Colorado CSCI 5454: Algorithms **Fall 2020**

Algorithms: Introduction

Instructor: Bo Waggoner *Lecture 1*

These notes review some basics we will rely on throughout the course.

Objectives for this lecture:

- Understand how we mathematically define algorithms in this course: the word RAM model.
- Understand how we rigorously analyze algorithms in this course: proofs of correctness and efficiency (what is a proof?), big-O notation.
- Familiarize yourself with the course outline as a breakdown of the kinds of algorithmic problems we will encounter.

1 What is an algorithm?

The word traces its roots to *al-Khwarizmi*, the Persian author of a ninth-century text on algebra. For a long time the term *algorithm* referred to methods of numerical calculation such as solving equations. Nowadays, it is much broader.

A working definition for the purposes of this class:

An **algorithm** is a series of steps* for solving a problem*.

where, importantly, the allowable steps and the problem are

*mathematically well-defined.

The rest of these notes discuss what it means for a problem and an algorithm's steps to be mathematically well-defined; then, what *design* and *analysis* of algorithms entails.

1.1 Well-defined problems

For a problem to be mathematically well-defined, we usually require it to have input and output represented in digital symbols, e.g. binary or ASCII characters. For example, the input may be a list of integers (separated by commas), while the output is required to be a sorted list of the same integers.

In first courses on algorithms, one generally studies problems like sorting where the algorithm begins with a certain input, must convert it to a certain output, and then stops.

Later in this course, we will expand our notions of *problems* in various ways. Sometimes the goal will be to find *any* "good enough" solution – an *approximate* solution. Sometimes, the input will not be available all at once: We will see part of the input, be forced to make some decision, then continue. This is an *online* algorithm. Perhaps our algorithm can use *randomness*, or the input itself is random in some way, and the goal is to produce outputs that are good in expectation or with high probability.

2 Well-defined steps; the word-RAM model

Around the 1930s onward, mathematicians including Gödel, Church, Turing, Post, and Kleene began to resolve a thorny question: what does it mean for a function or quantity to be "effectively calculable"? The answer essentially boiled down to "there must exist an *algorithm* for calculating it." Various formalizations of algorithms arose, including the lambda calculus, μ-recursive functions, and Turing Machines.

A key component of any definition of "algorithm" is that the individual steps should require no discretion or creativity to carry out. Hence, an algorithm is a problem-solving procedure that can be mechanized or automated — implemented on a computer.

So generally, we formalize an algorithm with a particular model of computation that has a set of legal steps (commands, instructions, etc). We could use a Turing Machine in theory, but it wouldn't be very easy nor useful. We need a model of computation that is still mathematically analyzable, but much closer to modern machines, which are based on the *von Neumann* and *Harvard* architectures. It should have these features:

- The **algorithm**, a program represented as a fixed sequence of instructions.
- The **input** and space for the **output**, each stored separately.
- The **working memory**, initially blank, that the computer uses to store local variables and data structures.

The computer executes the algorithm one step at a time, interacting with the input, output, and memory as directed.[1]

Specifically, we define an algorithm in the **word RAM model of computation** as follows.

- The input, output, and working memory are each represented as arrays. Initially, the input array is written while the output and working memory are blank.
- Each entry in an array can store an integer or floating-point number[2], represented in binary, up to a certain bit length. We may also let entries store characters (e.g. ASCII or Unicode).
- The maximum number of bits per entry in the arrays is called the **word size**.

The algorithm is a sequence of instructions, each of which executes in constant time. Legal, constant-time instructions include:

- Arithmetic operations on any entry or entries of the arrays: add, subtract, multiply, divide, remainder, floor, ceiling. Note these operations occur in **constant time** for numbers that fit into the arrays. Exponentiation may require more care and should be analyzed separately.
- Condition checks and branches, such as **if** statements; the checks and branches in **while** and **for** loops; and subroutine calls.
- Accessing any element of an array, if we have its index (also called address) stored in a known location. This is the Random Access Memory (RAM) component of the model. If the arrays can have length m, then a location index requires $\lceil \log_2(m) \rceil$ bits to write down, so the word size must be at least this large.

Below is an example description of an algorithm. Here the local variables x and i will be stored in working memory, but it is not necessary to describe where exactly on the working memory array they are located. We just need to know that they require one memory slot each.

We will generally assume that all numbers fit in the word size. For example, in analyzing correctness of Algorithm 1, we would generally simply assume that x always fits into one array slot of the word RAM model.

[1]This is the simplest-possible, single-threaded model; it can be extended to include parallel processors, distributed computation, randomness, etc.

[2]We may sometimes pretend these are arbitrary-precision real numbers, for convenience.

Algorithm 1 Summing a list

```
Input: list $A$ of length $n$
$x = 0$
for $i = 1$ to $n$:
  $x$ += $A[i]$
Output $x$
```

Exercise 1. What is the difference between the following two problems in the word RAM model? What would the difference be in the input array, output array, and algorithm to solve them? *(a)* Adding two integers, both of which are of size less than 2^{w-1} where w is the word size. *(b)* Adding two integers, which could have arbitrary size.

Exercise 2. (Optional) Consider the problem of sorting a list of strings. Formalize this as a mathematically well-defined problem. What is the format of the input array and expected format of the output array on the word RAM model? (There are many possible answers.)

