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

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

```
1: for $j = 2$ to $n$ do
2:     $key = A[j]$
3:     $i = j - 1$
4:     while $i > 0$ and $A[i] > key$ do
5:         $A[i + 1] = A[i]$
6:         $i = i - 1$
7:     end while
8:     $A[i + 1] = key$
9: 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

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

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

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

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

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

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

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

1.3 Recurrences

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

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

To solve this recurrence we may use three different tecniques:

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

1.3.1 Recursion tree

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

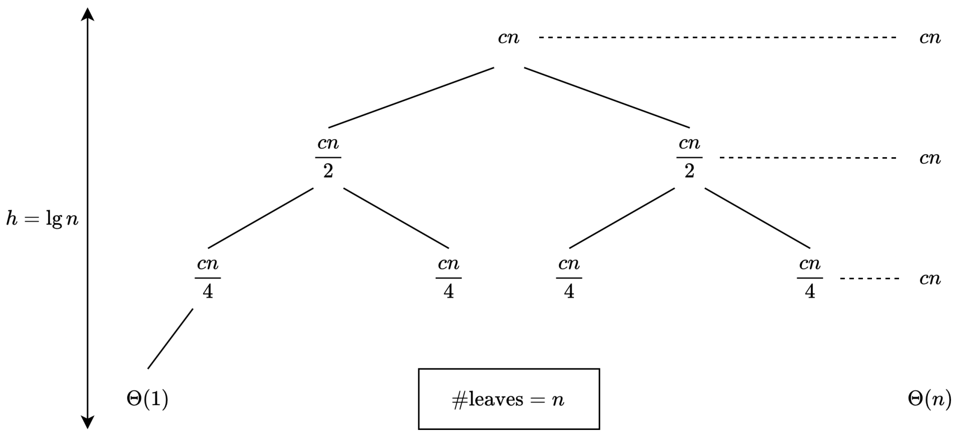


Figure 1.1: Partial recursion tree for merge sort algorithm

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

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

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

1.3.2 Substitution method

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

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

Example:
Consider the expression:

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

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

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

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

1.3.3 Master method

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

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

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

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

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then

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

2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then:

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

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

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

Example:
Let's analyze the expression:

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

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

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

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

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

Now consider the expression:

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

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

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

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

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

Next, consider:

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

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

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

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

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

Finally, consider the expression:

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

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

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

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

CHAPTER 2

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.

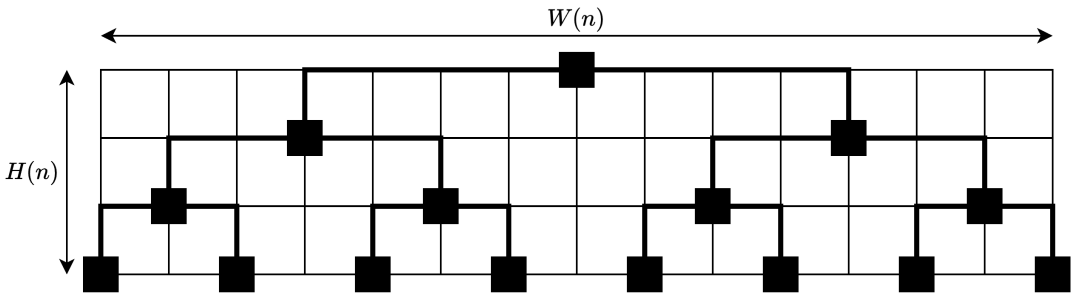


Figure 2.1: VLSI layout problem

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.

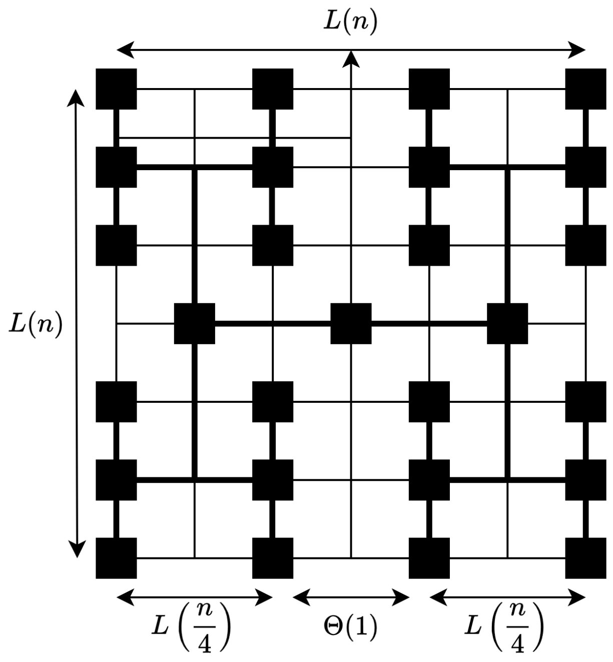


Figure 2.2: VLSI layout problem

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:

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

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.

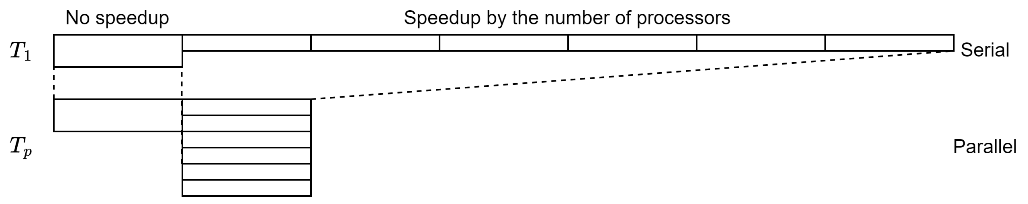


Figure 3.1: Serial and parallel models

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.

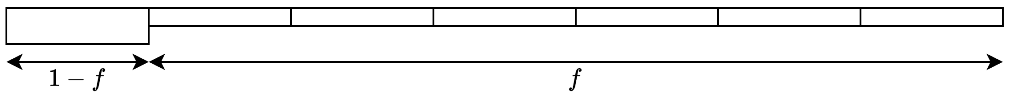


Figure 3.2: Serial model

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

Randomization

4.1 Introduction

Probabilistic analysis In probabilistic analysis, the algorithm is deterministic, meaning that for any given fixed input, the algorithm will always produce the same result and follow the same execution path each time it runs. This analysis assumes a probability distribution for the inputs and the the algorithm is then analyzed over this distribution. In this type of analysis we have to consider that certain specific inputs may result in significantly worse performance. Addittionally, if the assumed distribution of inputs is inaccurate, the analysis may present a misleading or overly optimistic view of the algorithm's behavior.

Random analysis In contrast, randomized algorithms introduce randomness into their execution, which means that, for a fixed input, the outcome may vary depending on the results of internal random decisions. Randomized algorithms generally work well with high probability for any input. However, a small chance that they may fail on any given input, though this probability is low. Key elements of randomized algorithms are:

- *Indicator variables*: to analyze a random variable X, which represents a combination of many random events, we can break it down using indicator variables X_i:

$$X = \sum X_i$$

- *Linearity of expectation*: suppose we have random variables X, Y, and Z, where X is the sum of Y and Z, then:

$$\mathbb{E}[X] = \mathbb{E}[Y + Z] = \mathbb{E}[Y] + \mathbb{E}[Z]$$

  This holds true regardless of whether the variables are independent.

- *Recurrence relations*: these relations describe the behavior of an algorithm in terms of smaller subproblems.

4.1.1 Hiring problem

Imagine you need to hire a new employee, and a headhunter sends you one applicant per day for n days. If an applicant is better than the current employee, you fire the current one and hire the new applicant. Since both hiring and firing are costly, you are interested in minimizing these operations.

We may have two estreme cases:

- *Worst-case scenario*: the headhunter sends applicants in increasing order of quality, meaning each new applicant is better than the previous one. In this case, you hire and fire each applicant, resulting in n hires.
- *Best-case scenario*: the best applicant arrives on the first day, so you hire them and make no further changes. The total cost is just one hire.

In the average case, the input to the hiring problem is a random ordering of n applicants. There are $n!$ possible orderings, and we assume that each is equally likely. We want to compute the expected cost of our hiring algorithm, which in this case is the expected number of hires.

Let $X(s)$ be the random variable representing the number of applicants hired given the input sequence s. To find $\mathbb{E}[X]$, we can break the problem down using indicator random variables X_i:

$$X_i = \begin{cases} 1 & \text{if applicant } i \text{ is hired} \\ 0 & \text{otherwise} \end{cases}$$

The total number of hires X is the sum of these indicator variables:

$$X = X_1 + X_2 + \cdots + X_n$$

Now, using the linearity of expectation, we have:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

