Advanced Algorithms And Parallel Programming

Theory

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 clearly defined computational procedure that accepts one or more input values and produces one or more output values.

An algorithm can be seen as a tool for solving a clearly defined computational problem. 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* (most common): 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* (occasionally used): 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* (often misleading): this scenario highlights a slow algorithm that performs well on specific inputs.

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

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

In mathematical terms, we define the complexitybound as:

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

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

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

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

Example:
Consider the following expression:

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

The corresponding theta notation is:

$$\Theta(n^3)$$

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

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

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

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

For the expression $\sqrt{n}$:

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

From this, we can redefine the theta notation as:

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

1.2.1 Sorting problem

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

Example:
Given an array:

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

The sorted version will be:

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

Algorithm 1 Insertion sort

```
for j = 2 to n do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    end while
    A[i + 1] = key
end for
```

The complexities for the insertion sort are:

Case	Complexity	Notes
Worst	$T(n) = \Theta\left(\sum_{j=2}^{n} j\right) = \Theta(n^2)$	Input in reverse order
Average	$T(n) = \Theta\left(\sum_{j=2}^{n} \frac{j}{2}\right) = \Theta(n^2)$	All permutations equally likely
Best	$T(n) = \Theta\left(\sum_{j=2}^{n} 1\right) = \Theta(n)$	Already sorted

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

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

Algorithm 2 Merge sort

```
if n = 1 then
    return A[n]
end if
Recursively sort the two half lists A[1 ... ⌈n/2⌉] and A[⌈n/2⌉ + 1 ... n]
Merge (A[1 ... ⌈n/2⌉], A[⌈n/2⌉ + 1 ... n])
```

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

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

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

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

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

$$T(n) = \Theta(n \log_2 n)$$

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.

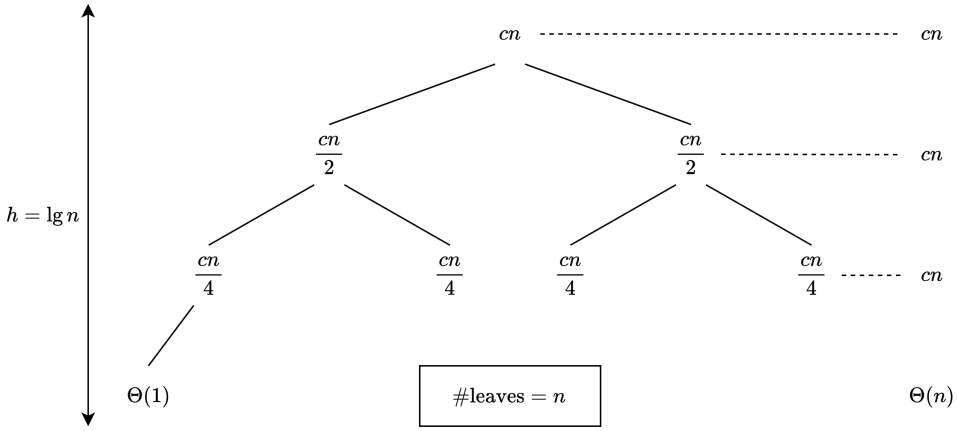


Figure 1.1: Partial recursion tree for merge sort algorithm

The depth of the tree is $h = \log_2 n$, and the total number of leaves is n. Thus, the complexity can be computed as:

$$T(n) = \Theta(n \log_2 n)$$

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 $c < 1$, 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:

$$\Theta(\log_2 n)$$

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 is achieved using the following factors:

$$\begin{cases} P_1 = a \cdot (f - h) \\ P_2 = (a + b) \cdot h \\ P_3 = (c + d) \cdot e \\ P_4 = d \cdot (g - e) \\ P_5 = (a + d) \cdot (e + h) \\ P_6 = (b - d) \cdot (g + h) \\ P_7 = (a - c) \cdot (e + f) \end{cases}$$

Using these products, we can compute the elements of the resulting matrix:

$$\begin{cases} r = P_5 + P_4 - P_2 + P_6 \\ s = P_1 + P_2 \\ t = P_3 + P_4 \\ u = P_5 + P_1 - P_3 - P_7 \end{cases}$$

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.

