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 any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm must terminate in a finite number of steps.

1.2 Running time analysis

The running time of an algorithm depends on the input: an already sorted sequence is easier to sort. So, we parametrize the running time by the size of the input, since short sequences are easier to sort than long ones. Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

The running time analysis can be based on:

- *Worst-case* (usually): in this case $T(n)$ is the maximum time of algorithm on any input of size n. This case is considered in situation where the time is a critical factor.
- *Average-case* (sometimes): in this case $T(n)$ is the expected time of algorithm over all inputs of size n. The average case needs assumption of statistical distribution of inputs.
- *Best-case* (bogus): cheat with a slow algorithm that works fast on some input.

To have a general measure of the complexity, we need a machine-independent evaluation. In order to do that we may ignore all machine-dependent constants or to look at the growth of $T(n)$ as n tends to infinity. This framework is called asymptotic analysis.

When the length of the input n gets large enough, an algorithm with a lower complexity will always beat the ones with higher degrees. However, we should not ignore asymptotically slower algorithms since in real world design situations often we often need a careful balancing of engineering objectives. Then, asymptotic analysis is a useful tool to help in designing a solution for a given problem

1.2.1 Theta notation

In mathematics we have that the theta notation:

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

Here, $f(n)$ is such that there exists 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 we simply ignore the lower-order terms and ignore the constants.

Example:
Consider the following expression:

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

The theta notation corresponding to it is as follows:

$$\Theta(n^3)$$

From the theta notation we can also define:

- *Upper bound*: the upper bound is defined as:

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

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

- *Lower bound*: the lower bound is defined as:

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

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

Example:
Consider the expression $2n^2$, we have that its upper bound is:

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

Consider the expression $\sqrt{n}$, we have that its upper bound is:

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

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 problem of sorting consists in, given an array of numbers $\langle a_1, a_2, \ldots, a_n \rangle$, we need to return the permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Insertion sort One possible solution to this problem is the insertion sort algorithm, that takes as input an array $A[n]$.

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 the insertion sort we have a worst case scenario in which the input is sorted but in reverse, and the complexity is:

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

For the average case we consider that all permutations are equally likely, yielding a complexity of:

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

For the best case scenario we have an ordered list, and so we simply need to check all the elements in the list:

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

As a final result, we have that this algorithm is fast for small n, but becomes infeasible for larger input dimensions.

1.3 Recursive running time analysis

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

Algorithm 2 Merge sort

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

The key subroutine of the given algorithm is merge, that makes the algorithm recursive. For the complexity analysis we have to consider the following elements:

- If the array contains only one element (first line), the complexity is constant, resulting in $\Theta(1)$.
- The recursive sort of the two lists (line four) cost a total of $(\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor)$, but since floor and ceiling does not matter asymptotically we have a total complexity of $2T\left(\frac{n}{2}\right)$. This is because the complexity of this sorting depends on the complexity of the algorithm itself.
- Finally, the merging of the lists has a linear cost, since we need to check all the elements in the lists, yielding a complexity of $\Theta(n)$.

The final result for the merge sort algorithm is:

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

In case in which the base case has a complexity of $\Theta(1)$ for sufficiently small n we can omit it. However, this can be done only when it has no effect on the asymptotic solution to the recurrence.

1.3.1 Recursion tree

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

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

To do so we can use the recursion tree, starting from the root. We continue to expand the leaf until we cannot go further down because we reached the base case. For this algorithm a partial recursion tree is the following:

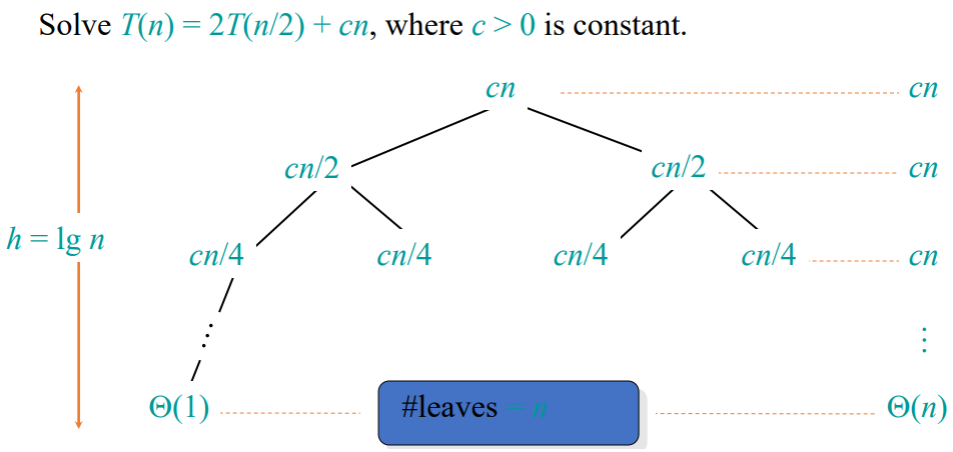

Figure 1.1: Partial recursion tree for merge sort algorithm

At this point we have a depth of $h = \log_2(n)$, and a total of n leaves. As a result, the algorithm's complexity is the product of number of leaves with the depth of the tree, that is:

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

The final result shows that $\Theta(\log_2(n) \cdot n)$ grows more slowly than $\Theta(n^2)$. Therefore, merge sort asymptotically beats insertion sort in the worst case. In practice, merge sort beats insertion sort for $n > 30$ or so.

