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 complexity

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 theta notation 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 $\langle a_1, a_2, \ldots, a_n \rangle$ and returning the permutation of the input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

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[1 ... ⌈n/2⌉] and A[⌈n/2⌉ + 1 ... n]
Merge (A[1 ... ⌈n/2⌉], A[⌈n/2⌉ + 1 ... n])
```

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.

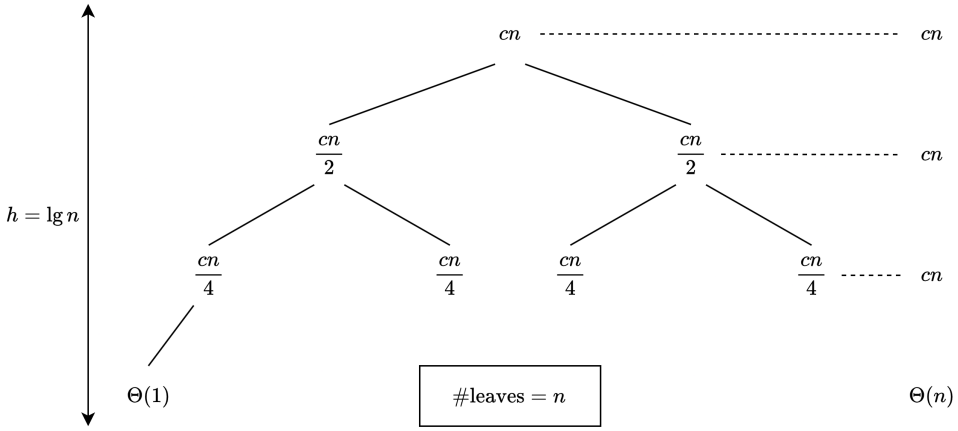


Figure 1.1: Partial recursion tree for merge sort algorithm

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

Divide and conquer algorithms

2.1 Introduction

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:

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

2.2 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:

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

2.3 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:

$$\Theta(\log n)$$

2.4 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

```
1: for i = 1 to n do
2:     for j = 1 to n do
3:         c_ij = 0
4:         for k = 1 to n do
5:             c_ij = c_ij + a_ik b_kj
6:         end for
7:     end for
8: 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.

2.5 VLSI layout

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

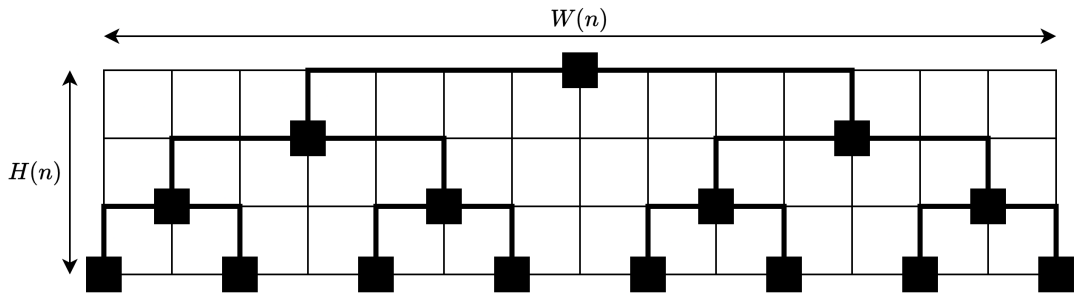


Figure 2.1: VLSI layout problem

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.

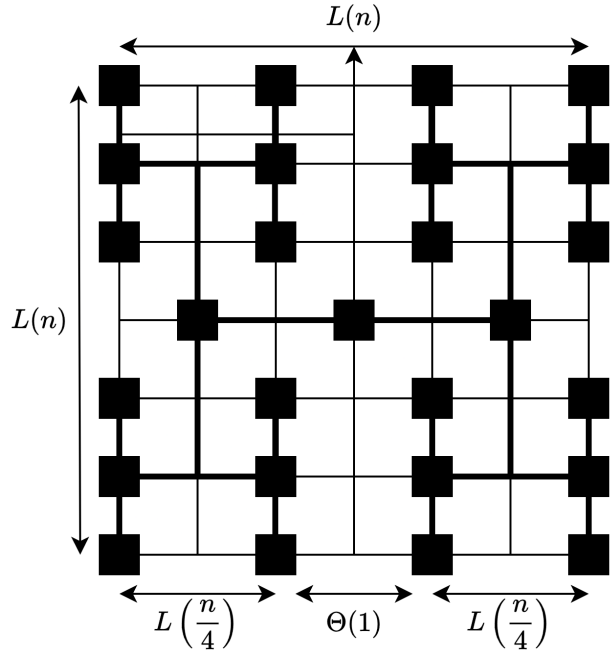


Figure 2.2: VLSI layout problem

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:

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

CHAPTER 3

Parallel machine model

3.1 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.

