2. Insert x into the bottom list, which holds all elements in sorted order.
3. Randomly promote x to higher levels based on coin flips. For each level, x is promoted with a probability of $\frac{1}{2}$, ensuring that, on average, only a small fraction of elements reach the top levels. This randomized promotion keeps the structure balanced with $\log n$ expected levels.

The insertion process results in a skip list with a logarithmic number of levels, where the promotion of elements ensures balance across the structure.

2.8.3.3 Implementation

Skip lists are widely used in practice due to their efficiency, with search operations typically taking average time $\mathcal{O}(\log n)$.

Theorem 2.8.6. *With high probability, the search time for an n-element skip list is $\mathcal{O}(\log n)$.*

Here, the phrase with high probability signifies that the probability of this time complexity holding is at least $1 - \mathcal{O}\left(\frac{1}{n^\alpha}\right)$ for a chosen constant $\alpha \geq 1$. By increasing α, the likelihood of search times exceeding $\mathcal{O}(\log n)$ can be made arbitrarily low, making this bound practically reliable.

Lemma 2.8.7. *With high probability, an n-element skip list has $\mathcal{O}(\log n)$ levels.*

CHAPTER 3

Amortized analysis

3.1 Introduction

Amortized analysis is a technique used to assess the average cost per operation over a sequence of operations, ensuring that the overall performance remains efficient even if certain individual operations are more expensive. Unlike probabilistic analysis, which relies on random events, amortized analysis provides a guaranteed bound on the average cost per operation, even in the worst-case scenario.

There are three primary methods of amortized analysis:

- *Aggregate method*: this method calculates a simple average cost over all operations, providing an overall estimate, though it lacks the precision of more nuanced approaches.
- *Accounting method*: this approach employs a "banking" system, allocating an amortized cost to each operation to ensure that the total cost is distributed appropriately over a sequence of operations.
- *Potential method*: this method introduces a potential function to track the stored energy of a system, which helps to manage and distribute the amortized costs dynamically.

Hash table resizing A well-designed hash table strikes a balance between compactness and size to minimize overflow and maintain efficient access. However, determining an optimal size for the table upfront is often impractical, as it may not accurately reflect the growth of stored entries. To address this issue, dynamic resizing is employed. When the hash table reaches its capacity, a larger table is allocated, all existing entries are rehashed into the new table, and the memory from the old table is freed. This dynamic resizing mechanism ensures that the hash table adapts to changes in size without sacrificing efficiency, avoiding the need for a predefined size.

3.2 Aggregate method

The aggregate method of amortized analysis calculates the average cost per operation over a sequence of operations, even when some individual operations may be expensive. This technique is especially useful for data structures that involve periodic costly operations, as it spreads the cost of these infrequent expensive operations across many cheaper ones.

Hash table resizing While a single insertion during a resizing operation might appear costly, with a time complexity of $\mathcal{O}(n)$, this does not imply that n insertions will collectively result in a cost of $\mathcal{O}(n^2)$. In practice, the total cost for n insertions remains closer to $\mathcal{O}(n)$, ensuring better overall efficiency.

To illustrate this, let the cost of the i-th insertion be denoted as c_i:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

In this scheme, when $i-1$ is an exact power of 2, the hash table doubles in size, and all existing entries must be rehashed and inserted into the new table, resulting in a higher cost. For all other insertions, the cost remains constant at 1.

Despite the occasional high cost of resizing, the total cost for n insertions over time is $\mathcal{O}(n)$. This means that, on average, the cost of each insertion is:

$$\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$$

3.3 Accounting method

In the accounting method of amortized analysis, each operation is assigned a fictitious amortized cost, denoted as $\hat{c}_i$, which represents an accounting balance for the operation. This balance can either be used immediately or saved to cover the cost of future operations. The amortized cost consists of two main components:

- *Immediate cost*: the actual cost incurred for performing the operation.
- *Banked cost*: any excess cost that is set aside and saved for future operations.

