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.

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; for instance, sorting an already sorted sequence is less complex. Therefore, we often parameterize running time by the input size, as shorter sequences are typically easier to sort than longer ones. Generally, we aim for upper bounds on running time, as guarantees are desirable.

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 involves disregarding machine-dependent constants or analyzing the growth of $T(n)$ as n approaches infinity. 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. Thus, asymptotic analysis serves as a valuable tool for crafting solutions to specific problems.

1.2.1 Theta notation

In mathematical terms, the theta notation is defined 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 $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

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:

$$\Theta(n^3)$$

From theta notation, we can also define:

- *Upper bound*:

  $$O(g(n)) = f(n)$$

  Here, $f(n)$ meets the condition that there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- *Lower bound*:

  $$\Omega(g(n)) = f(n)$$

  Here, $f(n)$ meets the condition that there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$.

- *Strict upper bound*: the strict upper bound is defined as:

  $$o(g(n)) = f(n)$$

  Here, $f(n)$ meets the condition that there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$.

- *Strict lower bound*: the strict lower bound is defined as:

  $$\omega(g(n)) = f(n)$$

  Here, $f(n)$ meets the condition that there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$.

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

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

For the expressio $\sqrt{n}$:

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

From this, we can redefine the strict bound as:

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

1.2.2 Sorting problem

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

Insertion sort One approach to solving this problem is the insertion sort algorithm, which takes an array $A[n]$ as input.

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
```

For insertion sort, the worst-case scenario occurs when the input is sorted in reverse order, leading to a complexity of:

$$T_{\text{worst}}(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

For the average case, assuming all permutations are equally likely, the complexity is:

$$T(n)_{\text{average}} = \sum_{j=2}^{n} \Theta\left(\frac{j}{2}\right) = \Theta(n^2)$$

In the best-case scenario, where the list is already sorted, we only need to check each element:

$$T(n)_{\text{best}} = \sum_{j=2}^{n} \Theta(1) = \Theta(n)$$

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

1.3 Recursion complexity

Merge sort A recursive solution to the sorting problem is the merge sort algorithm, which takes an array $A[n]$ as input.

Algorithm 2 Merge sort

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

The key subroutine that makes this algorithm recursive is the merge operation.
To analyze the complexity, we consider the following components:

- When the array has only one element (base case), the complexity is constant: $\Theta(1)$.
- The recursive sorting of the two halves (line 4) contributes a total cost of $2T\left(\frac{n}{2}\right)$ since both halves are sorted independently.
- Finally, the merging of the two sorted lists requires linear time to check all elements, yielding a complexity of $\Theta(n)$.

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.

1.3.1 Recursion tree

To determine the complexity of the merge sort algorithm, we need to solve the equation:

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

We can use a recursion tree for this purpose, expanding nodes until we reach the base case. A partial recursion tree for the algorithm is illustrated below:

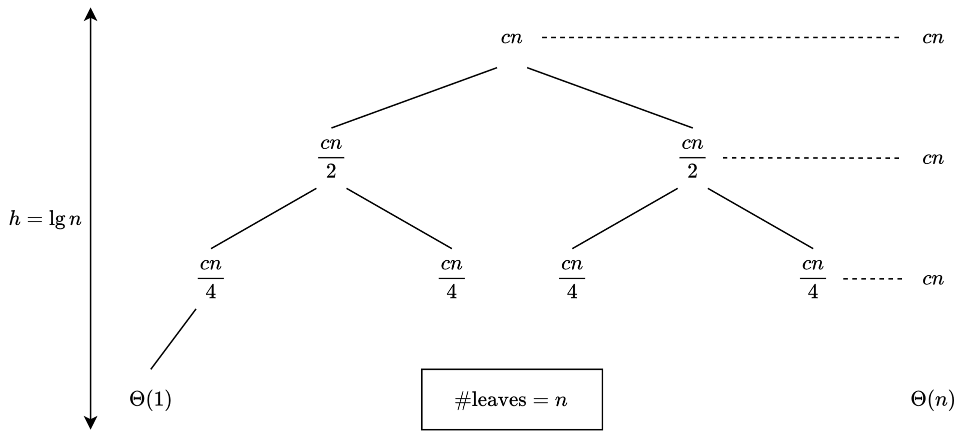

Figure 1.1: Partial recursion tree for merge sort algorithm

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

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

This result shows that $\Theta(\log_2(n) \cdot n)$ grows slower than $\Theta(n^2)$ indicating that merge sort outperforms insertion sort in the worst case. 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 $O(n^3)$, so we assume $T(k) \leq c \cdot k^3$ for $k < n$.
2. Verify by induction that $T(n) \leq c \cdot n^3$.

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 $f(n) = O(n^{2-\varepsilon})$ 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.

n 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:

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

Here, we have $a = 1$ and $b = 2$, leading to:

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

We can apply the second case of the master method with $k = 0$, resulting in 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:

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

By applying the master method, we find a final complexity of $\Theta(\log_2 n)$, which is significantly more efficient than the naive approach.

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:             $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$
6:         end for
7:     end for
8: end for
```