Exercise 3. (Optional) The *selection sort* algorithm works like this: go through the array and find the smallest element, and swap it with the first element. Now go through the rest of the array and find the smallest remaining element, and swap it with the second element. Continue until the end.
Formalize this algorithm in the word RAM model (i.e. write an algorithm analogous to Algorithm 1).

3 Why study algorithms?

Here's a brief set of answers:

- Understand performance and capabilities of algorithms you interact with.
- Understand kinds of problems and typical solutions or approaches.
- Designing new algorithms or variants.
- Preparation for more advanced or specialized topics, e.g. machine learning.
- Intellectually interesting and enjoyable! Fascinating questions and challenges.

4 Analyzing algorithms

Given an algorithm, we want to analyze **two properties:**

1. **Correctness:** does the algorithm solve the problem? I.e. given a well-formed input, does it always produce a correct output?
2. **Efficiency:** what *resources* does the algorithm use? The most common resources are **time** and **space**. (There can be others: randomness, communication, amount of access needed to the input data, ...) In the word RAM model, time is usually measured in terms of number of operations; space is the number of working-memory array slots used.

We will generally measure efficiency in the *worst case*, but we will also see analysis on average over a random process.

Because the word RAM model is only a rough approximation or abstraction, we don't want to obsess too closely over exact performance; big-O is usually enough to tell the key story.

4.1 Big-O notation: recap

An informal recap of big-O notation:

Symbol	**similar to**	**English meaning (up to a constant factor)**
$O(\cdot)$	$\leq$	asymptotically at most
$o(\cdot)$	$<$	asymptotically less; shrinking compared to
$\Omega(\cdot)$	$\geq$	asymptotically at least
$\omega(\cdot)$	$>$	asymptotically more; diverging compared to
$\Theta(\cdot)$	$=$	asymptotically the same

More formally, for two positive-valued functions f, g, we say:

- $f(n) \in O(g(n))$ if there exist positive numbers C, N such that, for all $n \geq N$, $f(n) \leq C \cdot g(n)$.
- $f(n) \in o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) \in \Omega(g(n))$ if $g(n) \in O(f(n))$.
- $f(n) \in \omega(g(n))$ if $g(n) \in o(f(n))$.
- $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$.

You may sometimes see big-O notation in equations and inequalities and it can take some work to interpret. For example, if someone writes

$$x + O(x^2) \in O(x^2)$$

or even

$$x + O(x^2) = O(x^2)$$

what they mean is "for any $f(x) \in O(x^2)$, the function $x + f(x)$ is in $O(x^2)$."

Try to avoid using big-O notation in equations like these unless it is very clear what you mean; when in doubt, explain.

Exercise 4. Let $f(n) = 2n^2 + 3n + 2$. True or false?

(a) $f(n) \in O(n^3)$.

(b) $f(n) \in o(n^3)$.

(c) $f(n) \in O(n^2)$.

(d) $f(n) \in o(n^2)$.

(e) $f(n) \in \Omega(n^3)$.

(f) $f(n) \in \Omega(n^2)$.

(g) $f(n) \in \omega(n^3)$.

(h) $f(n) \in \omega(n^2)$.

(i) $f(n) \in \Theta(n^3)$.

(j) $f(n) \in \Theta(n^2)$.

4.2 Proofs

In this class we must prove our claims about an algorithm's correctness or efficiency. For example, we may have a theorem saying that an algorithm always correctly sorts its input, or that it runs in time $O(n^2)$ in the word RAM model. As a refresher:

- All terms used must be clearly mathematically defined. If you are stuck, the first step is to always write the definitions of all terms involved.
- A proof is a series of steps.
- Each step makes a new claim and argues that it follows directly from things that have already been shown using a basic rule of logic.
- At the beginning of the proof, we have all facts already known and any assumptions made in the theorem statement.
- At the end, the final claim is "what was to be demonstrated", i.e. the theorem statement's claim.

Example. Recall Algorithm 1, which sums a list of numbers. Let us prove an efficiency claim.

Proposition 1. *Algorithm 1 runs in $O(n)$ time and uses $O(1)$ space in the word RAM model, where n is the length of the input array.*

Proof. The initial assignment $x = 0$ takes constant time. Each iteration of the loop involves checking if i exceeds n, accessing $A[i]$, adding x and $A[i]$, storing the result into x, incrementing i and storing the result, and branching back to the start of the loop. All of these are constant-time operations, so each iteration of the loop takes $O(1)$ time. The loop executes n times, so it takes a total of $O(n)$ time. Finally, writing x onto one cell of the output array takes constant time. These sum to $O(n)$ time.

Meanwhile, the only working memory the algorithm needs is for x and i. So it uses 2 working memory slots, i.e. $O(1)$ space. □

5 Designing algorithms

Understanding algorithms in general may seem a large, daunting task. Luckily, it's not the case that every new algorithmic problem must be solved from scratch in an ad-hoc. There are general categories of problems and toolsets for each category. In this class, we will cover "case studies" of some of the most important problems and tools in various categories. Along the way, we will learn general skills for rigorous design and analysis.

Outline of course topics

- **Module 1: Graph and Combinatorial algorithms**
  Graph search, dynamic programming, flows, matchings.
- **Module 2: Approximation, Online, and Randomized algorithms**
  Examples: greedy matching, ski rental problem, "online" secretary-type problems, hash tables, Bloom filters.
- **Module 3: Continuous, Linear, and Convex methods**
  Examples: random walks and PageRank, no-regret learning and zero-sum games, linear-programming applications.