3.2 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.

3.2.1 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.

3.2.2 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.

3.3 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
$\mathrm{SU}_p = \frac{T^*(n)}{T_p(n)}$	Speedup on p processors
$E_p = \frac{T_1}{pT_p(n)}$	Efficiency
$C(n) = pT_p(n)$	Cost
$W(n)$	Work (total number of operations)

3.3.1 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 $\mathbf{A}$ of size $n \times n$ and a vector $\mathbf{v}$ of size n, the resulting vector $\mathbf{u}$ 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 $\mathbf{A}$. The algorithm that computes the vector $\mathbf{u}$ is:

Algorithm 4 Matrix-vector multiplication

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

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

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

3.3.2 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 5 SPMD sum

1: Global read $x \leftarrow \mathbf{b}$	▷ Broadcast array **b** to all processors
2: Global write $y \rightarrow \mathbf{c}$	▷ Broadcast array **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 **a**

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

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

3.3.3 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 **A** of size $m \times n$ and a matrix **B** of size $n \times p$, the resulting matrix **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 **A** and b_{kj} are the elements of matrix **B**.

Algorithm 6 Matrix-matrix multiplication

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

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

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

3.4 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.

3.5 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 3.5.1. *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 3.5.2. *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.

3.5.1 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.

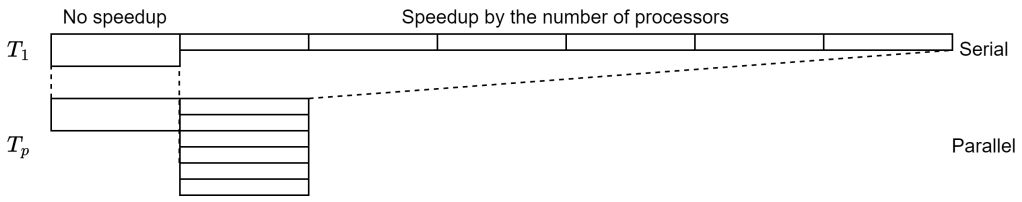


Figure 3.1: Serial and parallel models

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.

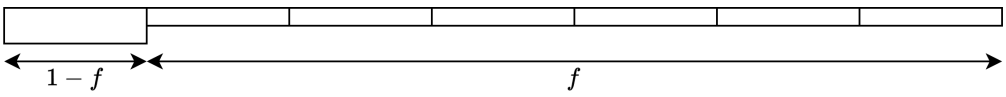


Figure 3.2: Serial model

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.

3.5.2 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.

CHAPTER 4

Advanced algorithms

4.1 Introduction

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.

4.1.1 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.

4.2 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.

4.2.1 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:

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

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

4.2.2 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.

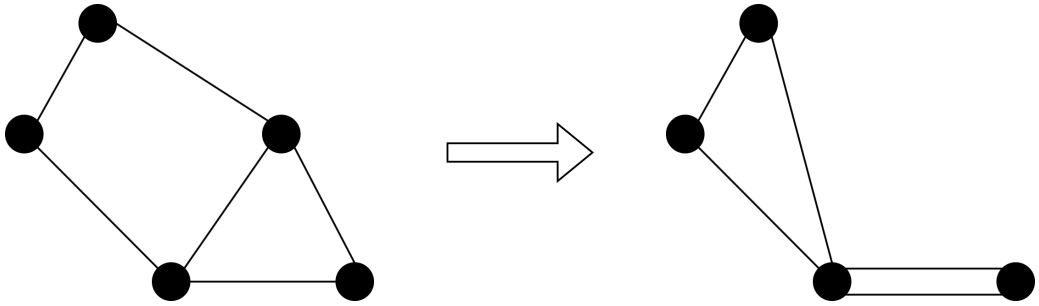


Figure 4.1: Edge contraction

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 4.2.1. *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:

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

And an error probability of at most:

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

4.2.3 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 7 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:

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

Corollary 4.2.1.1. *Any graph has at most $\mathcal{O}(n^2)$ distinct minimum cuts.*

4.3 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.