Next, we need to compute $\mathbb{E}[X_i]$. An applicant i is hired only if they are better than all previous $i-1$ applicants. For a uniformly random order of applicants, the probability that applicant i is better than the previous $i-1$ applicants is $\frac{1}{i}$. Thus, the expected value for each X_i is:

$$\mathbb{E}[X_i] = \Pr(\text{applicant } i \text{ is hired}) = \frac{1}{i}$$

Finally, the expected total number of hires is:

$$\mathbb{E}[X] = \sum_{i=1}^{n} \frac{1}{i}$$

This sum is the harmonic series, which is bounded by $\ln n + 1$. Therefore, the average number of hires is approximately $\ln n$.

Analysis The analysis above assumes that the headhunter sends applicants in a random order. However, if the headhunter is biased you cannot rely on this randomness. In such cases, if you have access to the entire list of applicants in advance, you can take control by randomizing the input yourself. By randomly permuting the list of applicants before interviewing them, you essentially convert the hiring problem into a randomized algorithm. This way, the hiring process no longer depends on the headhunter's input order, and you maintain the same expected number of hires, $\mathcal{O}(\log n)$, regardless of the original order. In general, randomized algorithms allow for multiple possible executions on the same input, which ensures that no single input can guarantee worst-case performance. Instead of assuming some distribution for the inputs, you actively create your own distribution, thereby moving from passive probabilistic analysis to a more robust, actively randomized approach.

4.1.2 Randomized algorithms

Randomized algorithms can be broadly classified into two main types:

- *Las Vegas algorithms*: these algorithms enures the correctness of the output (randomness affects only the running time).
- *Monte Carlo algorithms*: these algorithms return an incorrect solution with a known probability (randomness affects both running time and 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 $\delta(S) = \{(u, v) \in E : u \in S, v \in S'\}$ represents a cut. The goal of the minimum cut problem is to find the cut with the fewest number of edges.

Source-target cut problem In graph theory, a source-target cut refers to a way of partitioning the vertices of a graph into two disjoint subsets such that: one subset contains a designated source vertex s, and the other subset contains a designated target vertex t. In this version, for specified vertices $s \in V$ and $t \in V$, we restrict attention to cuts $\delta(S)$ where $s \in S$ and $t \notin S$. The goal here is to find the cut that minimizes the number of edges crossing between S and S'.

Traditional solution Traditionally, the minimum cut problem could be solved by computing $n - 1$ minimum source-target cuts, one for each pair of vertices. In the minimum source-target cut problem, we are given two vertices s and t, and the goal is to find the set S such that $s \in S$ and $t \notin S$, minimizing minimizing $|\delta(S)|$. By linear programming duality, the size of the minimum source-target cut is equal to the value of the maximum flow. The fastest known algorithm for solving the maximum flow problem runs in time:

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

4.2.1 Karger's algorithm

Karger introduced a randomized algorithm to solve the minimum cut problem, which avoids the need for maximum flow computations. This approach relies on randomly contracting edges to shrink the graph while preserving the minimum cut with high probability.

Multigraphs In a multigraph, multiple edges can exist between any pair of vertices. However, there are no edges of the form (v, v), known as self-loops.

Edge contraction The contraction of an edge $e = (u, v)$ merges the vertices u and v into a single vertex. In the contracted graph the vertices u and v are replaced by a new vertex, denoted w.

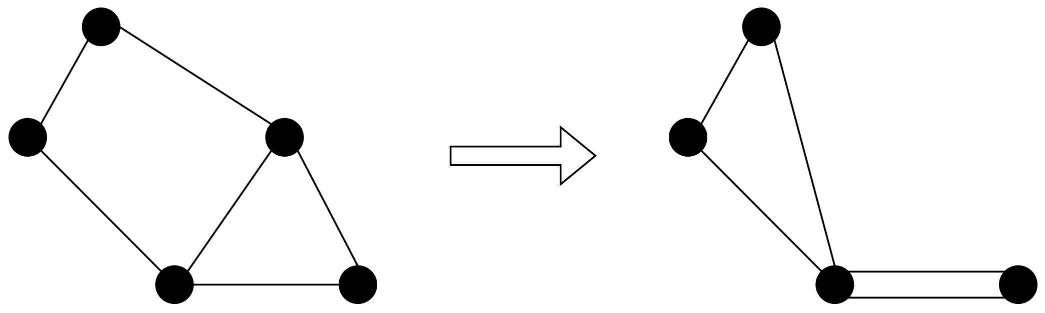


Figure 4.1: Edge contraction

Let $G = (V, E)$ be a multigraph without self-loops, and let $e = \{u, v\} \in E$. Formally, the contraction of e, denoted $G \setminus e$, is formed by:

1. Replacing vertices u and v with a new vertex w.
2. Replacing all edges (u, x) or (v, x) with an edge (w, x).
3. Removing any self-loops involving w.

After contraction, the graph $G \setminus e$ is still a multigraph. A crucial observation is that contracting an edge (u, v) preserves cuts where both u and v belong either to the same set S or to its complement S'. For a cut $\delta(S)$ in the original graph G, if $u, v \in S$, then after contraction, $\delta_G(S) = \delta_{G\setminus e}(S)$, where w is substituted for u and v.

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

1. Pick an edge uniformly at random and contract the two vertices at its endpoints.
2. Repeat the contraction process until only two vertices remain (i.e., after $n-2$ edges have been contracted).

The two remaining vertices represent a partition (S, S') of the original graph, and the edges between them correspond to the cut $\delta(\mathrm{S})$ in the input graph.

Let k be the size of the minimum cut. Focus on a particular minimum cut $\delta(S)$ where $|\delta(S)| = k$

Lemma 4.2.1. *Let $\delta(S)$ be a minimum cut in the graph $G = (V, E)$. The probability that Karger's algorithm outputs the cut $\delta(S)$ is at least:*

$$\Pr(\textit{Karger's algorithm ends with the cut } \delta(S)) \geq \frac{1}{\binom{n}{2}}$$

To increase the probability of success, we run the algorithm $l \cdot \binom{n}{2}$ times. The probability that at least one run succeeds is:

$$\Pr(\text{one success}) = 1 - \left(1 - \frac{1}{\binom{n}{2}}\right)^{l \cdot \binom{n}{2}} \geq 1 - e^{l}$$

By setting $l = c \log n$, the error probability becomes $\frac{1}{n^c}$.

Complexity One run of Karger's algorithm takes $\mathcal{O}(n^2)$ time. Hence, 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 $\frac{1}{\text{poly}(n)}$.

4.2.2 Karger and Stein algorithm

Karger and Stein introduced a more efficient version of Karger's original algorithm by refining its edge contraction process. The key insight comes from analyzing the telescoping product that emerges when calculating the probability of contracting an edge from the minimum cut set $\delta(S)$.

In the early stages of the contraction, it is very unlikely that an edge from the minimum cut is contracted. However, as the graph shrinks, the likelihood of contracting such an edge increases. The probability that a fixed minimum cut $\delta(S)$ survives down to a smaller subgraph with l vertices is at least:

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

By choosing $l = \frac{n}{\sqrt{2}}$, we ensure that the probability of retaining the minimum cut is at least $\frac{1}{2}$. This means that, in expectation, two trials of the algorithm should suffice to find the minimum cut with high probability.

Algorithm The steps of Karger and Stein algorithm are:

1. From a multigraph G with at least six vertices, repeat the following twice:

   (a) Run the original edge contraction algorithm until the graph is reduced to $\frac{n}{\sqrt{2}} + 1$ vertices.

   (b) Recur on the resulting contracted graph.

2. Return the minimum of the cuts found in the two recursive calls.

The choice of six as the threshold for recursion depth does not impact the overall asymptotic complexity, but only affects the running time by a constant factor.

Algorithm 7 Karger and Stein