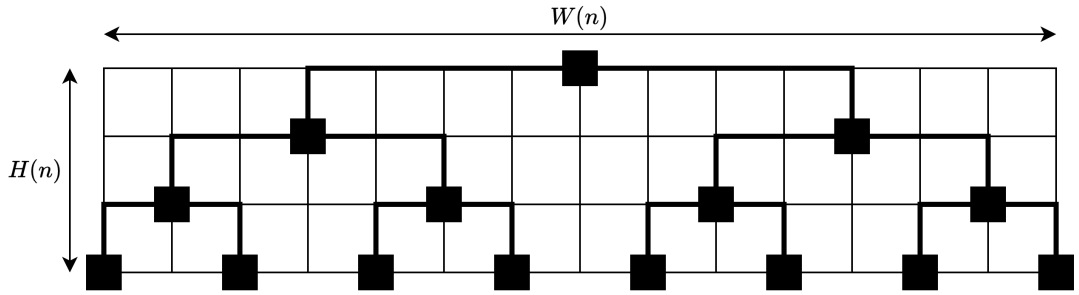


Figure 2.1: VLSI layout problem

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.

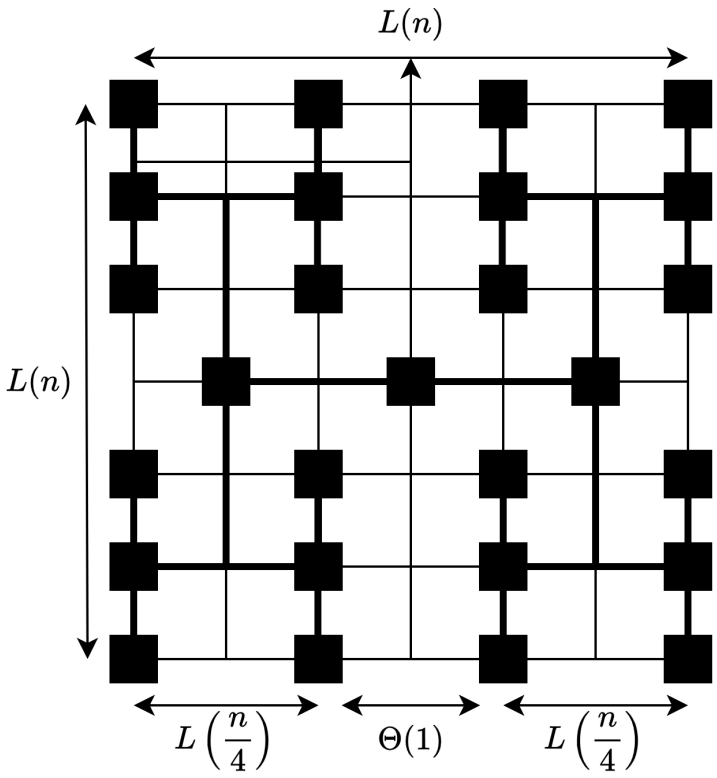


Figure 2.2: VLSI layout problem

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 RAMs.
- 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:

$T_1(n)$	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
$\mathrm{SU}_p = \dfrac{T_1(n)}{T_p(n)}$	Speedup on p processors
$E_p = \dfrac{T_1}{pT_p(n)}$	Efficiency
$C(n) = P(n)T(n)$	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	E_p	C	W
Complexity	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^2}{p}\right)$	1	$\mathcal{O}(n^2)$	p