4.3.1 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 8 Quicksort

```
1: function PARTITION(A, p, q)
2:     x = A[p]
3:     i = p
4:     for j = p + 1 to q do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             exchange A[i] and A[j]
8:         end if
9:     end for
10:    exchange A[p] and A[i]
11:    return i
12: end function

13: procedure QUICKSORT(A, p, r)
14:    if p < r then
15:        q = PARTITION(A, p, r)
16:        QUICKSORT(A, p, q − 1)
17:        QUICKSORT(A, q + 1, r)
18:    end if
19: 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 $\Theta(n \log n)$.

4.3.2 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:

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

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.

4.3.3 Comparison sort analysis

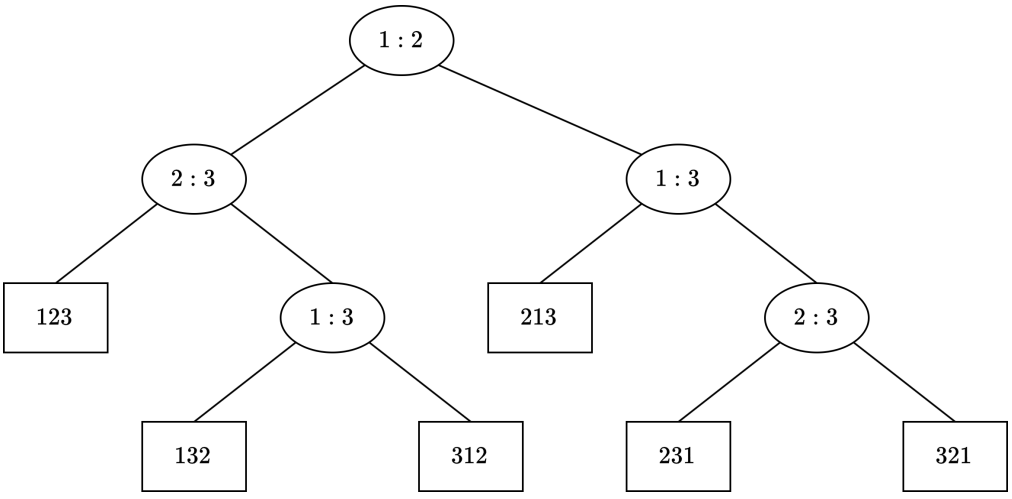
All the sorting algorithms discussed so far are comparison sorts, where element ordering is determined by comparing pairs of elements. The best worst-case time complexity for these algorithms is $\mathcal{O}(n \log n)$.

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

Corollary 4.3.1.1. *Heapsort and Merge Sort achieve this asymptotic lower bound, making them optimal comparison-based sorting algorithms.*

4.3.4 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 9 Counting Sort

```
1: for i = 1 to k do                                   ▷ Set all elements of array C to zero.
2:     C[i] = 0
3: end for                                             ▷ Complexity Θ(k)
4: for j = 1 to n do   ▷ Count how many times each element appears in the input array A
5:     C[A[j]] = C[A[j]] + 1
6: end for                                             ▷ Complexity Θ(n)
7: for i = 2 to k do                    ▷ Compute the prefix sum for each element in C
8:     C[i] = C[i] + C[i − 1]
9: end for                                             ▷ Complexity Θ(k)
10: for j = n to 1 do          ▷ Place elements into output array B and reduce counters in C
11:     B[C[A[j]]] = A[j]
12:     C[A[j]] = C[A[j]] − 1
13: end for                                            ▷ Complexity Θ(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

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

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

$$\Theta(n)$$

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

Property 4.3.1. Counting Sort is a stable sort.

4.3.5 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:

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

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.

4.4 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.

4.4.1 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:

$$\Theta(n \log n)$$

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.

4.4.2 Minmax

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

Algorithm 10 Minimum and maximum

```
1: function MINIMUM(A)
2:     min = A[1]
3:     for i = 2 to length(A) do
4:         if min > A[i] then
5:             min = A[i]
6:         end if
7:     end for
8:     return min
9: end function

10: function MAXIMUM(A)
11:     max = A[1]
12:     for i = 2 to length(A) do
13:         if max < A[i] then
14:             max = A[i]
15:         end if
16:     end for
17:     return max
18: 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%.

4.4.3 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 11 Quickselect

```
1: function RAND-SELECT(A, p, q, i)
2:     if p = q then
3:         return A[p]
4:     end if
5:     i = RAND-PARTITION(A, p, q)
6:     k = r − p + 1                                   ▷ k = rank(A[r])
7:     if i = k then
8:         return A[r]
9:     end if
10:    if i < k then
11:        return RAND-SELECT(A, p, r − 1, i)
12:    else
13:        return RAND-SELECT(A, r + 1, q, i − k)
14:    end if
15: 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:

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

Taking expectations, we get:

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

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.