```
 1: function CONTRACT($G = (V, E), t$)
 2:     while $|V| > t$ do
 3:         Choose $e \notin E$ uniformly at random
 4:         $G = G \setminus e$
 5:     end while
 6:     return $G$
 7: end function

 8: function FASTMINCUT($G = (V, E)$)
 9:     if $|V| < 6$ then
10:         return mincut($V$)
11:     else
12:         $t = \left\lceil 1 + \frac{|V|}{\sqrt{2}} \right\rceil$
13:         $G_1 = $ CONTRACT($G, t$)
14:         $G_2 = $ CONTRACT($G, t$)
15:         return min{FASTMINCUT($G_1$), FASTMINCUT($G_2$)}
16:     end if
17: end function
```

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

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

Which solves to:

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

The algorithm succeeds with probability at least $\geq \frac{1}{2}$ at each recursive step. To boost the probability of success, we repeat the algorithm multiple times. Specifically, the recurrence for the success probability $\Pr(n)$ is:

$$\Pr(n) \geq 1 - \left(1 - \frac{1}{2}\Pr\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

This implies that:

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

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.*

Like the original Karger's algorithm, the Karger-Stein algorithm is a Monte Carlo algorithm. This means that it guarantees a correct solution with high probability, but there remains a small probability of failure.

4.3 Sorting problem

The sorting problem is a fundamental computational task that involves arranging a collection of elements in a specific order, typically ascending or descending. The input is an unsorted list or array, and the goal is to rearrange the elements such that they follow a predefined sequence based on some criteria.

4.3.1 Quicksort

Quicksort is a highly efficient divide-and-conquer sorting algorithm proposed by Hoare in 1962. It is widely used due to its practical performance and ability to sort in-place, meaning it requires only a small, constant amount of extra storage space. With proper tuning, Quicksort outperforms many other algorithms in practical applications.

1. *Divide*: select a pivot element from the array and partition the array into two subarrays:
    - Elements in the lower subarray are less than or equal to the pivot.
    - Elements in the upper subarray are greater than or equal to the pivot.
2. *Conquer*: recursively apply quicksort to each of the two subarrays.
3. *Combine*: since the subarrays are already sorted, no additional work is needed to combine them.

The key to quicksort's efficiency is the linear-time partitioning subroutine, which runs in $\mathcal{O}(n)$ time.

Given an array $A[p \cdots q]$, the goal is to select a pivot element x and rearrange the array so that all elements less than x appear before it, and all elements greater than or equal to x appear after it.

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 $A[j] \leq x$ then
 6:             $i = i + 1$
 7:             exchange $A[i] \leftrightarrow A[j]$
 8:         end if
 9:     end for
10:     exchange $A[p] \leftrightarrow A[i]$
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
```

Worst case complexity The worst case occurs when the pivot element always ends up at one of the ends of the array, resulting in highly unbalanced partitions. In this case, the recurrence relation for the running time is:

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

This leads to the worst-case time complexity of $\Theta(n^2)$.

Best case complexity In the best case, the partitioning process splits the array into two even halves. The recurrence relation for the best case is:

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

Solving this recurrence gives a time complexity of $\Theta(n \log n)$, which is typical for efficient sorting algorithms.

Average case complexity Even though the worst-case scenario yields $\Theta(n^2)$, quicksort's average-case time complexity is $\Theta(n \log n)$. This happens because, on average, the pivot tends to split the array into reasonably balanced parts, making the overall performance efficient for large datasets.

Unbalanced case Complexity If the partitioning process always splits the array in a highly unbalanced way, such as 10% and 90% split, the recurrence is:

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

Even in such cases, the overall time complexity remains $\Theta(n \lg n)$, although the constant factors may differ.

In practice, specialized partitioning strategies are used to efficiently handle cases where there are duplicate elements in the input array. The performance of quicksort is highly dependent on the choice of the pivot. Common strategies include selecting the first element, the last element, or the median-of-three (choosing the median of the first, middle, and last elements).

4.3.2 Randomized quicksort

Randomized quicksort is an enhanced version of the classic quicksort algorithm. It improves performance by ensuring that the pivot is selected randomly, thus minimizing the likelihood of worst-case behavior. This approach ensures that the running time is not dependent on the input's structure. The running time is independent of the input's initial order, avoiding worst-case scenarios caused by already sorted or reverse-ordered data.

Suppose we alternate between lucky and unlucky partitioning scenarios:

- *Lucky scenario*: the pivot divides the array evenly, making recursive calls on approximately half the array each time:

$$L(n) = 2U\left(\frac{n}{2}\right) + Q(n)$$

- *Unlucky scenario*: the pivot divides the array in a highly unbalanced way, for example, making a recursive call on almost the entire array:

$$U(n) = L(n-1) + Q(n)$$

Solving this system gives us:

$$L(n) = 2\left(L\left(\frac{n}{2} - 1\right) + Q\left(\frac{n}{2}\right)\right) + Q(n)$$

This simplifies to:

$$L(n) = Q(n \log n)$$

Thus, the randomized pivot selection helps ensure that the algorithm performs efficiently in expectation, typically resulting in a time complexity of $\mathcal{O}(n \log n)$.