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	SU_p	E_p	C	W
Complexity $p = n$	$\mathcal{O}(n)$	$\mathcal{O}\left(2 + \log n\right)$	$\mathcal{O}\left(\frac{n}{2 + \log n}\right)$	$\frac{1}{\log n}$	$\mathcal{O}(n \log n)$	-
Complexity $p = n$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{p} + \log p\right)$	$\mathcal{O}(p)$	1	$\mathcal{O}(n)$	$\mathcal{O}(n)$

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	SU_p	E_p	C
Complexity	$\mathcal{O}(n^3)$	$\mathcal{O}\left(\log n\right)$	$\mathcal{O}\left(\frac{n^3}{\log n}\right)$	$\frac{1}{\log n}$	$\mathcal{O}\left(n^3 \log n\right)$

3.4 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 3.4.1. *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 $\mathcal{O}\left(\frac{TM}{M'}\right)$ steps.*

Proof. We can partition the M simulated shared memory cells into M' continuous segments S_i of size $\frac{M}{M'}$ each. Each simulating processor P'_i $(1 \leq i \leq P)$ will simulate processor P_i of the original PRAM. Each simulating processor P'_i $(1 \leq i \leq M')$ stores the initial contents of segment S_i into its local memory and will use $M'[i]$ as an auxiliary memory cell for simulating accesses to cells of S_i.

Each P'_i $(i = 1, \ldots, \max(P, M'))$ repeats the following for $k = 1, \ldots, \frac{M}{M'}$. Write the value of the k-th cell of segment S_i into $M'[i]$ for $i = 1, \ldots, M'$. Read the value that the simulated processor P_i $(i = 1, \ldots, P)$ would read in this simulated substep, if it appeared in the shared memory. The local computation substep of P_i $(i = 1, \ldots, P)$ is simulated in one step by P'_i. The simulation of one original write operation is analogous to that of the read operation. □

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. In some cases, PRAM can be implemented directly by translating its concepts to hardware. PRAM's CRCW model can be implemented using detect-and-merge techniques, where write conflicts are resolved by merging results. Priority CRCW, on the other hand, resolves conflicts by detecting and prioritizing certain writes over others.

PRAM is an attractive model for parallel computing due to several key factors. One major advantage is the large body of algorithms developed specifically for it, offering a rich resource for problem-solving. Its simplicity makes it easy to conceptualize, as PRAM abstracts away the complexities of synchronization and communication, allowing a pure focus on algorithm design. The synchronized shared memory model in PRAM eliminates many of the challenges related to synchronization and communication that arise in practical implementations. However, PRAM retains the flexibility to incorporate these issues when necessary, enabling exploration of more complex scenarios. A notable strength of PRAM is its adaptability. Algorithms initially designed for this model can often be converted into asynchronous versions, which are better suited to real-world architectures.

3.5 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.

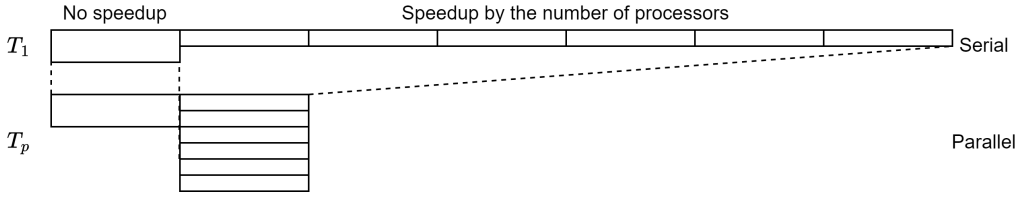


Figure 3.1: Serial and parallel models

When using more than one processor, the speedup (SU_P) is always less than the number of processors (P). The relationship can be expressed as:

$$T_P > \frac{T_1}{P} \implies \mathrm{SU}_P < P$$

Here, T_P is the time taken with P processors, and T_1 is the time for a serial execution.

In a program, the parallelizable portion is often represented by a fixed fraction f.

