Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2024

Topic Notes: Introduction and Overview

Welcome to Design and Analysis of Algorithms!

What is an Algorithm?

A possible definition: a step-by-step method for solving a problem.

An *algorithm* does not need to be something we run on a computer in the modern sense. The notion of an algorithm is much older than that. But it does need to be a formal and unambiguous set of instructions.

The good news: if we can express it as a computer program, it's going to be pretty formal and unambiguous.

Example: Computing the Max of 3 Numbers

Let's start by looking at a couple of examples and use them to determine some of the important properties of algorithms.

Our first example is finding the maximum among three given numbers.

Any of us could write a program in our favorite language to do this:

```
int max(int a, int b, int c) {
  if (a > b) {
    if (a > c) return a;
    else return c;
  }
  else {
    if (b > c) return b;
    else return c;
  }
}
```

The algorithm implemented by this function or method has *inputs* (the three numbers) and one *output* (the largest of those numbers).

The algorithm is defined *precisely* and is *deterministic*.

This notion of determinism is a key feature: if we present the algorithm multiple times with the same inputs, it follows the same steps, and obtains the same outcome.

A *non-deterministic* procedure could produce different outcomes on different executions, even with the same inputs.

Code is naturally deterministic – how can we introduce non-determinism?

It's also important that our algorithm will eventually terminate. In this case, it clearly does. In fact, there are no loops, so we know the code will execute in just a few steps. An algorithm is supposed to solve a problem, and it's not much of a solution if it runs forever. This property is called *finiteness*.

Finally, our algorithm gives the right answer. This very important property, *correctness*, is not always easy to achieve.

It's even harder to *verify* correctness. How can you tell if you algorithm works for all possible valid inputs? An important tool here: formal *proofs* (you did some of those in Discrete Math, hopefully).

A good algorithm is also *general*. It can be applied to all sets of possible input. If we did not care about generality, we could produce an algorithm that is quite a bit simpler. Consider this one:

```
int max(int a, int b) {
   if (a > 10 && b < 10) return a;
}
```

This gives the right answer when it gives any answer. But it does not compute any answer for many perfectly valid inputs.

We will also be concerned with the *efficiency* in both time (number of instructions) and space (amount of memory needed).

Why Study Algorithms?

The study of algorithms has both *theoretical* and *practical* importance.

Computer science is about problem solving and these problems are solved by applying algorithmic solutions.

Theory gives us tools to understand the efficiency and correctness of these solutions.

Practically, a study of algorithms provides an arsenal of techniques and approaches to apply to the problems you will encounter. And you will gain experience designing and analyzing algorithms for cases when known algorithms do not quite apply.

We will consider both the *design* and *analysis* of algorithms, and will implement and execute some of the algorithms we study.

We said earlier that both time and space efficiency of algorithms are important, but it is also important to know if there are other possible algorithms that might be better. We would like to establish theoretical *lower bounds* on the time and space needed by any algorithm to solve a problem, and to be able to prove that a given algorithm is *optimal*. We would also like to be able to prove that some things are *impossible*!

Some Course Topics

Some of the problems whose algorithmic solutions we will consider include:

- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Sorting
- Program termination

Some of the approaches we'll consider:

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking and Branch and bound
- Space and time tradeoffs

The study of algorithms often extends to the study of advanced data structures. Most should be familiar; others might be new to you:

- lists (arrays, linked, strings)
- stacks/queues

- priority queues
- graph structures
- tree structures
- sets and dictionaries

Finally, the course will often require you to write formal analysis and often proofs.

Pseudocode

We will spend a lot of time looking at algorithms expressed as *pseudocode*.

Unlike a real programming language, there is no formal definition or standard "dialect" of "pseudocode". In fact, any given textbook is likely to have its own style for pseudocode.

Our text has a specific pseudocode style. I will aim to approximate the book's style, but sometimes my own style might drift to look more like Java or C code. Please try to do the same when you write pseudocode. It doesn't have to match the text exactly, but should be close.

The book's dialect:

- omits variable declarations
- indentation shows scope of `for`, `if`, and `while` statements (no curly braces!)
- arrow $\leftarrow$ used for assignment
- single = for equality comparison
- `//` used for comments
- no semicolons!

A big advantage of using pseudocode is that we do not need to define types of all variables or specify complex structures.