The core idea of the accounting method is that the total accumulated banked cost should never be negative, ensuring that there are sufficient funds to cover the costs of future operations. Mathematically, this can be expressed as:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Here, c_i represents the true cost of the i-th operation, and $\hat{c}_i$ is the amortized cost. By ensuring that the bank balance remains non-negative, the amortized cost provides an upper bound on the true total cost of all operations, thereby guaranteeing efficient performance.

Hash table resizing For a dynamic hash table that adjusts its size as needed, the accounting method can be applied to model the costs associated with insertions and table expansions. In this case, each insertion is assigned an amortized cost of $\hat{c}_i = 3$:

- *Immediate cost*: the immediate cost of inserting an element into the expanded table is one unit.

- *Banked cost*: two units are banked with each insertion, to cover the cost of future expansions.

When the table doubles in size, the banked cost ensures that the expansion remains efficient. Specifically, half of the banked units are used to insert the existing elements into the new table, while the remaining units help move the elements that were added after the last expansion. This strategy ensures that the bank balance never goes negative, allowing the amortized cost to provide an upper bound on the true cost of all operations.

By charging each insertion an amortized cost of three, and using the banked cost to effectively cover the resizing expenses, the dynamic hash table remains efficient. The amortized cost guarantees that the average cost per operation remains constant over time, even during resizing operations, thus ensuring sustained performance.

3.4 Potential method

The potential method of amortized analysis conceptualizes the "bank account" as the potential energy of a dynamic sequence of operations. The aim is to use a potential function to account for the work done by each operation and how it impacts the overall cost over time.

In this framework, we begin with an initial data structure, denoted as D_0. Each operation i transitions the data structure from D_{i-1} to D_i, incurring a cost c_i. To analyze the costs using the potential method, we define a potential function Φ that maps each data structure state D_i to a real number ($\Phi : \{D_i\} \to \mathbb{R}$). The potential function must satisfy the following properties:

$$\Phi(D_0) = 0 \qquad \Phi(D_i) \geq 0$$

The amortized cost $\hat{c}_i$ for an operation i is then defined as:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + \Delta\Phi_i$$

Here, $\Delta\Phi_i$ is the potential difference between the states before and after the operation.

The potential difference $\Delta\Phi_i$ can be either positive or negative:

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$, meaning the operation deposits work into the data structure to be used by future operations.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$, meaning the data structure delivers stored work to help cover the current operation's cost.

The total amortized cost over n operations is:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

This inequality ensures that the total amortized cost provides an upper bound on the true total cost of the operations.

Hash table resizing To apply the potential method to the dynamic resizing of a hash table, we define the potential of the table after the i-th insertion as:

$$\Phi(D_i) = 2i - 2^{\lceil \log i \rceil}$$

Note that we assume $2^{\lceil \log 0 \rceil} = 0$, which accounts for the table's growth during resizing.

The amortized cost of the i-th insertion is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + (2i - 2^{\lceil \log i \rceil}) - (2(i-1) - 2^{\lceil \log(i-1) \rceil})$$

The true cost c_i of the i-th insertion is:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

When $i-1$ is an exact power of 2 (i.e., the table needs to be resized), the amortized cost becomes:

$$\hat{c}_i = i + 2 - 2i + 2 + i - 1 = 3$$

In the case where $i-1$ is not a power of 2, the amortized cost remains:

$$\hat{c}_i \approx 3$$

Thus, in both cases, the amortized cost per insertion is three. Consequently, after n insertions, the total cost is $\Theta(n)$ in the worst case, ensuring that the dynamic hash table remains efficient even with frequent resizing operations.

CHAPTER 4

Parallel computing model

4.1 Random Access Machine

Definition (*Random Access Machine*)**.** A Random Access Machine (RAM) is a theoretical computational model that features the following characteristics:

- *Unbounded memory cells*: the machine has an unlimited number of local memory cells.
- *Unbounded integer capacity*: each memory cell can store an integer of arbitrary size, without any constraints.
- *Simple instruction set*: the instruction set includes basic operations such as arithmetic, data manipulation, comparisons, and conditional branching.
- *Unit-time operations*: every operation is assumed to take a constant, unit time to complete.

