Colorado CSCI 5454: Algorithms	August 27, 2019
Lecture 1	
Lecturer: Bo Waggoner	*Scribe: Bo Waggoner*

Review and Basics

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

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 calculating, specifically with decimal-system numerals; 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.

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

2 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 beginner algorithms courses, we generally study problems like sorting where the algorithm begins with a certain input, must convert it to a certain exact 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.

3 Well-defined steps; the word-RAM model

Developing algorithms to run on a Turing Machine would be not 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] up to a certain bit length.

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:

1. Arithmetic operations on any entry/ies of the arrays: add, subtract, multiply, divide, remainder, floor, ceiling. Note these operations occur in **constant time** for numbers that fit into the arrays.

2. Condition checks and branches, such as **if** statements; the checks and branches in **while** and **for** loops; and subroutine calls.

3. 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. This requires that the word size be at least $\lceil \log_2(m) \rceil$, where m is the maximum amount of input, working memory, and/or output used, so the index of any memory location can be stored.

For convenience, we will use named local variables and not carefully describe where on the working memory they are stored. Also, remember each a function call requires constant space to store the return address.[3]

This example gives a typical description of an algorithm:

Algorithm 1 Summing a list

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

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. This algorithm takes $O(n)$ time: 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. Finally, writing x onto one cell of the output array takes constant time.

The algorithm also takes $O(1)$ space: the only working memory it needs is for x and i.

Of course, there are some caveats. If the word size is too small compared to n and the size of the inputs, then x will "overflow" as the number stored in it grows too large for the bits available to represent it.

4 Why study algorithms

- Understand performance and capabilities of algorithms you interact with

[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.

[3] On most computing architectures these are both taken care of by the "stack", but here there is no need to go into these details.

- Understand kinds of problems and typical solutions or approaches
- Designing new algorithms or variants
- Preparation for more advanced or specialized topics (such as machine learning)
- Intellectually interesting and enjoyable! Fascinating questions and challenges.

5 Analyzing algorithms; big-O notation

We must mathematically analyze (1) correctness and (2) performance. Performance is in terms of how many resources they require to solve the given problem. The most important resources are time and space. 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 performance in the *worst case*, but we will also see analysis on average over some 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.

Informal recap:

Symbol	**similar to**	**English meaning**
$O(\cdot)$	$\leq$	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

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 $g(n) \in o(f(n))$.

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 unless it is very clear what you mean; when in doubt, explain.

6 Designing algorithms

Understanding algorithms in general may seem a large, daunting task. Luckily, it's generally not ad-hoc or case-by-case. We have general classes of problems and general algorithmic approaches, tools, tricks, etc. You will learn from "case studies" and specific important problems and algorithms; and also pick up general approaches.

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, secretary problem, 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