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
1.2 Efficiency of the Worst Input 4

2 Divide and Conquer **6**

3 Greedy Algorithm **7**

4 Dynamic Programming **8**

5 Graph **9**

Chapter 1

Introduction

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 gradually build up a theory based on this model.

Thus, the first thing to do is to come up with a computation model. For the study of algorithms, we need to consider the following criteria:

1. **Mechanically Implementable:** it can run real algorithms.

2. **Sufficiently General:** it can capture all the natural steps of problem-solving.

3. **Sensitive Enough:** it can distinguish differences in resource usage (time and space).

We choose the **Random Access Machine (RAM)** as our model.

1.1 The RAM Model

The RAM model has a **memory** and a **CPU**.

Memory is defined as an infinite sequence of cells, each containing the same number of w bits, with addresses 1, 2, and so on. The CPU contains a fixed number of registers, each of which has w bits. 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.

1.1.1 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 or to the content of another register.

2. **Arithmetic**

This operation takes two integers stored in two registers and performs basic arithmetic calculations.

3. **Comparison / Branching**

This operation takes two integers stored in two registers, compares them, and determines the result of the comparison.

4. **Memory Access**

This operation takes a memory address that is currently stored in a register, then performs either reading (to register) or writing (to memory).

5. **Randomness**

`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.

An execution is defined as a sequence of atomic operations, where the cost (running time) of an execution is the length of such a sequence, i.e., the number of atomic operations it performs.

1.1.2 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**.

1.1.3 Expected Running Time

A deterministic algorithm has a fixed cost for the same input, whereas for a randomized algorithm, the cost is a random variable, since for each input the cost might change every time the algorithm is executed.

Algorithm 1.1: Find a One

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

Here, we cannot know how many times line 2 will be executed, as each execution can produce a new sequence of atomic operations.

Thus, we use the **expected cost** to evaluate the cost of a randomized algorithm.

Definition 1.1.1. Let X be a random variable that represents the cost of an algorithm on a given input. The expected cost of the algorithm on that input is the expectation of X.

We use the expected running time, rather than other metrics, for the following reasons:

1. Ease of computation
2. Linearity of expectation: it allows us to break the analysis into smaller parts
3. Concentration bounds

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.

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 $1 - \left(\frac{1}{3}\right)^3 = 0.963$.

1.2 Efficiency of the Worst Input

By looking at the cost of an algorithm on the worst input, we are able to measure the speed of an algorithm.

Definition 1.2.1. Define $\mathcal{I}_n$, where n is an integer, to be the set of all inputs to a problem that have the same problem size n. Given an input $I \in \mathcal{I}_n$, the cost $X_{\mathcal{A}}(I)$ of an algorithm $\mathcal{A}$ is the length of its execution on I.

The worst-case cost of $\mathcal{A}$ under the problem size n is the maximum $X_{\mathcal{A}}(I)$ over all $I \in \mathcal{I}_n$.

The worst expected cost of $\mathcal{A}$ under the problem size n is the maximum $\mathbb{E}[X_{\mathcal{A}}(I)]$ over all $I \in \mathcal{I}_n$.

Example (Dictionary Search). In memory, a set S of n integers has been arranged in ascending order in the memory cells from address 1 to n. The value of n has been placed in Register 1 of the CPU. Another integer v has been placed in Register 2, where n is the problem size, and $\mathcal{I}_n$ is the set of all possible (S, v).

We want to determine whether v exists in S.

The worst-case cost of the binary search algorithm is $\mathcal{O}(\log n)$. This means that on any input in $\mathcal{I}_n$, the maximum number $f(n)$ of atomic operations performed by the algorithm grows no faster than $\log_2 n$.

Example (Randomized Algorithm). Consider the following randomized algorithm:

Algorithm 1.2: Find a Zero

Data: A is an array of size n that contains at least one 0

```
1 while A[r] ≠ 0 do
2     r = RANDOM(0, n)
3 return r
```

The expected cost depends on the input array. For example, if all numbers in A are 0, then the algorithm finishes in $\mathcal{O}(1)$. If it has only one 0, the algorithm will finish in $\mathcal{O}(n)$ because each $A[r]$ has a $\frac{1}{n}$ probability of being 0, and we need to repeat n times in expectation to find the 0.

Thus, we have worst-case cost $= \infty$ and worst expected cost $= \mathcal{O}(n)$.

Example (Find a Zero). Let A be an array of n integers, among which half are 0. Design an algorithm to report an arbitrary position of A that contains a 0.

Algorithm 1.3: Find a Zero

```
1 while A[r] ≠ 0 do
2     r = RANDOM(0, n)
3 return r
```

The algorithm finishes in $\mathcal{O}(1)$ expected time on every input A.

Proof. Let X be the number of iterations until the algorithm picks a zero. Each iteration chooses $r \in \{0, \ldots, n\}$.

Then we have

$$p = \mathbb{P}[A[r] = 0] = \frac{\#\text{ of zeros}}{n} = \frac{\frac{n}{2}}{n} = \frac{1}{2}.$$

Then X is a geometric random variable with success probability $p = \frac{1}{2}$.

For a geometric random variable, we have

$$\mathbb{E}[X] = \frac{1}{p} = \frac{1}{\frac{1}{2}} = 2.$$

Since each iteration takes $\mathcal{O}(1)$ time to pick a random index, we need two such iterations in expectation. Thus, the expected running time is $2 \times \mathcal{O}(1) = \mathcal{O}(1)$. ■

Remark. In contrast, any deterministic algorithm must probe at least $\frac{n}{2}$ integers of A in the worst case. In other words, any deterministic algorithm must have a worst-case time of $\Theta(n)$ — provably slower than the above randomized algorithm (in expectation).

Chapter 2

Divide and Conquer

Chapter 3

Greedy Algorithm

Chapter 4

Dynamic Programming

Chapter 5

Graph