4.4.4 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 12 Median of medians

```
1: function SELECT(i, n)
2:     Divide the array in 5 elements groups (each with the median)      ▷ Complexity Θ(n)
3:     Recursively select the median x of the groups medians as pivot    ▷ Complexity T(n/5)
4:     Partition around the pivot x, and let k = rank(x)                 ▷ Complexity Θ(n)
5:     if i = k then                                                     ▷ Complexity T(3n/4)
6:         return x
7:     else if i < k then
8:         SELECT(i-th smallest element in the lower part, n)
9:     else
10:        SELECT((i − k)-th smallest element in the upper part, n)
11:    end if
12: 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.

4.5 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.

4.5.1 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 13 Naive primality test

```
1: if n = 2 then
2:     return true
3: end if
4: if n is even then
5:     return false
6: end if
7: for i = 1 to √(n/2) do
8:     if 2i + i divides n then
9:         return false
10:    end if
11: end for
12: return true
```

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

4.5.2 Fermat primality test

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

Theorem 4.5.1 (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 14 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).

4.5.3 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 15 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
```

4.5.4 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 16 Miller-Rabin primality test

```
1: function POWER(a, p, n)
2:     if p = 0 then                                   ▷ Compute a^p mod n
3:         return 1
4:     end if
5:     x = POWER(a, p/2, n)
6:     res = (x · x) % n
7:     if res = 1 and x ≠ 1 and x ≠ n − 1 then      ▷ check x^2 mod n = 1 and x ≠ 1, n − 1
8:         isProbablyPrime = false
9:     end if
10:    if p % 2 = 1 then
11:        res = (a · res) % n
12:    end if
13:    return res
14: end function

15: function PRIMALITYTEST(n)
16:    a = random(2, n − 1)
17:    isProbablyPrime = true
18:    result = POWER(a, n − 1, n)
19:    if res ≠ 1 or !isProbablyPrime then
20:        return false
21:    else
22:        return true
23:    end if
24: 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 4.5.2. *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 4.5.3. *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.

4.6 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	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

4.6.1 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.

4.7 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 4.7.1 (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 4.7.2 (Heap)**.** For any pair of nodes x and y, if y is a child of x, then $\text{prio}(y) > \text{prio}(x)$.

Lemma 4.7.1. *Given n elements with keys $\text{key}(x_i)$ and priorities $\text{prio}(x_i)$, 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.

4.7.1 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 17 Search

```
1:  v = root
2:  while v ≠ null do
3:      if key(v) = k then
4:          return v                                  ▷ Element found
5:      end if
6:      if key(v) < k then
7:          v = RIGHTCHILD(v)
8:      end if
9:      if key(v) > k then
10:         v = LEFTCHILD(v)
11:     end if
12: end while
13: 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 $x_1, \ldots, x_n$, and let x_m be the element we are searching for.

Lemma 4.7.2 (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 4.7.3 (Unsuccesful search)**.** *The expected number of nodes on the path during an unsuccessful search is:*

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

4.7.2 Insertion and deletion

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

Algorithm 18 Insert

```
1: Choose prio(x)
2: Search for the position of x in the tree
3: Insert x as a leaf
4: while prio(parent(x)) > prio(x) do          ▷ Restore the heap property
5:     if x is left child then
6:         RotateRight(parent(x))
7:     else
8:         RotateLeft(parent(x))
9:     end if
10: end while
```

Algorithm 19 Delete

```
1: Find x in the tree
2: while x is not a leaf do
3:     u = child with smaller priority
4:     if u is left child then
5:         RotateRight(x)
6:     else
7:         RotateLeft(x)
8:     end if
9: end while
10: Delete x
```

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

Lemma 4.7.4. *The expected running time of the insert and delete operations is $\mathcal{O}(\log n)$, with an expected 2 rotations per operation.*

4.7.3 Split and union

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.

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

4.7.4 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.

4.8 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.

4.8.1 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:

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

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

4.8.2 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.

4.8.3 Implementation

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

Theorem 4.8.1. *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 4.8.2. *With high probability, an n-element skip list has* $\mathcal{O}(\log n)$ *levels.*

CHAPTER 5

Dynamic programming

5.1 Introduction

The term dynamic programming was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. The word dynamic was chosen by Bellman to capture the time-varying aspect of the problems. The word programming referred to the use of the method to find an optimal program, in the sense of a military schedule for training or logistics.

5.2 Longest common subsequence problem

Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$, the goal is to find the longest subsequence that is common to both sequences. A subsequence is derived by deleting some (or no) elements from the original sequence without rearranging the remaining elements. The Longest Common Subsequence (LCS) problem, therefore, seeks a subsequence of maximum possible length that appears in both sequences in the same relative order.