Analysis Let $T(n)$ be the random variable for the running time of randomized Quicksort on an input of size n, assuming that the random numbers are independent. For $k = 0, 1, \ldots, n-1$, define an indicator random variable X_k, where:

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

The expected value of X_k is:

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

This is because all splits are equally likely, assuming the elements are distinct. Therefore, the expected running time of the algorithm can be written as:

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

Breaking this down:

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

This simplifies to:

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

In this case we have:

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

For large enough n, where $n \geq 2$, we find that:

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

Thus, for a sufficiently large constant a, the $\Theta(n)$ term is dominated, leading to an overall time complexity of $\mathcal{O}(n\log n)$.

Randomized quicksort is generally more than twice as fast as merge sort in practice. Quicksort's performance can be improved significantly with careful code tuning and optimized implementations. It behaves well with caching and virtual memory, making it suitable for use in systems with large datasets or hierarchical memory structures.

4.3.3 Comparison sort analysis

All the sorting algorithms discussed so far belong to the class of comparison sorts. In these algorithms, sorting is achieved by performing pairwise comparisons between elements to determine their relative order. The best worst-case time complexity we have seen for these comparison-based sorting algorithms is $\mathcal{O}(n\log n)$.

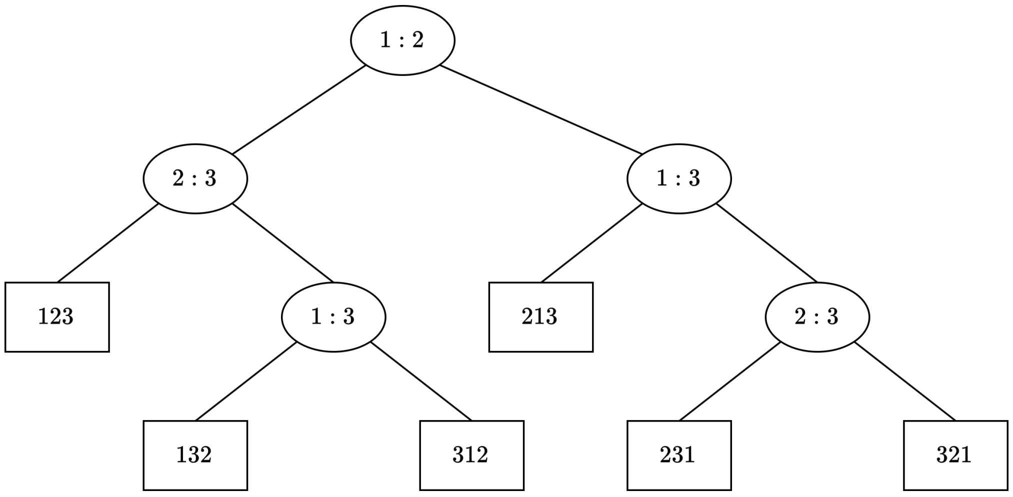
To understand why $\mathcal{O}(n\log n)$ is the best achievable worst-case performance for comparison sorts, we can model sorting algorithms using decision trees.

Consider sorting an array $\langle a_1, a_2, \ldots, a_n\rangle$. We represent each comparison made by the sorting algorithm as a node in a decision tree. Each internal node in this tree corresponds to a comparison between two elements a_i and a_j from the array. The tree has the following properties:

- Each internal node is labeled with a pair i,j (indices of elements being compared).
- The left child represents the path taken if $a_i \leq a_j$, while the right child represents the path if $a_i > a_j$.

Each leaf node in the tree represents one of the possible final sorted orders (permutations) of the array. The entire tree represents all possible sequences of comparisons that can occur for different input arrays of size n.

Example:
Consider sorting the array $\langle 9, 4, 6\rangle$. The corresponding decision tree might look like the following:



This decision tree helps visualize the comparisons made to sort the array, with each path from the root to a leaf representing a specific sequence of comparisons.

Each leaf in the decision tree represents a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n)\rangle$, indicating the established sorted order $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$.

The height of the decision tree corresponds to the worst-case running time of the algorithm, as it reflects the maximum number of comparisons made during the sorting process. The worst-case running time is the length of the longest path from the root to a leaf node, which is the height of the tree.

Since there are $n!$ possible ways to order n distinct elements, the decision tree must have at least $n!$ leaves. A binary tree of height h has at most 2^h leaves. Therefore, we must have $2^h \ge n!$. Taking the logarithm on both sides:

$$h \ge \log n!$$

Applying Stirling's approximation $n! \approx \left(\frac{n}{e}\right)^n$, we have:

$$\log n! \ge n \log n - n \log e$$

Thus, the height of the decision tree (and hence the worst-case running time of any comparison sort) is at least:

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

This proves that any comparison-based sorting algorithm must have a worst-case time complexity of $\Omega(n \log n)$, which is a lower bound for all comparison sorts.

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

Corollary 4.3.1.1. *Since heapsort and merg esort both achieve a worst-case time complexity of* $\mathcal{O}(n \log n)$, *they are asymptotically optimal comparison sorting algorithms.*

4.3.4 Counting sort

Counting sort is a non-comparison-based sorting algorithm. It leverages the range of possible values in the input to efficiently organize the elements.

This algorithm takes as input an array $A[n]$, where each element $A[j] \in \{1, 2, \ldots, k\}$, and returns a sorted array $B[n]$. It also needs an auxiliary array $C[n]$.