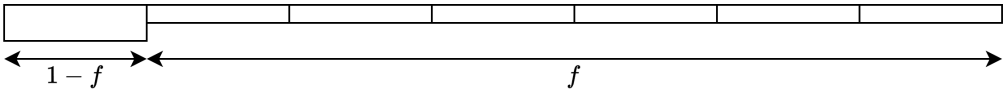


Figure 3.2: Serial model

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

$$\mathrm{SU}_P(f) = \frac{T_1}{T_P} = \frac{T_1}{T_1(1-f) + \frac{T_1 f}{P}} = \frac{1}{1 - f + \frac{f}{P}}$$

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.

3.5.1 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:

$$\mathrm{SU}(P) = \frac{T_1}{T_P} = \frac{s + P(1-s)}{s + 1 - s} = s + P(1-s)$$

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.

3.5.2 Conclusion

Amdahl's Law assumes that the overall computing workload remains constant, and thus adding more processors merely reduces execution time for the same task. This makes Amdahl's model practical for scenarios where the problem size is fixed, but speedup is limited by the serial portion of the task.

On the other hand, Gustafson's Law takes a more expansive view, arguing that as computing power increases, the problem size grows accordingly. More processors allow for deeper, more detailed analysis, which wouldn't have been feasible with limited computational resources. Increasing the power of computation enables more comprehensive simulations, something impossible within fixed-size, limited-time models.

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.

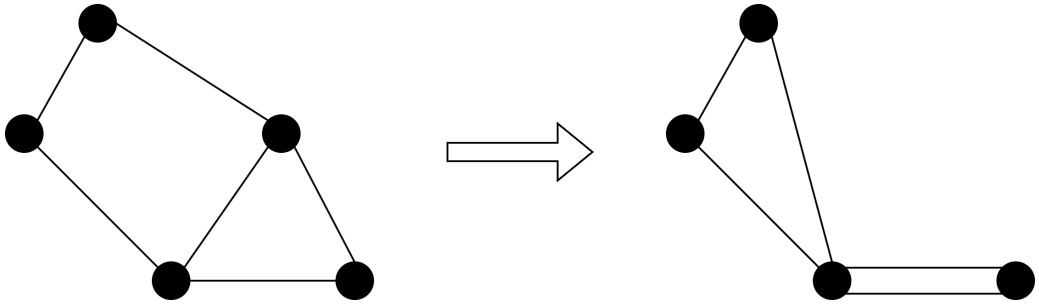


Figure 4.1: Edge contraction

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 δ(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:*

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

Proof. Let the contracted edges be $\{e_1, e_2, \ldots, e_{n-2}\}$. The algorithm succeeds if none of the contracted edges are part of the minimum cut $\delta(S)$.

Initially, the input graph G has at least $\frac{nk}{2}$ edges, since the minimum degree in G is at least k. Otherwise, we could disconnect a vertex with fewer than k edges, forming a smaller cut, which contradicts the assumption that $\delta(S)$ is a minimum cut.

After j contractions, the multigraph G_j contains $n - j$ vertices, and its minimum degree is still at least k. Thus, G_j has at least $\frac{(n-j)k}{2}$ edges.

The probability that the algorithm successfully finds the minimum cut $\delta(S)$ is the probability that none of the contracted edges belong to $\delta(S)$. This probability is bounded below by:

$$\begin{aligned}
\Pr(\text{final graph} = \delta(S)) &= \Pr(e_1, e_2, \ldots, e_{n-2} \notin \delta(S)) \\
&= \Pr(e_1 \notin \delta(S)) \prod_{j=1}^{n-3} \Pr(e_{j+1} \notin \delta(S) \mid e_1, \ldots, e_j \notin \delta(S)) \\
&\geq \prod_{j=0}^{n-3} \left(1 - \frac{k}{(n-j)k/2}\right) \\
&= \frac{n-2}{n} \times \frac{n-3}{n-1} \times \cdots \times \frac{2}{4} \times \frac{1}{3} \\
&= \frac{2}{n(n-1)} \\
&= \frac{1}{\binom{n}{2}}
\end{aligned}$$

□

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

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

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

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

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

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] ≤ x then
6:             i = i + 1
7:             exchange A[i] ↔ A[j]
8:         end if
9:     end for
10:    exchange A[p] ↔ 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 $\mathcal{O}(n\log n)$.