The time complexity of a RAM is determined by the number of instructions executed during computation, while the space complexity is measured by the number of memory cells utilized.

4.2 Parallel Random Access Machine

A Parallel Random Access Machine (PRAM) is an abstract machine designed to model algorithms for parallel computing.

Definition (*Parallel Random Access Machine*)**.** A Parallel Random Access Machine (PRAM) is defined as a system $M' = \langle M, X, Y, A \rangle$, where:

- M represent an infinite collection of identical RAM processors without memory.
- X represent the system's input.
- Y represent the system's output.
- A are shared memory cells between processors.

The set of RAMs M contains an unbounded collection of processors P, that have unbounded registers for internal storage. The set of shared memory cells A is unbounded and can be accessed in constant time. This set is used by the processors P to communicate with each other.

4.2.1 Computation

The computation in a PRAM consists of five phases, carried out in parallel by all processors. Each processor performs the following actions:

1. Reads a value from one of the input cells X_i.
2. Reads from one of the shared memory cells A_i.
3. Performs some internal computation.
4. May write to one of the output cells Y_i.
5. May write to one of the shared memory cells A_i.

Some processors may remain idle during computation.

Conflicts Conflicts can arise in the following scenarios:

- *Read conflicts*: two or more processors may simultaneously attempt to read from the same memory cell.
- *Write conflicts*: two or more processors attempt to write simultaneously to the same memory cell.

PRAM models are classified based on their ability to handle read/write conflicts, offering both practical and realistic classifications:

PRAM model	Operation
Exclusive Read	Read from distinct memory locations
Exclusive Write	Write to distinct memory locations
Concurrent Read	Read from the same memory locations
Concurrent Write	Write to the same memory locations

When a write conflict occurs, the final value written depends on the conflict resolution strategy:

- *Priority CW*: processors are assigned priorities, and the value from the processor with the highest priority is written.
- *Common CW*: all processors are allowed to complete their write only if all values to be written are equal.
- *Arbitrary CW*: a randomly chosen processor is allowed to complete its write operation.

4.2.2 Conclusion

The PRAM model is both attractive and important for parallel algorithm designers for several reasons:

- *Natural*: the number of operations executed per cycle on P processors is at most P.
- *Strong*: any processor can access and read/write any shared memory cell in constant time.
- *Simple*: it abstracts away communication or synchronization overhead.
- *Benchmark*: if a problem does not have an efficient solution on a PRAM, it is unlikely to have an efficient solution on any other parallel machine.

Some possible variants of the PRAM machine model are:

- *Bounded number of shared memory cells*: when the input data set exceeds the capacity of the shared memory, values can be distributed evenly among the processors.
- *Bounded number of processors*: if the number of execution threads is higher than the number of processors, processors may interleave several threads to handle the workload.
- *Bounded size of a machine word*: limits the size of data elements that can be processed in a single operation.
- *Handling access conflicts*: constraints on simultaneous access to shared memory cells must be considered.

4.3 Performance

The main values used to evaluate the performance are:

Parameter	**Description**
$T^*(n)$	Time to solve a problem of input size n on one processor using best sequential algorithm
$T_1(n)$	Time to solve a problem on one processor
$T_p(n)$	Time to solve a problem on p processors
$T_\infty(n)$	Time to solve a problem on ∞ processors
$\mathrm{SU}_p = \dfrac{T^*(n)}{T_p(n)}$	Speedup on p processors
$E_p = \dfrac{T_1}{pT_p(n)}$	Efficiency
$C(n) = pT_p(n)$	Cost
$W(n)$	Work (total number of operations)

4.3.1 Matrix-vector multiplication

Matrix-vector multiplication involves multiplying a matrix by a vector.