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 $\Theta(k)$
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 $\Theta(n)$
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 $\Theta(k)$
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 $\Theta(n)$
```

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

1. *Initial state*: we initialize the array C to all zeros: $C = \langle 0, 0, 0, 0 \rangle$.
2. *Count elements*: we count the occurrences of each element in A, resulting in: $C = \langle 1, 0, 2, 2 \rangle$.
3. *Compute the prefix sum*: we 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, we 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 $C = \langle 0, 1, 1, 3 \rangle$.

We have that the complexity of the algorithm is the sum of the complexities of the four loops:

$$\Theta(k) + \Theta(n) + \Theta(k) + \Theta(n)$$

Thus, the total time complexity of counting sort is:

$$\Theta(n + k)$$

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

Stability Counting sort is a stable sort, meaning it preserves the relative order of equal elements from the input array. This stability can be particularly important in scenarios where sorting must maintain additional orderings or properties of the data.

4.3.5 Radix sort

Radix sort is a classic algorithm with historical significance, dating back to Herman Hollerith's card-sorting machine developed for the 1890 U.S. Census. The core concept of radix sort is to sort numbers digit by digit, which can be efficient for large inputs under certain conditions.

The original method proposed by Hollerith was to sort starting with the most significant digit first, but this approach turned out to be inefficient. Instead, the more effective strategy is to sort based on the least significant digit first, using an auxiliary stable sort like counting sort.

Correctness To ensure correctness assume that the numbers have already been sorted by their least significant $t-1$ digits. Now sort the numbers by the t-th digit:

- Two numbers that differ in the t-th digit will be correctly ordered.
- Two numbers that are identical in the t-th digit will retain their original relative order (thanks to the stability of the auxiliary sort).

This results in the numbers being in the correct final order.

Performance analysis To analyze the efficiency of radix sort, assume we are using counting sort as the stable auxiliary sorting method. Suppose we are sorting n computer words, each consisting of b bits. Each word can be viewed as having $\frac{b}{r}$ digits, where each digit is based on 2^r.

The time complexity of each pass of counting sort is $\Theta(n+2^r)$, since we process n numbers and there are 2^r possible values for each digit. The total number of passes is $\frac{b}{r}$, so the overall time complexity is given by:

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

To minimize the time complexity, we must choose an optimal r. Increasing r reduces the number of passes (since $\frac{b}{r}$ becomes smaller), but it also increases 2^r, which can inflate the running time. For efficiency, we should avoid letting 2^r become larger than n, as this leads to exponential growth in time. Thus, an optimal choice is to set $r = \log n$, which balances the number of passes and the cost of each pass. With this choice, the time complexity becomes:

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

For numbers in the range from 0 to $n^d - 1$, where $b = d\log n$, the radix sort runs in $\Theta(dn)$ time.

In practice, radix sort is particularly fast for large datasets and is simple to implement. However, it exhibits poor locality of reference compared to algorithms like quicksort, which can cause performance issues on modern processors with complex memory hierarchies. As a result, a well-optimized quicksort may outperform radix sort in environments where memory access speed is crucial.

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. In other words, the goal is to find the element with rank i in the sorted order of the set A. We may have three extreme cases:

- $i = 1$: the minimum element in the set.
- $i = n$: the maximum element in the set.
- $i = \lfloor \frac{n+1}{2} \rfloor$ or $\lceil \frac{n+1}{2} \rceil$: the lower or upper median of the set.

A straightforward solution is to first sort the array and then return the i-th element. The steps are:

1. Sort the array A using a comparison-based sorting algorithm like merge sort or heapsort.
2. Return the i-th element from the sorted array.

The worst-case running time for this approach is:

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

Sorting takes $\Theta(n \log n)$ and selecting the i-th element is a constant-time operation.

However, it is possible to solve the selection problem in linear time. There are two main approaches:

- *Randomized algorithm*: this algorithm, based on quickselect, has an average-case time complexity of $\mathcal{O}(n)$. The idea is to use a partitioning method similar to quicksort, but only recurse on the side of the array that contains the desired element.
- *BFPTRT algorithm*: this approach uses a carefully chosen median of medians strategy to guarantee that each partition reduces the problem size by a constant fraction. This deterministic algorithm runs in $\mathcal{O}(n)$ in the worst case.

In both versions, we efficiently select the i-th smallest element without needing to fully sort the array, making them optimal for large inputs when sorting would be unnecessarily slow. The deterministic linear-time algorithm is particularly notable for its worst-case guarantee, which can be critical in scenarios requiring predictable performance.

4.4.1 Minimum and maximum algorithms

To determine the minimum (or maximum) of a set of n elements, the number of comparisons required is $n - 1$. The following algorithm finds the minimum element:

Algorithm 10 Minimum and maximum

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

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

This algorithm performs exactly $n - 1$ comparisons, making it optimal for finding the minimum. The same number of comparisons is required to find the maximum element.

Maximum and minimum simultaneously When trying to determine both the minimum and maximum of a set of n elements, a simple approach would be to run two separate passes over the array, one for the minimum and one for the maximum, resulting in $2n-2$ comparisons. However, we can do better.

Algorithm 11 Minimum and maximum algorithm

```
if length($A$) is odd then
    max = min = $A[1]$
    $i = 2$
else if $A[1] < A[2]$ then
    min = $A[1]$
    max = $A[2]$
    $i = 3$
else
    min = $A[2]$
    max = $A[1]$
    $i = 3$
end if
while $i \leq$ length($A$) do
    if $A[i] < A[i+1]$ then
        if min > $A[i]$ then
            min = $A[i]$
        end if
        if max < $A[i+1]$ then
            max = $A[i+1]$
        end if
    else
        if min > $A[i+1]$ then
            min = $A[i+1]$
        end if
        if max < $A[i]$ then
            max = $A[i]$
        end if
    end if
    $i = i + 2$
end while
```

Thus, the total number of comparisons required to find both the minimum and maximum is always fewer than $3\left\lfloor\frac{n}{2}\right\rfloor$.

4.4.2 Randomized algorithm

The randomized selection algorithm is an efficient divide-and-conquer algorithm that selects the i-th smallest element from an unsorted array in expected linear time. It is based on the idea of using a random pivot to partition the array, similarly to randomized quicksort, but recursing only on the side of the partition that contains the desired element.

Algorithm 12 Randomized selection

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

The running time of the algorithm depends on the random choices made during partitioning. The time complexity can vary between the best-case and worst-case scenarios:

- *Best case*: if the partition always divides the array evenly, the recurrence relation is:

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

  Solving this gives $T(n) = \Theta(n)$, indicating that in the best case, the algorithm runs in linear time.

- *Worst case*: if the partition always picks the smallest or largest element, the recurrence relation becomes:

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

  This leads to a time complexity of $T(n) = \Theta(n^2)$, making the worst-case performance quadratic.

In practice, the algorithm performs well on average, and its expected time complexity can be shown to be linear. The analysis follows a similar approach to the analysis of randomized quicksort but with subtle differences. The key idea is that the random pivot choice results in a balanced partition with high probability.

To analyze the expected time complexity, let $T(n)$ be the random variable representing the running time of RAND-SELECT on an input of size n. Define an indicator random variable X_k as:

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

The expected running time is then expressed as:

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

Taking expectations, we get:

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

By carefully bounding the expectation, we can show that:

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

For constant $c > 0$ leading to the conclusion that the expected running time is linear, $\Theta(n)$, when c is chosen large enough.

The algorithm is highly efficient in practice, often outperforming deterministic algorithms for selection due to its linear expected running time. The worst case occurs when the partition is unbalanced, leading to $\Theta(n^2)$ performance. However, this is rare, and the average performance is much better.

4.4.3 BFPTRT algorithm

The BFPTRT algorithm, also known as the median of medians algorithm, is a deterministic selection algorithm that guarantees a worst-case linear time complexity $\Theta(n)$ for selecting the i-th smallest element from an unsorted array. Unlike randomized algorithms, it provides a worst-case time guarantee and avoids the risk of quadratic behavior.

Algorithm 13 Blum, Floyd, Pratt, Rivest, and Tarjan