1.3.2 Substitution method

The substitution method is the most general method to solve recursive complexity equations. The steps to apply the substitution method are:

1. Guess the form of the solution by performing a preliminary analysis on the algorithm.
2. Verify by induction.
3. Solve for constants.

Example:
Consider the following expression:

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

We assume that the base case is $T(1) = \Theta(1)$. We can now apply the substitution method with by performing the following steps:

1. We guess that the solution has a complexity of $O(n^3)$, ad so we assume that $T(k) \leq c \cdot k^3$ for $k < n$.
2. We verify by induction that $T(n) \leq c \cdot n^3$.

The problem with this approach is that it is not simple to apply in every situation.

1.3.3 Master method

To overcome the complexity of the substitution method we may use the master method. The master method applies 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. So, it is less general than substitution, but it is more straightforward.

In general, this method is always applicable for divide and conquer style algorithms.

To apply this method we have to compare $f(n)$ with $n^{\log_b a}$. We could have three outcomes:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$. In this case we have that the function $f(n)$ grows polynomially slower than $n^{\log_b a}$ by an n^{ε} factor. In this scenario the solution is:

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

2. $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$. In this case we have that the function $f(n)$ and $n^{\log_b a}$ grow at similar rates. In this scenario the solution is:

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

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$. In this case we have that the function $f(n)$ grows polynomially faster than $n^{\log_b a}$ by an n^{ε} factor, and $f(n)$ satisfies the regularity condition that $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some constant $c < 1$. In this scenario the solution is:

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

Example:
Let's consider the expression:

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

In this expression we have that $a = 4$ and $b = 2$, so we have that:

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

So we have are in the first case of the theorem, that is $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$. So, the solution is:

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

Let's consider the expression:

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

In this expression we have that $a = 4$ and $b = 2$, so we have that:

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

So we have are in the second case of the theorem, that is $f(n) = \Theta(n^2 \log^k n)$ for $k = 0$. So, the solution is:

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

Let's consider the expression:

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

In this expression we have that $a = 4$ and $b = 2$, so we have that:

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

So we have are in the second case of the theorem, that is $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$. So, the solution is:

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

Let's consider the expression:

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

In this expression we have that $a = 4$ and $b = 2$, so we have that:

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

In this case the method does not apply in this case. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\log n)$.

CHAPTER 2

Divide and conquer algorithms

2.1 Introduction

The divide and conquer design paradigm consists in the following parts:

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

This procedure allow to have a more problems with less input length that can be solved in less time.

The divide step is constant since we have to simply split an array in two parts of equal length. The second step depends on the algorithm that is going to be analyzed. The combine step depends again on the algorithm: it can either constant or take some time.

Merge sort The merge sort considered before is performed with the following steps:

- *Divide*: the array is trivially divided in two sub-arrays.
- *Conquer*: the two sub-arrays are recursively sorted.
- *Combine*: the two sub-arrays are merged in a linear time.

The recursive expression for the complexity of this algorithm is then divided into the following components:

2.2 Binary search

The problem of binary search consists in finding an element in a sorted array. This problem can be solved in a divide and conquer manner in the following way:

1. *Divide*: check the middle element of the array .
2. *Conquer*: recursively search one sub-array.

3. *Combine*: if the result is found, return the index of the element in the array.

In this case we have one sub-problem, that is the new sub-array. The sub-array has a length that is half the length of the original problem. And the complexity of the divide and combine steps is constant. Therefore, the final expression for the complexity is:

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

In this case we have that $a = 1$, and $b = 2$, and so we have that:

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

We can then use the second case of the master method with $k = 0$, obtaining a final complexity of $T(n) = \Theta(\log n)$.

2.3 Powering a number

The problem is the computation of the value of a^n, where $n \in \mathbb{N}$. The naive approach consists in multiplying n times the value of a, with a total complexity of $\Theta(n)$.

We can apply a divide an conquer algorithm also to this problem, by dividing the power by two in the following way:

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

In this scenario, we have that the divide and the combination phase have a constant complexity since can be performed with a single division or a single multiplication, respectively. We have half input dimension at each iteration step, and we have exactly one sub-problem to solve (since we have two equal sub-problems). Thus, the final recurrent formula for complexity is:

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

By applying the master method, we obtain a final complexity of $\Theta(\log_2 n)$, that is lower than the naive situation.