CSCI3160 Design and Analysis of Algorithms

Ryan Chan

November 1, 2025

Abstract

This is a note for **CSCI3160 Design and Analysis of Algorithms**.
Contents are adapted from the lecture notes of CSCI3160, prepared by Xiao Liang and Yufei Tao, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

1 Introduction **2**
1.1 The RAM Model 2

2 Divide and Conquer **5**

3 Greedy Algorithm **6**

4 Dynamic Programming **7**

5 Graph **8**

Chapter 1

Introduction

In mathematics, everything must be rigorous, including terms and symbols. As we say that computer science is a branch of mathematics, we can think of the same for computer science itself.

More specifically, computer science is a subject where we first define a computation model, which is a simple yet accurate abstraction of a computing machine, and then we slowly build up a theory based on this model.

However, since there isn't a universally ideal model — different models serve different purposes — how do we choose one suitable for studying algorithms? We have three criteria:

1. Mechanically Implementable

This ensures that we can implement the algorithm to solve real-world problems.

2. Sufficiently General

The model should be general enough to capture all natural steps involved in solving problems strategically.

3. Sensitive Enough

The model should be able to capture fine-grained differences in resource usage, such as time and space.

Then, based on these criteria, we can choose a computation model suitable for studying algorithms: the Random Access Machine (RAM) model.

1.1 The RAM Model

1.1.1 Overview

The RAM model is a machine that has a **memory** and a **CPU**.

Memory is defined as an infinite sequence of cells, each containing the same number of w bits. Every cell has an address: the first cell of memory has address 1, the second cell 2, and so on.

The CPU contains a fixed number of registers (we consider 8 throughout this course), each of which has w bits.

1.1.2 Atomic Operations

We say that there are a few atomic operations that the CPU can perform.

1. **Register (Re-)Initialization**

We can set a register to a fixed value (e.g., 0, -1, 100, etc.) or to the content of another register.

2. **Arithmetic**

This operation takes the integers a, b stored in two registers. It calculates one of the following and stores

the result in a register:

$$a+b,\ a-b,\ a\times b,\ a/b$$

> **Remark.** Here, a/b denotes integer division, where only the integer part of the quotient is returned.

3. **Comparison / Branching**

This operation takes the integers a, b stored in two registers, compares them, and determines which of the following is true:

$$a>b,\ a=b,\ a<b$$

4. **Memory Access**

This operation takes a memory address A currently stored in a register, then performs one of the following:

- Read the content of the memory cell with address A into a designated register (overwriting the bits there).

- Write the content of a designated register into the memory cell with address A (overwriting the bits there).

5. **Randomness**

The command `RANDOM(x, y)` returns an integer chosen uniformly at random in $[x, y]$, where $x \leq y$. The resulting random integer is then placed in a register.

We look at the atomic operations because an execution is a sequence of atomic operations. The cost of an execution (also called running time) is simply the length of the sequence, i.e., the number of atomic operations it requires.

A word is a sequence of w bits, where w is called the word length. In other words, each memory cell and CPU register stores a word.

1.1.3 Algorithms

We first take a look at some terminologies.

An input refers to the initial state of the registers and the memory before an execution starts.

An algorithm is a description that, given an input, can be utilized to **unambiguously** produce a sequence of atomic operations — namely, the execution of the algorithm. In other words, it should always be clear what the next atomic operation should be, given the outcomes of all the previous atomic operations.

The cost of an algorithm on an input is the length of its execution on that input (i.e., the number of atomic operations required).

The space of an algorithm on an input is the largest memory address accessed by the algorithm's execution on that input.

We define an algorithm as **deterministic** if it never invokes the atomic operation `RANDOM`; otherwise, the algorithm is **randomized**.

1.1.4 Expected Running Time

For the same input, the cost of a deterministic algorithm is a fixed integer, since it remains the same every time the algorithm is executed. However, the cost of a randomized algorithm is a random variable, since for each input the cost may change every time the algorithm is executed.

Algorithm 1.1: Example

```
1 while r ≠ 1 do
2     r = RANDOM(0, 1)
3 return r = 1
```

Here, we are not able to know how many times line 2 will be executed since each execution can produce a new sequence of atomic operations. It can even be executed an infinite number of times.

Thus, we can use the **expected cost** to evaluate the cost of a randomized algorithm. Let X be a random variable that represents the cost of an algorithm on an input. Then, the expected cost of the algorithm on that input is the expectation of X.

We consider the expected running time a good metric for analyzing randomized algorithms for the following reasons.

First, the expected running time is usually easier to compute and analyze than other probabilistic metrics. It provides a single, concrete number that summarizes the algorithm's typical performance.

Second, we can make use of the **Linearity of Expectation**, which allows us to break a complex algorithm into smaller parts and sum their expected costs.

Lastly, if the running time of a randomized algorithm has very high variance, the expectation may be misleading; however, we can address this by utilizing concentration bounds.

Here we look at one of the concentration bounds — the Markov inequality — as an example.

Theorem 1.1.1 (Markov's Inequality)**.** Suppose that a random variable $X \geq 0$ only takes non-negative values. Then, for every $t > 0$,

$$\mathbb{P}(X \geq t) \leq \frac{\mathbb{E}[X]}{t}.$$

The theorem here shows that if the average value of X is small, then it is less likely that X is large. This provides an upper bound on the probability that a non-negative random variable is much larger than its expectation.

We can then use this theorem to analyze running time. Let T be the random variable representing the running time of a randomized algorithm, and suppose $T \geq 0$ and $\mathbb{E}[T] = \mu$. Then, for any $c > 1$,

$$\mathbb{P}(T \geq c\mu) \leq \frac{1}{c}.$$

Example (Las Vegas Algorithm)**.** A Las Vegas algorithm always gives the correct result but has a random running time.

Suppose $\mathbb{E}[T] = \mu$, and we run this algorithm with a timeout of $c\mu$. Then we have

$$\mathbb{P}[\text{timeout}] \leq \frac{1}{c}.$$

This shows that if we run the algorithm for c times its expected running time, then the probability that it fails will be less than or equal to $\frac{1}{c}$.

For example, if we run for 3μ steps, then it will fail with probability $\leq \frac{1}{3}$. Therefore, if we repeat it 3 times, we will succeed with probability $1 - \left(\frac{1}{3}\right)^3 = 0.963$.

Chapter 2

Divide and Conquer

Chapter 3

Greedy Algorithm

Chapter 4

Dynamic Programming

Chapter 5

Graph