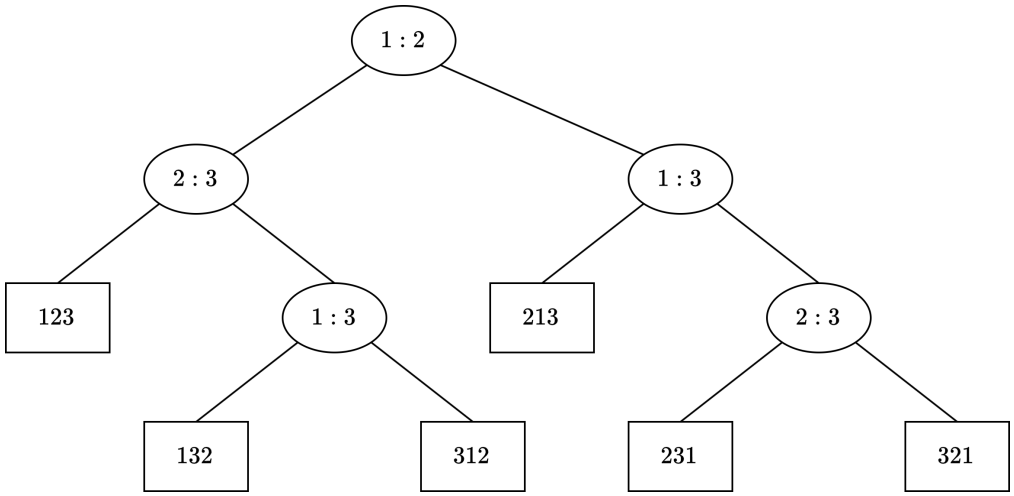
To understand why $\mathcal{O}(n\log n)$ is the best achievable worst-case performance for comparison sorts, we can model sorting algorithms using decision trees.

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 $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

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 $2^h \geq n!$. Taking the logarithm on both sides:

$$h \geq \log n!$$

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

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

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

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

1. *Initial state*: 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.
- $i = \left\lfloor \frac{n+1}{2} \right\rfloor$ or $\left\lceil \frac{n+1}{2} \right\rceil$: 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

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

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

This algorithm performs exactly $n - 1$ comparisons, making it optimal for finding 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

```
1: if length(A) is odd then
2:     max = min = A[1]
3:     i = 2
4: else if A[1] < A[2] then
5:     min = A[1]
6:     max = A[2]
7:     i = 3
8: else
9:     min = A[2]
10:    max = A[1]
11:    i = 3
12: end if
13: while i ≤ length(A) do
14:    if A[i] < A[i + 1] then
15:        if min > A[i] then
16:            min = A[i]
17:        end if
18:        if max < A[i + 1] then
19:            max = A[i + 1]
20:        end if
21:    else
22:        if min > A[i + 1] then
23:            min = A[i + 1]
24:        end if
25:        if max < A[i] then
26:            max = A[i]
27:        end if
28:    end if
29:    i = i + 2
30: 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

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

The running time of 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:

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

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

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 $\lfloor \frac{n}{10} \rfloor$ elements are less than or equal to x. Similarly, at least $\lfloor \frac{n}{10} \rfloor$ elements are greater than or equal to x. Thus, at least $\lfloor \frac{3n}{10} \rfloor$ elements are discarded after each partition.

Therefore, in the worst case, the recursive call to SELECT operates on at most $\lfloor \frac{3n}{4} \rfloor$ 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.