```
1: function SELECT($i, n$)
2:     Divide the array in 5 elements groups (each with the median)    ▷ Complexity $\Theta(n)$
3:     Recursively select the median $x$ of the groups medians as pivot    ▷ Complexity $T\left(\frac{n}{5}\right)$
4:     Partition around the pivot $x$, and let $k = \text{rank}(x)$    ▷ Complexity $\Theta(n)$
5:     if $i = k$ then    ▷ Complexity $T\left(\frac{3n}{4}\right)$
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
```

To understand the efficiency of the algorithm, observe that at least half of the group medians are less than or equal to x, the selected median of medians. Since each group has 5 elements, at least $\left\lfloor \frac{n}{10} \right\rfloor$ elements are less than or equal to x. Similarly, at least $\left\lfloor \frac{n}{10} \right\rfloor$ elements are greater than or equal to x. Thus, at least $\left\lfloor \frac{3n}{10} \right\rfloor$ elements are discarded after each partition.

Therefore, in the worst case, the recursive call to SELECT operates on at most $\left\lfloor \frac{3n}{4} \right\rfloor$ elements, leading to the recurrence relation:

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

This recurrence can be solved using the substitution method to show that $T(n) \leq cn$, meaning that the algorithm runs in linear time.

While the BFPTRT algorithm has excellent theoretical performance, it is often not as fast in practice due to the large constant factors involved in partitioning and recursively finding the median of medians. Specifically:

- *Work per level*: the work done at each level of recursion is a constant fraction smaller than the previous level (roughly $\frac{19}{20}$ of the work at the root). This means that while the algorithm is linear, the constant factors are non-trivial.
- *Efficiency in practice*: the randomized selection algorithm tends to perform better in practice due to its smaller constant factors, despite its worst-case performance of $\Theta(n^2)$.

The BFPTRT algorithm provides a strong worst-case guarantee of linear time for selection problems, making it valuable in scenarios where predictable performance is crucial. However, for most practical purposes, especially with large datasets, the randomized selection algorithm is often preferred due to its faster performance in practice despite its potential for worst-case behavior.

4.5 Primality problem

Definition (*Prime number*)**.** An integer $p \geq 2$ is called prime if and only if:

$$(a|p \to a = 1 \text{ or } a = p) \qquad \forall a \in \mathbb{Z}$$

A straightforward algorithm to check if a number n is prime involves testing its divisibility by every integer up to $\sqrt{n}$. Here is the naive deterministic primality test:

Algorithm 14 Naive primality test

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

The time complexity of this naive algorithm is $\mathcal{O}(\sqrt{n})$, which can be inefficient for very large numbers.

4.5.1 Randomized primality tests

To achieve a more efficient solution, we can use a randomized primality test. These tests provide a faster answer but may return false positives with a small probability. However, we can make the probability of error arbitrarily small by repeating the test multiple times. The core idea behind these tests stems from Fermat's Little Theorem.

Theorem 4.5.1 (Fermat)**.** *For any prime number p and integer a, $0 < a < p$:*

$$a^{p-1} \mod p = 1$$

This forms the basis of a simple randomized primality test:

Algorithm 15 Fermat's primality test

```
1: Calculate $z = 2^{n-1} \mod n$
2: if $z = 1$ then
3:     $n$ is possibly prime
4: else
5:     $n$ is composite
6: end if
```

The advantage of this approach is that it runs in polynomial time, with the power computation being performed efficiently using modular exponentiation.

Despite the efficiency, Fermat's test can incorrectly identify some composite numbers as prime. These numbers are known as pseudoprimes.

Definition (*Pseudo-prime*)**.** A composite number n is called a pseudoprime to base a if:

$$a^{n-1} \mod n = 1$$

Definition (*Base two pseudo-prime*)**.** A composite number $n \geq 2$ is a base-2 pseudoprime if:

$$2^{n-1} \mod n = 1$$

Additionally, a more problematic class of numbers are the Carmichael numbers, which behave like primes under Fermat's test for all a coprime with n.

Definition (*Carmichael numbers*)**.** A composite number $n \geq 2$ is a Carmichael number if for every a such that $\text{gdc}(a, n) = 1$, it holds that:

$$a^{n-1} \mod n = 1$$

Algorithm 16 Randomized primality test

```
1: Randomly choose $a \in [2, n-1]$
2: Calculate $a^{n-1} \mod n$
3: if $a^{n-1} \mod n = 1$ then
4:     $n$ is possibly prime
5: else
6:     $n$ is composite
7: end if
```

Miller-Rabin primality test The Miller-Rabin test improves upon Fermat's test by also checking for non-trivial square roots of 1 modulo n. This test significantly reduces the chances of false positives.

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

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

Here is the Miller-Rabin randomized primality test:

Algorithm 17 Miller-Rabin randomized primality test

```
function POWER($a, p, n$)
    if $p = 0$ then                                          ▷ Compute $a^p \mod n$
        return 1;
    end if
    $x =$ POWER($a, \frac{p}{2}, n$)
    $res = (x \cdot x)\%n$
    if $res = 1$ and $x \neq 1$ and $x \neq n - 1$ then     ▷ check $x^2 \mod n = 1$ and $x \neq 1, n - 1$
        $isProbablyPrime$ = false
    end if
    if $p\%2 = 1$ then
        $res = (a \cdot res)\%n$
    end if
    return $res$
end function

function PRIMALITYTEST($n$)
    $a = random(2, n - 1)$
    $isProbablyPrime$ = true
    $result =$ POWER($a, n - 1, n$)
    if $res \neq 1$ or $!isProbablyPrime$ then
        return false
    else
        return true
    end if
end function
```

The time complexity of the Miller-Rabin test is $\mathcal{O}(\log^2 n)$, with a small probability of error. Repeating the test k times reduces the probability that a composite number is incorrectly classified as prime to $\left(\frac{1}{4}\right)^k$.

Theorem 4.5.2. *If p prime and $0 < a < p$, then the only solutions to the equation $a^2 \mod p = 1$ are $a = 1$ and $a = p - 1$.*

Theorem 4.5.3. *If n is composite, the probability that the Miller-Rabin test mistakenly identifies n as prime for a randomly chosen base a is at most $\frac{n-9}{4}$.*

4.5.2 Applications to cryptosystems

Traditional encryption methods rely on secret keys shared between participants. While this method provides efficient encryption and decryption, it has several notable drawbacks:

- *Key exchange*: the secret key k must be exchanged between two parties (A and B) before any secure communication can take place.
- *Scalability issues*: for secure communication between n participants, $\frac{n(n-1)}{2}$ distinct keys are required, which becomes cumbersome as the number of participants increases.

The advantage of this approach lies in the efficiency: both encryption and decryption operations can be computed very efficiently using secret key cryptosystems.

Modern electronic communication requires additional guarantees to ensure security. These include: confidentiality, integrity, authenticity, and non-repudiation.

Diffie-Hellman key exchange The Diffie-Hellman protocol introduced the idea of using public and private key pairs to resolve the scalability and key exchange issues in traditional encryption. Each participant, say A, has a pair of keys P_A and S_A which are public and private, respectively. Let D represent the set of all legal messages. The key pair P_A and S_A are functions from $D \to D$ with the following properties:

- Both keys can be computed efficiently.
- The keys are inverses.
- The private key cannot be feasibly computed from the public key with realistic computational effort.

The exchange of a message with the Diffie-Hellman protocol unfolds in the following steps:

1. *Key access*: participant A retrieves B's public key P_B (from a public directory or directly from B).
2. *Encryption*: A encrypts a message M using B's public key to obtain the ciphertext C:
$$C = P_B(M)$$
A then sends c to B.
3. *Decryption*: after receiving the encrypted message C, B decrypts it using their private key S_B to retrieve the original message:
$$M = S_B(C)$$

Digital signatures Digital signatures are used to authenticate the sender of a message. To send a digitally signed message M' to B, participant A follows these steps:

1. *Signature generation*: A computes the digital signature σ for message M' using their private key:
$$\sigma = S_A(M')$$
2. *Message transmission*: A sends the pair (M', σ) to B.
3. *Signature verification*: after receiving (M', σ), B verifies the digital signature using A's public key:
$$P_A(\sigma) = M'$$
Anyone with access to P_A can check σ, ensuring the authenticity of the sender.

RSA cryptosystem The RSA algorithm, developed by R. Rivest, A. Shamir, and L. Adleman, is one of the most well-known public-key cryptosystems. The key generation process in RSA is as follows:

1. *Prime selection*: randomly select two large prime numbers p and q, each with at least $l+1$ bits (where $l \geq 500$).
2. *Modulus calculation*: compute $n = p \cdot q$.
3. *Public exponent*: choose an integer e such that e is relatively prime to $(p-1)(q-1)$, i.e., $\text{gdc}(e,(p-1)(q-1)) = 1$
4. *Private exponent*: compute the multiplicative inverse of e modulo $(p-1)(q-1)$, denoted by d, using the extended Euclidean algorithm. This satisfies:
$$d \cdot e \equiv 1 \mod (p-1)(q-1)$$
5. *Key publication*: the public key is $P = (e, n)$, while the private key is $S = (d, n)$.

To encrypt a message M using RSA, the message is divided into blocks, each represented as a binary number where $0 \leq M < 2^{2l}$. The encryption and decryption processes are as follows:

- *Encryption*: the sender computes the ciphertext C using the public key P as:$C = M^e \mod n$.
- *Decryption*: the receiver decrypts the ciphertext C using the private key S as: $M = C^d \mod n$.

To compute the multiplicative inverse in RSA, we use the extended Euclidean algorithm. This algorithm finds integers x and y such that $ax + by = \gcd(a, b)$, where $a = (p-1)(q-1)$ and $b = e$. The pseudocode for the extended Euclidean algorithm is as follows:

Algorithm 18 Extended Euclidean Algorithm