To perform the multiplication, each element of the resulting vector is computed by taking the dot product of the rows of the matrix with the vector. Specifically, if you have a matrix **A** of size $n \times n$ and a vector **v** of size n, the resulting vector **u** will have size $n \times n$:

$$\mathbf{u} = \mathbf{A}\mathbf{v}$$

The entry u_i of the resulting vector is calculated as:

$$u_i = \sum_{j=1}^{n} a_{ij} v_j$$

Here, a_{ij} are the elements of the matrix **A**. The algorithm that computes the vector **u** is:

Algorithm 21 Matrix-vector multiplication

1: Global read $x \leftarrow \mathbf{v}$ ▷ Broadcast vector **v** to all processors
2: Global read $y \leftarrow \mathbf{a}_i$ ▷ Read corresponding rows of matrix **A**
3: Compute $w = xy$ ▷ Multiply matrix row with vector **v**
4: Global write $w \rightarrow u_i$ ▷ Write result to the output vector **u**

The performance measures of this algorithm in the best-case scenario are shown in the following table:

Measure	T_1	T_p
Complexity	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^2}{p}\right)$

4.3.2 Single program multiple data sum

In single program multiple data (SPMD), each processor operates independently on its subset of the data, typically using the same code but possibly with different input data. This model is commonly used in high-performance computing, scientific simulations, and data analysis tasks, enabling significant performance improvements by leveraging parallelism.

In the context of SPMD, a sum refers to the process of aggregating data from multiple processors or cores that are executing the same program on different segments of data. Here's how it typically works:

1. *Data distribution*: the data is divided into chunks, with each CPU assigned a specific subset to work on.

2. *Local computation*: each processor executes the same summation program on its assigned data.

3. *Local results*: after computing their local sums, each processor has a partial sum.

4. *Reduction*: the partial sums are then combined (reduced) to get the final sum.

5. *Final output*: the final result is the total sum of all the partial sums computed by the individual processors.

Algorithm 22 SPMD sum

1: Global read $x \leftarrow \mathbf{b}$ ▷ Broadcast array $\mathbf{b}$ to all processors
2: Global write $y \rightarrow \mathbf{c}$ ▷ Broadcast array $\mathbf{c}$ to all processors
3: Compute $z = x + y$ ▷ Sum all vectors elements
4: Global write $z \rightarrow \mathbf{a}$ ▷ Write result to the output array $\mathbf{a}$

The performance measures of this algorithm are shown in the following table:

Measure	T_1	T_p
Complexity	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{p} + \log p\right)$

4.3.3 Matrix-matrix multiplication

Matrix-matrix multiplication involves multiplying a matrix by another matrix.

To perform the multiplication, each element of the resulting matrix is computed by taking the dot product of the rows of the first matrix with the columns of the second matrix. Specifically, if you have a matrix $\mathbf{A}$ of size $m \times n$ and a matrix $\mathbf{B}$ of size $n \times p$, the resulting matrix $\mathbf{C}$ will have size $m \times p$:

$$\mathbf{C} = \mathbf{AB}$$

The entry c_{ij} of the resulting matrix is calculated as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Here, a_{ik} are the elements of matrix $\mathbf{A}$ and b_{kj} are the elements of matrix $\mathbf{B}$.

Algorithm 23 Matrix-matrix multiplication

1: Global read $x \leftarrow \mathbf{a}_i$ ▷ Read corresponding rows of matrix $\mathbf{A}$
2: Global read $y \leftarrow \mathbf{b}_i$ ▷ Read corresponding columns of matrix $\mathbf{B}$
3: Compute $w = xy$ ▷ Multiply matrix $\mathbf{A}$ row with matrix $\mathbf{B}$ column
4: Global write $w \rightarrow \mathbf{u}_i$ ▷ Write result to corresponding row of output matrix $\mathbf{u}$

The performance measures of this algorithm are shown in the following table:

Measure	T_1	T_p
Complexity	$\mathcal{O}(n^3)$	$\mathcal{O}\left(\frac{n^3 \log n}{p}\right)$

4.4 Prefix sum

Given a sequence of values $\{a_1, \ldots, a_n\}$, the prefix sum S_i up to position i is defined as:

$$S_i = \sum_{j=1}^{i} a_j$$

In the case of prefix sums, the total computational work required by a parallel algorithm exceeds that of a serial algorithm.

For a serial algorithm, computing each prefix sum is straightforward: each element in the prefix sum can be computed in sequence, where S_i simply depends on S_{i-1} and a_i. This approach only requires $\mathcal{O}(n)$ operations, with each element added once.