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

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 $\Theta(n^3)$, the same as the standard algorithm.

2.4.1 Strassen algorithm

To improve efficiency, Strassen proposed a method that reduces the number of multiplications from eight to seven for 2×2 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 $\Theta\left(n^{\log_2 7}\right)$, which is approximately $\Theta\left(n^{2.81}\right)$.

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.

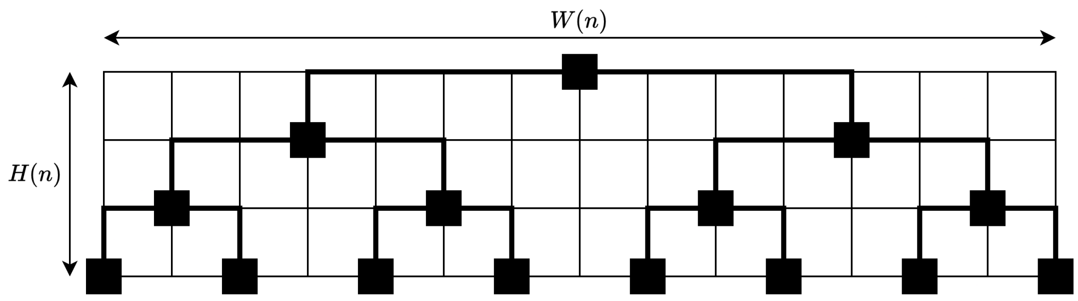

Figure 2.1: VLSI layout problem

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:

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

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

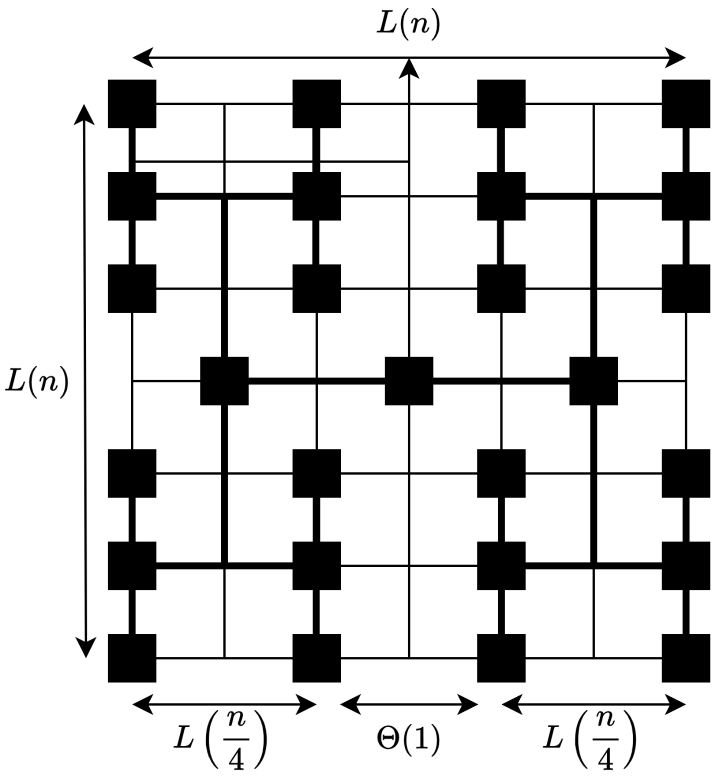

Figure 2.2: VLSI layout problem

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

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

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

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

2.6 Summary

Divide and conquer is one of several effective techniques for algorithm design. Algorithms that employ this strategy can be analyzed using recurrences and the master method. This approach frequently results in efficient algorithms.