Naive algorithm The brute-force approach to solving the LCS problem generates all possible subsequences of the first sequence x and checks each one to see if it is also a subsequence of y. The steps for the brute-force LCS algorithm are outlined below:

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

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

3. *Track the Longest Common Subsequence*: while iterating over all subsequences, keep track of the longest one that is a subsequence of both x and y. Once all possible subsequences have been checked, the longest of these will be the LCS.

The brute-force algorithm is highly inefficient due to its exponential time complexity: There are 2^m subsequences of x, as each element in x has the option to either be included or excluded from any given subsequence. To verify if a subsequence of x is also a subsequence of y, we must iterate through y, which requires $\mathcal{O}(n)$ time for each subsequence. Since there are 2^m subsequences to check, and each check takes $\mathcal{O}(n)$ time, the total time complexity of the brute-force LCS algorithm is:

$$\mathcal{O}(n2^m)$$

In the worst case, this approach quickly becomes computationally infeasible for even moderately large input sizes.

5.2.1 Recursive algorithm

To solve the LCS problem recursively, we break it down as follows:

1. *Find the length of the LCS*: define a recursive function that returns the length of the longest common subsequence between prefixes of two sequences.
2. *Extend to return the LCS itself*: modify the algorithm to keep track of subsequence characters for reconstructing the LCS.

Define $c[i,j] = |\text{LCS}(x[1\ldots i], y[1\ldots j])|$. hen, the length of the LCS for the full sequences x and y is given by $c[m,n] = |\text{LCS}(x,y)|$.

Theorem 5.2.1. *The recursive formula for $c[i,j]$ is as follows:*

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

This approach leverages the optimal substructure property of LCS, which means that an optimal solution for the problem contains optimal solutions to its subproblems.

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

If $z = \text{LCS}(x,y)$, then any prefix of z is an LCS of some prefix of x and some prefix of y.

Algorithm 20 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, when $x[i] \neq y[j]$ at most recursive calls, the algorithm evaluates two subproblems for each pair (i,j). This results in an exponential time complexity $=(2^{m+n})$, as the recursion tree height can reach $m+n$.

However, many subproblems are solved repeatedly, making the recursive approach highly inefficient.

5.2.2 Memoization algorithm

To avoid redundant calculations, we use memoization by storing results of subproblems in a table, allowing subsequent calls to simply retrieve stored values instead of recalculating them.

Definition (*Memoization*)**.** After computing a solution to a subproblem, store it in a table so that future calls can retrieve the result without redoing the work.

Algorithm 21 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
```

The memoization approach has a time complexity of $\Theta(mn)$ since we solve each subproblem only once, and there are mn possible subproblems (one for each pair (i, j) where $1 \leq i \leq m$ and $1 \leq j \leq n$). The space complexity is also $\Theta(mn)$, as we store each subproblem result in a table.

5.2.3 Dynamic programming

The dynamic programming (DP) solution to the LCS problem builds the solution bottom-up, filling out a table from smaller subproblems to larger ones. This avoids the recursive overhead and is efficient for both time and space. In particular, the steps are:

1. *Build the table bottom-up*: create a table c where $c[i, j]$ represents the length of the LCS of prefixes $x[1 \ldots i]$ and $y[1 \ldots j]$. Start from $c[0, 0]$ and fill the table up to $c[m, n]$ based on the recurrence relation from the recursive algorithm.

2. *Backtrack to reconstruct the LCS*: after computing $c[m, n]$, use the table to trace back from $c[m, n]$ to $c[0, 0]$ to reconstruct the LCS.

Algorithm 22 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
```

The DP approach has time complexity $\mathcal{O}(mn)$ because we fill each entry of the $m \times n$ table once. The space complexity is also $=(mn)$, as we store the results in the table. This solution is both efficient and avoids redundant calculations, making it suitable for large inputs.

5.3 Binary Decision Diagram

The core idea behind ROBDDs is to avoid the inefficiencies of sequential sub-case exploration by instead storing sub-cases in memory, allowing for faster retrieval and processing. This approach relies on two crucial hashing mechanisms: a unique table and a computed table. The unique table identifies and consolidates identical sub-cases to prevent redundancy, while the computed table stores the results of previously computed sub-cases to reduce redundant calculations.

ROBDDs represent logic functions as Directed Acyclic Graphs (DAGs), which often provide a more compact form than traditional Sum of Products (SOP) expressions. With careful structuring, ROBDDs can be made canonical, meaning they provide a unique representation of a function. This can shift the focus in Boolean reasoning from solving SAT problems to efficiently managing function representation.