In contrast, a parallel algorithm introduces additional overhead. To achieve parallelism, the algorithm needs to divide the work among processors, requiring intermediate calculations and combining steps. Thus, the parallel prefix sum algorithm typically involves $\mathcal{O}(n \log n)$ operations, as it requires multiple rounds to propagate intermediate results across processors.

4.5 Model analysis

Definition (*Computationally Stronger*)**.** A model A is said to be computationally stronger than model B ($A \geq B$) if any algorithm written for B can run unchanged on A with the same parallel time and basic properties.

Lemma 4.5.1. *Assume $P' < P$, and same size of shared memory. Any problem that can be solved for a P-processor PRAM in T steps can be solved in a P' processor PRAM in $T' = O(T\frac{P}{P'})$ steps*

Lemma 4.5.2. *Assume $M' < M$. Any problem that can be solved for a P-processor and M-cell PRAM in T steps can be solved on a $\max(P, M')$-processor M'-cell PRAM in $\mathcal{O}\left(T\frac{M}{M'}\right)$ steps.*

The direct implementation of a PRAM on real hardware poses certain challenges due to its theoretical nature. Despite this, PRAM algorithms can be adapted for practical systems, allowing the abstract model to influence real-world designs.

4.5.1 Amdahl law

In parallel computing, we consider two types of program segments: serial segments and parallelizable segments. The total execution time depends on the proportion of each.

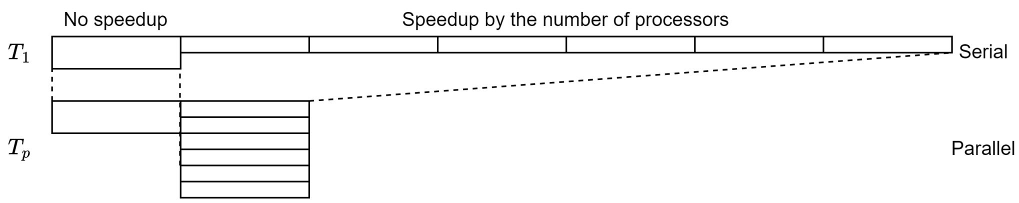


Figure 4.1: Serial and parallel models

When using more than one processor, the speedup is always less than the number of processors. In a program, the parallelizable portion is often represented by a fixed fraction f.

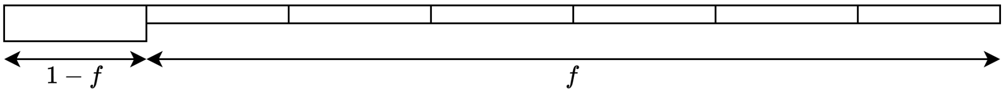


Figure 4.2: Serial model

Using the serial version of the model, the speedup function $\mathrm{SU}(p, f)$ is derived as follows:

$$\mathrm{SU}(p, f) = \frac{1}{(1-f) + \frac{f}{p}}$$

As the number of processors p approaches infinity, the speedup is limited by the serial portion:

$$\lim_{p \to \infty} \mathrm{SU}(p, f) = \frac{1}{1-f}$$

This shows that even with an infinite number of processors, the maximum speedup is constrained by the serial fraction of the program.

4.5.2 Gustafson law

In contrast to Amdahl's Law, John Gustafson proposed a different view in 1988, challenging the assumption that the parallelizable portion of a program remains fixed. Key differences include:

- The parallelizable portion of the program is not a fixed fraction.
- Absolute serial time is fixed, while the problem size grows to exploit more processors.

Amdahl's law is based on a fixed-size model, while Gustafson's law operates on a fixed-time model, where the problem grows with increased processing power. The speedup in Gustafson's model is expressed as:

$$\mathrm{SU}(p) = s + p(1-s)$$

Here, s is the fixed serial portion of the program. As a result, this model suggests linear speedup is possible as the number of processors increases, especially for highly parallelizable tasks. Gustafson's law is empirically applicable to large-scale parallel algorithms, where increasing computational power enables solving larger and more complex problems within the same time frame.