```
function EXTENDED-EUCLID($a, b$)
    if $b = 0$ then
        return $(a, 1, 0)$
    end if
    $(d, x', y')$ = EXTENDED-EUCLID($b, a \mod b$)
    $x = y'$
    $y = x' - \lfloor \frac{a}{b} \rfloor y'$
    return $(d, x, y)$
end function
```

This algorithm guarantees that d and e satisfy $d \cdot e \equiv 1 \mod (p-1)(q-1)$.

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. This is analogous to maintaining a database where entries may be added, removed, or searched periodically.

Given a universe U of keys with a total order, the task is to manage a set $S \subseteq U$ and support the following operations:

- *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 total 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 less or equal than x and all elements in S_2 are greater than x.

Various data structures can be used to solve the dictionary problem, each with distinct trade-offs in performance and complexity.

4.6.1 Arrays

The array can be:

- *Unordered array*: searching and deleting an element take $\mathcal{O}(n)$ time, while inserting an element takes $\mathcal{O}(1)$. This is useful for scenarios like log files, where insertions are frequent, but searches and deletions are rare.
- *Ordered array*: searching takes $\mathcal{O}(\log n)$ time using binary search, while inserting and deleting take $\mathcal{O}(n)$. This is applicable for lookup tables, where searches are frequent, but insertions and deletions are infrequent.

4.6.2 Binary search trees

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 are less or equal than k.
- Keys in the right subtree of v are greater than 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.

4.6.3 Balanced trees

To overcome the drawback of unbalanced BSTs, several balanced tree structures have been developed. These ensure that the tree height remains $\mathcal{O}(\log n)$, providing efficient operations.

AVL trees An AVL tree is a self-balancing binary search tree where the heights of the two child subtrees of any node differ by at most one. Rebalancing operations (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 trees Splay trees are another type of self-adjusting binary search tree that guarantee an amortized time complexity of $\mathcal{O}(\log n)$ per operation. 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.

One of the primary benefits is that they offer amortized logarithmic time for all operations, making them efficient on average over a sequence of operations. Additionally, splay trees are relatively simple to implement compared to other balanced trees because they do not require storing explicit balance information at each node. Another notable advantage is their adaptability to arbitrary access patterns, ensuring that frequently accessed elements are kept near the root, which improves the performance of subsequent accesses.

Each operation can involve multiple rotations, leading to a logarithmic number of rotations per operation, which can make individual operations slower in some cases. Furthermore, splay trees are inefficient when used with higher-dimensional search trees. This inefficiency arises because the secondary data structures associated with each node may need to be recomputed after every rotation, significantly increasing the cost from a constant to a super-linear function of the subtree size. Lastly, while splay trees provide good amortized performance, they do not guarantee that every individual operation will run quickly. Instead, the performance guarantee applies only to the total cost of a sequence of operations.

4.7 Random treaps

Treaps provide efficient time bounds similar to other balanced search tree structures, without requiring explicit balance information to be maintained. The expected number of rotations performed during each operation is small, making them simple to implement. Like skip lists, treaps benefit from randomization, providing good performance for search and update operations.

When n elements are inserted in random order into a binary search tree, the expected depth is approximately $1.39 \log n$ The goal of using treaps is to maintain a search tree structure that resembles the one produced if the elements were inserted in order of their randomly assigned priorities.

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

The tree satisfies the following two properties:

- *Search tree property*: 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 $\text{key}(y) > \text{key}(x)$.
- *Heap property*: 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 $x_1, x_2, \ldots, x_n$ with keys key(x_i) and priorities prio(x_i), there exists a unique treap that satisfies both the search tree property and the heap property.*

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

4.7.1 Search

The search operation in a treap is straightforward, following the same logic as in binary search trees. The search path is determined by comparing the search key with the keys of the nodes along the path from the root.

Algorithm 19 Random treaps search

```
 1: $v = root$
 2: while $v \neq$ 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 running time of this algorithm depends only on the number of elements along the search path. Let $x_1, x_2, \ldots, x_n$ be elements with sorted keys such that $\text{key}(x_1) < \text{key}(x_2) < \text{key}(x_3) < \cdots < \text{key}(x_n)$. Let M be a subset of the elements, and define $P_{\min}(M)$ as the element in M with the lowest priority. The following lemma describes the relationship between elements based on their priorities.

Lemma 4.7.2. *For elements $x_1, x_2, \ldots, x_n$:*

a. If $i < m$, then x_i is an ancestor of x_m if and only if $P_{\min}(\{x_i, \ldots, x_m\}) = x_i$.

b. If $i > m$, then x_i is an ancestor of x_m if and only if $P_{\min}(\{x_m, \ldots, x_i\}) = x_i$.