Another significant advantage is the efficiency of Boolean operations on BDDs. Many operations, such as checking for tautology or computing complements, can be performed quickly—often in linear time relative to the result's size or even in constant time. However, the size of a BDD is critically influenced by variable ordering, with the right orderings resulting in significantly smaller, more manageable diagrams.

Directed Acyclic Graph representation In an ROBDD, the logic function is represented by a Directed Acyclic Graph (DAG) with a structure rooted in a single root node and terminating in two terminal nodes, labeled 0 and 1. Each internal node in the graph is associated with a variable and has exactly two children. The DAG's structure is based on a Shannon co-factoring tree, modified to be both reduced and ordered, creating the canonical form known as ROBDD:

- *Reduction*: this process simplifies the graph by eliminating redundancy: if a node has two identical children, it is removed, else if two nodes have identical subgraphs, they are merged into a single node.

- *Ordering*: co-factoring (splitting) variables are processed in a consistent, predefined order across all paths, ensuring that each path from the root to any terminal visits variables in ascending order, typically denoted $x_1 < \cdots < x_n$.

An Ordered Binary Decision Diagram (OBDD) applies only the ordering rule, while a Reduced Ordered Binary Decision Diagram (ROBDD) applies both ordering and reduction. The ROBDD's reduction rules are:

1. If a node's two children are identical, the node is removed, effectively reducing the function to $f = vf + \bar{v}f$.

2. If two nodes have isomorphic graphs, they are replaced by a single instance, so each node uniquely represents a distinct logic function.

The onset of the function represented by an ROBDD can be identified by tracing all paths leading to the 1-terminal node. This set of paths corresponds to a cover of pairwise disjoint cubes, providing an efficient and compact representation of the function without needing to explicitly enumerate every path.

5.3.1 Implementation

The BDD can be implemented via:

- *Unique table*: prevents duplication by ensuring that each node in the BDD is unique. Implemented as a hash table, where each node's properties (key) map to an existing or new node (value).

- *Computed table*: stores results of previously computed operations, avoiding redundant calculations.

BDDs represent a compressed form of the Shannon co-factoring tree:

$$f = vf_v + \bar{v}f_{\bar{v}}$$

where the leaf nodes are constants, either 0 or 1.

To maintain a canonical form, ROBDDs follow three rules (as demonstrated by Bryant in 1986):

1. Unique nodes for constants 0 and 1.

2. Consistent ordering of decision variables along all paths.

3. Use of a hash table that ensures:

$$(\text{node}(f_v) = \text{node}(g_v)) \wedge (\text{node}(\bar{f}_v) = \text{node}(\bar{g}_v)) \implies \text{node}(f) = \text{node}(g)$$

   This ensures uniqueness of node(f) using the unique hash table.

The order of variables is fixed, so if $v < w$, then v appears higher in the ROBDD structure. The top variable associated with the root node of f becomes the key variable for subsequent operations.

5.3.2 If-then-else operator

The ITE operator can implement any two-variable logic function, representing 16 possible operations (all subsets of B^2):

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

Unique hash table Before adding a new node (v, g, h) to the BDD, the unique table is checked to ensure it doesn't duplicate an existing node. If a match exists, the existing node pointer is reused; otherwise, a new node is added to the table. This process maintains the BDD's canonical form, ensuring that a node (v, g, h) exists in the unique table if and only if it has been explicitly added. Thus, there's only one pointer to each unique table entry, supporting multi-rooted Directed Acyclic Graphs (DAGs) for multiple functions.

Algorithm 23 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)
    fn = ITE(f_v, g_v, h_v)
    gn = ITE(f_v̄, g_v̄, h_v̄)
    if fn == gn then
        return gn
    end if
    if !(p = HashLookupComputedTable(v, fn, gn)) then
        p = CreateNode(v, fn, gn)
        Insert p into the unique table
    end if
    key = HashKey(f, g, h)
    InsertComputedTable(p, key)
    return p
end function
```

5.3.3 Computed table

The computed table stores triplets of the form (f, g, h) representing results already computed by the ITE operator. Implemented as a software cache, this table reduces redundant calculations and improves efficiency. To facilitate rapid lookups, the computed table uses a hash table designed as a lossy cache without collision handling chains.

Complemented edges Complemented edges allow the BDD to represent inverted functions efficiently, reducing memory usage and accelerating operations like NOT and ITE. This structure is similar to optimizations in circuit design. However, to maintain the strong canonical form of the BDD, four edge equivalences must be managed, ensuring consistent representation across nodes.

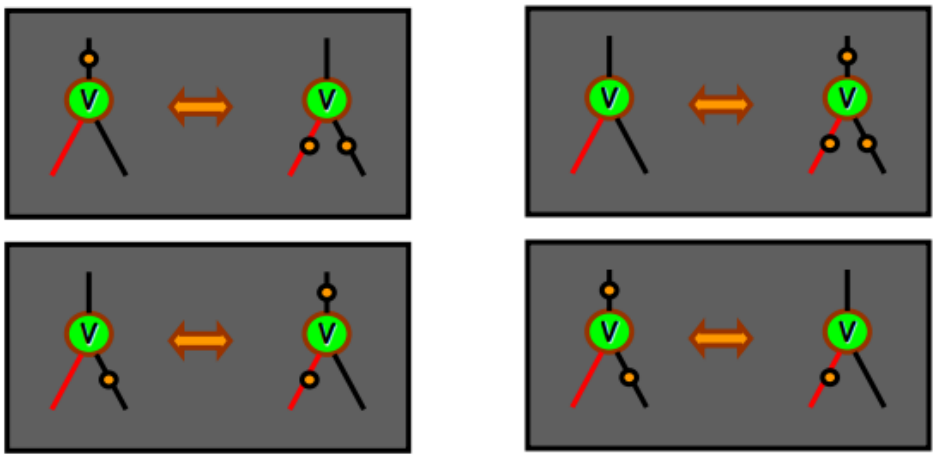


Figure 5.1: Ege equivalences

The preferred convention is to position the complement-free edge on the left (or "then" leg) of the structure.

Optimization To maximize successful matches in the computed table, the following conventions are typically observed:

1. The first argument is chosen based on the smallest top variable to maintain a predictable order.
2. When variables are tied, the smallest address pointer is chosen, though this can sometimes limit portability.

Efficient BDD operations require caching mechanisms:

- *Local cache*: temporary storage for single operations, which is cleared after the operation completes.
- *Operation-specific cache*: dedicated caches for specific operations to reduce the need for type-tracking.
- *Shared cache*: a universal cache for all operations, improving memory management but requiring storage of operation types.

5.3.4 Garbage collection

Effective garbage collection is crucial for managing memory in BDDs, as unused nodes consume resources. BDD nodes can be deallocated by external `bdd_free` operations for nodes that are no longer referenced externally or by reclaiming nodes created temporarily during BDD operations. Mechanisms to Detect Unreferenced Nodes

Unreferenced nodes

1. *Reference counting*: each node maintains a counter that increments when a new reference is created and decrements when a reference is removed. If a counter overflows, the node freezes and remains in memory.

2. *Mark-and-sweep algorithm*: this method, which doesn't rely on reference counts, marks all referenced BDD nodes in a first pass and frees unmarked nodes in a second pass. An external reference handle layer may be needed for accuracy.

Timing Since garbage collection can be resource-intensive, its timing is critical. Options include:

- *Immediate deallocation*: freed nodes are reclaimed right away, though this approach risks reallocation in the next operation.

- *Scheduled collections*: regular garbage collection can be triggered based on statistics gathered during BDD operations.

- *Death row retention*: nodes are temporarily retained before final deletion to improve reuse in subsequent operations.

Because computed tables don't use reference tracking, they must be cleared independently during garbage collection. Sorting freed nodes also improves memory locality, enhancing cache performance and overall efficiency.

CHAPTER 6

Amortized analysis

6.1 Introduction

Amortized analysis is a technique used to evaluate the average cost per operation over a sequence, ensuring that the overall performance remains efficient even if individual operations can be costly. Unlike probabilistic analyses, amortized analysis provides a guarantee on the average cost of each operation, even in the worst case.

The three primary methods of amortized analysis are:

- *Aggregate method*: provides a simple overall average but lacks precision.
- *Accounting method*: uses a banking approach with amortized costs per operation.
- *Potential method*: relies on a potential function to manage the amortized cost.

Hash table resizing A well-designed hash table should balance compactness with sufficient size to minimize overflow and ensure efficient performance. However, determining the optimal size in advance is often impractical. To address this, we use a dynamic table that expands as needed. When the table reaches its capacity, a larger table is allocated, all entries are rehashed into the new table, and the memory from the old table is released. This dynamic resizing allows the hash table to grow as entries are added, maintaining efficiency without predefined size constraints.

6.2 Aggregate method

The amortized cost analysis in the aggregate method provides an average cost per operation over a sequence of operations, even if individual operations can sometimes be costly. This approach is particularly useful for data structures that undergo periodic costly operations because it spreads the cost of these occasional expensive operations across many cheaper ones.

Hash table resizing Although a single insertion might seem costly in the worst-case scenario, with a time complexity of $\mathcal{O}(n)$, this does not mean that n insertions would collectively cost $\mathcal{O}(n^2)$. In practice, the total cost for n insertions remains close to $\mathcal{O}(n)$, resulting in far greater efficiency.

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

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

When $i-1$ is an exact power of 2, the table size doubles, requiring all existing entries to be reinserted. For all other insertions, the cost remains 1.