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 4.7.3 (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 4.7.4 (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

We describe the following algorithms for insertion and deletion in a random treap:

Algorithm 20 Random treaps 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 21 Random treaps 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 $x$
```

The rotations maintain the search tree property and restore the heap property.

Lemma 4.7.5. *The expected running time of the insert and delete operations is $\mathcal{O}(\log n)$, and the expected number of rotations is 2.*

4.7.3 Other operations

For a treap T with n elements, we have the following operations:

-
- *Minimum*(T): return the smallest key with a complexity of $\mathcal{O}(\log n)$.
- *Maximum*(T): return the largest key with a complexity of $\mathcal{O}(\log n)$.
- *List*(T): output elements in increasing order with a complexity of $\mathcal{O}(n)$.
- *Union*(T_1, T_2): merge T_1 and T_2 under the condition that:

$$\text{key}(x_1) < \text{key}(x_2) \qquad \forall x_1 \in T_1, x_2 \in T_2$$

  The complexity of this operation is $\mathcal{O}(\log n)$.

- *Split*(T, k, T_1, T_2): split T into T_1 and T_2 such that:

$$\text{key}(x_1) \le k < \text{key}(x_2) \qquad \forall x_1 \in T_1, x_2 \in T_2$$

  The complexity of this operation is $\mathcal{O}(\log n)$.

Split To split treap T by key k:

1. Insert a new element x with $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$.
2. Insert x into T.
3. Delete the new root. The left subtree becomes T_1, and the right subtree becomes T_2.

Union To merge two treaps T_1 and T_2:

1. Select a key k such that $\text{key}(x_1) < k < \text{key}(x_2)$ for all $x_1 \in T_1$ and $x_2 \in T_2$.
2. Create a new element x with $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$.
3. Create a new treap with x as the root, T_1 as the left subtree, and T_2 as the right subtree.
4. Delete x from the new treap.

Lemma 4.7.6. *The expected running time of the union and split operations is* $\mathcal{O}(\log n)$.

4.7.4 Implementation

In treaps, priorities are drawn from the range $[0, 1)$, and are used to maintain the heap property. Below are some key aspects of the priority handling:

- *Priorities are only used for comparisons*: When two elements are compared, their priorities determine which node becomes the root of a subtree during operations like insertion and deletion. The element with the smaller (higher) priority moves higher in the tree, maintaining the heap property.
- *Handling priority ties*: In the rare case that two elements have equal priorities, a tie-breaking mechanism is employed. Specifically, the priorities are extended by adding random bits, chosen uniformly, to break the tie. This process continues until a difference is found between the corresponding bits, ensuring that ties are resolved without compromising randomness. This guarantees that the heap property is maintained.

4.8 Skip lists

Skip lists are a simple, randomized dynamic search structure invented by William Pugh in 1989. They are easy to implement and maintain a dynamic set of n elements with expected $\mathcal{O}(\log n)$ time complexity for operations like search, insertion, and deletion. Skip lists offer strong probabilistic guarantees, ensuring efficient performance in both expectation and with high probability.

Skip lists improve upon the basic (sorted) linked list, where searches take $\Theta(n)$ time in the worst case. By introducing multiple layers of linked lists, similar to express and local subway lines, searches can be made more efficient. The local line connects all elements, like a standard linked list, while the express line connects a subset of elements, allowing faster traversal over the local line. The express line acts like a shortcut, speeding up searches by reducing the number of elements to check.

4.8.1 Search

The search process in a skip list involves:

1. Walking right in the top linked list until proceeding further would overshoot the target.
2. Dropping down to the next level and continuing the search.
3. Repeat this process until reaching the bottom linked list, where the target element is either found or not.

Higher-levels lists represent popular stations in the subway analogy. Even spacing of these nodes provides the best worst-case performance. For simplicity, evenly distributing nodes across levels results in optimal performance.

Analysis For two levels of linked lists, the search cost is approximately:

$$|L_1| + \frac{|L_2|}{|L_1|}$$

This is minimized when:

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

Thus, the search cost becomes:

$$|L_1| + \frac{|L_2|}{|L_1|} = \sqrt{n} + \frac{n}{\sqrt{n}} = 2\sqrt{n}$$

Extending this idea to k levels of linked lists results in:

$$k\sqrt[k]{n}$$

Finally, for $\log n$ levels, the cost becomes:

$$2\log n$$

This structure of $\log n$ linked lists is analogous to a binary tree, making skip lists an efficient and well-balanced data structure that supports fast search, insertion, and deletion.

4.8.2 Insertion

To insert an element x into a skip list:

1. Perform a search to determine where x belongs in the bottom-most list.
2. Insert x into the bottom list, ensuring that the bottom list contains all elements.
3. Randomly promote x to higher levels using coin flips. Thet is, for each level, with probability $\frac{1}{2}$, promote x to the next higher level. On average, half of the elements are promoted 0 levels, a quarter of the elements are promoted 1 level, and an eighth are promoted 2 levels, and so on.

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 Skip lists in pratice

Skip lists are highly efficient in practice, with searches taking $\mathcal{O}(logn)$ time on average.

Theorem 4.8.1. *With high probability, every search in an n-element skip list takes $\mathcal{O}(\log n)$ time.*

The term with high probability refers to events that occur with probability at least $1 - \mathcal{O}\left(\frac{1}{n^{\alpha}}\right)$, where $\alpha \geq 1$. By setting α large enough, the probability of search time exceeding $\mathcal{O}(logn)$ becomes negligibly small.

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

Backward analysis Analyzing the search process from the leaf to the root provides further insight. Each up move in the search corresponds to the node being promoted during insertion. The total number of up moves is bounded by the number of levels, which is $\mathcal{O}(\log n)$ with high probability.

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

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

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

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

Algorithm 22 Recursive LCS

```
function LCS($x, y, i, j$)
    if $x[i] = y[j]$ then
        $c[i, j] = \mathrm{LCS}(x, y, i-1, j-1) + 1$
    else
        $c[i, j] = \max[\mathrm{LCS}(x, y, i-1, j), \mathrm{LCS}(x, y, i, j-1)\ ]$
    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 23 Memoized recursive LCS

```
procedure LCS($x, y, i, j$)
    if $c[i, j]$ = null then
        if $x[i] = y[j]$ then
            $c[i, j] = \text{LCS}(x, y, i-1, j-1) + 1$
        else
            $c[i, j] = \max[\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)]$
        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 $x[1 \dots i]$ and $y[1 \dots j]$. 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 24 Dynamic Programming LCS

```
procedure LCS($x, y$)
    Initialize $c[0 \dots m, 0 \dots n]$ 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 25 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($f_{\bar{v}}, g_{\bar{v}}, h_{\bar{v}}$)
    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.

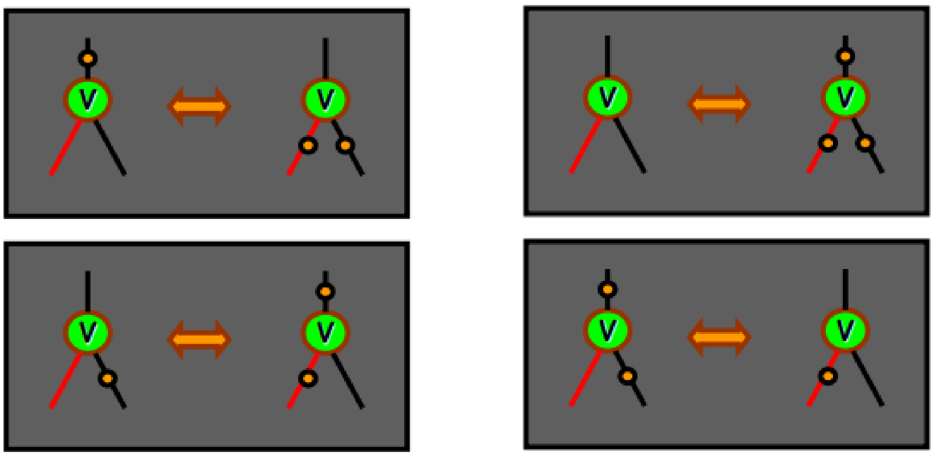


Figure 5.1: Ege equivalences

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.