Thus, the amortized cost per insertion is $\mathcal{O}(1)$, meaning that each insertion, on average, is a constant-time operation. This amortized efficiency allows dynamic hash tables to effectively manage an unpredictable number of entries while ensuring consistent performance.

6.3 Accounting method

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

- *Immediate cost*: this is the actual cost of the operation performed.
- *Banked cost*: any excess cost that is saved and banked for future operations.

The key principle behind the accounting method is that the total accumulated banked cost must never be negative, ensuring that the resources saved are always sufficient to fund future operations. This can be expressed mathematically as:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \qquad \forall n$$

Here, c_i is the true cost of the i-th operation and $\hat{c}_i$ is the amortized cost. By ensuring the bank balance never goes negative, the total amortized cost provides an upper bound on the true total cost of the operations, ensuring efficiency.

Hash table resizing For a dynamic hash table that expands its capacity as needed, we can apply the accounting method to model the cost of insertions and table expansions. In this case, each insertion is charged an amortized cost of $\hat{c}_i = 3$:

- *Immediate cost*: the immediate cost of performing the insertion is 1 unit, which represents the cost of adding an entry to the table.
- *Banked cost*: 2 units is banked for future table expansions, which helps cover the cost of rehashing and moving entries during a table expansion.

When the table doubles in size, the banked cost ensures that the expansion process remains efficient. Specifically:

- 1 unit of the banked cost is used to reinsert the newly added items into the larger table.
- The remaining 1 unit banked cost is used to cover the cost of moving the existing items to the new table.

This approach ensures that the bank balance never falls below zero, and thus the total amortized cost provides an upper bound on the true costs. With each insertion being charged an amortized cost of 3, and with the banked cost effectively covering the expansion costs, the dynamic hash table remains efficient. The amortized cost guarantees that the average cost per operation stays constant over time, ensuring good performance even in the face of resizing operations.

6.4 Potential method

The potential method of amortized analysis views the bank account as the potential energy of a dynamic set of operations. The goal is to use the potential function to account for the work done by an operation and how it affects the overall cost over time.

In this framework, we start with an initial data structure, denoted as D_0. Each operation i transforms this data structure from D_{i-1} to D_i, with a cost of c_i. We now define a potential function Φ that maps each data structure status D_i to a real number:

$$\Phi : \{D_i\} \to \mathbb{R}$$

This functions has the following properties:

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

We can finally define the amortized cost $\hat{c}_i$ for an operation i with respect to the potential function as:

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

Here, $\Delta\Phi_i$ is termed potential difference.

The potential difference may be positive ore negative:

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$, meaning the operation stores work in the data structure for future use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$, meaning the data structure delivers stored work to help pay for the operation.

The total amortized cost over n operations is given by:

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

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, define the potential of the table after the i-th insertion by:

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

We assume that $2^{\lceil \log 0 \rceil} = 0$ (this accounts for the growth of the table as it resizes).

The amortized cost of the i-th insertion is:

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

Here, the true cost c_i of the i-th insertion is given by:

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

For the case in which $i-1$ is an exact power of 2, the amortized cost is:

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

For the second case, the amortized cost is:

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

In both cases, the amortized cost per insertion is 3, and therefore, after n insertions, the total cost is $\Theta(n)$ in the worst case.

6.5 Considerations

Amortized costs offer a powerful abstraction for understanding the performance of data structures over a sequence of operations, smoothing out the effects of occasional expensive operations by focusing on average performance. However, when choosing a method for amortized analysis, it's important to consider the strengths and weaknesses of different approaches.

Any of the amortized analysis methods can be used in different scenarios. The choice of method largely depends on the nature of the operations and the data structure being analyzed. While all methods aim to provide a bound on the total cost, some methods are more intuitive or easier to apply in certain cases. Some analysis methods may be simpler or more precise for specific data structures.

In methods like the accounting and potential methods, there are often multiple valid ways to assign amortized costs or potentials. The choice of how to assign these values can lead to different amortized cost bounds, and sometimes these bounds can vary significantly. In some cases, one scheme may yield a more precise or tighter bound, while another might provide a simpler, more intuitive analysis. Therefore, when applying these methods, it's important to consider the context and the trade-offs between accuracy and simplicity.