Computer Science 501
Data Structures & Algorithms
The College of Saint Rose
Fall 2014

Topic Notes: Introduction and Overview

Welcome to Data Structures and Algorithm Analysis!

Why Take Data Structures and Algorithms?

In this course, you will become a more sophisticated programmer and problem solver, as you learn about designing correct and efficient algorithms and data structures for use in your programs. Along the way, you will:

- hone your problem solving skills,
- gain experience in programming in general, Java in particular,
- learn how to implement algorithms and data structures in Java,
- learn how to evaluate and visualize data structures and algorithms,
- learn how to understand (and prove) some properties of data structures and algorithms,
- learn how to consider the relative merits of different structures and algorithms, and
- learn how to design large programs (in an object-oriented way) so that it is easy to modify them

I think of this as two courses in one - you become a more expert programmer with new data structures and algorithms, and you become a better computer scientist by analyzing those data structures and algorithms so you can design and use them efficiently and appropriately.

We will do very little with graphics and animations, instead choosing to focus on the relatively simple textual interface often used by advanced programmers. But the algorithms and data structures may be used in (and are often essential to) those graphical programs. Your additional programming experience will allow you to understand and make use of the extensive base of reusable code, Java and otherwise, that is available to today's programmers, even though we will use only a limited subset of those tools here.

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*.

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*.

Sample Problems

Here are some examples of the kinds of problems you will learn to solve. In some cases, we will consider algorithms at a high level. In others, we will consider them more carefully and analyze their efficiency. And in some cases, we will implement them.

1. Find items in a large collection with particular features (perhaps the 10 largest).
2. Find the shortest path from Albany to Albuquerque on the national highway system (and do it efficiently).
3. Develop a game decision tree to allow a computer player for a game such as chess.
4. Design and implement a scientific calculator.

5. Design and implement a simulator that lets you study traffic flow in a city or airport.
6. Design and implement a pattern matching system to find a particular sequence of nuceleotides in the sequenced DNA of a given organism.
7. Design and implement a simulation for some physical phenomenon (e.g., fluid flow).
8. Analyze solutions of problems such as the Towers of Hanoi.

Some of the approaches we'll consider include:

- 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. Some should be familiar; others likely will be new to you:

- lists (arrays, linked, strings)
- stacks/queues
- priority queues
- graph structures
- tree structures
- sets and dictionaries

Example: Greatest Common Denominator

We first consider a very simple but surprisingly interesting algorthmic example: computing a greatest common denominator (or divisor) (GCD).

Recall the definition of the GCD:

The gcd of m and n is the largest integer that divides both m and n evenly.

For example: gcd(60,24) = 12, gcd(17,13) = 1, gcd(60,0) = 60.

One common approach to finding the gcd is *Euclid's Algorithm*, specified in the third century B.C. by Euclid of Alexandria.

Euclid's algorithm is based on repeated application of the equality:

$$\text{gcd}(m,n) = \text{gcd}(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

More precisely, application of Euclid's Algorithm follows these steps:

Step 1 If n = 0, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

And a pseudocode description:

```
// m,n are non-negative, not both zero
Euclid(m, n) {

  while (n != 0) {

    r = m mod n
    m = n
    n = r
  }
  return m
}
```

It may not be obvious at first that this algorithm must terminate.

How can we convince ourselves that it does?

- the second number (n) gets smaller with each iteration and can never become negative
- so the second number in the pair eventually becomes 0, at which point the algorithm stops.

Euclid's Algorithm is just one way to compute a GCD. Let's look at a few others:

Consecutive integer checking algorithm: check all of the integers, in decreasing order, starting with the smaller of the two input numbers, for common divisibilty.

Step 1 Assign the value of min$\{m,n\}$ to t

Step 2 Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4

Step 4 Decrease t by 1 and go to Step 2

This algorithm will work. It always stops because every time around, Step 4 is performed, which decreases t. It will eventually become t=1, which is always a common divisor.

Let's run through the computation of gcd(60,24):

Step 1 Set t=24

Step 2 Divide m=60 by t=24 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=23, proceed to Step 2

Step 2 Divide m=60 by t=23 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=22, proceed to Step 2

Step 2 Divide m=60 by t=22 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=21, proceed to Step 2

Step 2 Divide m=60 by t=21 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=20, proceed to Step 2

Step 2 Divide m=60 by t=20 and check the remainder. It is 0, so we proceed to Step 3

Step 3 Divide n=24 by t=20 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=19, proceed to Step 2

Step 2 Divide m=60 by t=19 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=18, proceed to Step 2

Step 2 Divide m=60 by t=18 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=17, proceed to Step 2

Step 2 Divide m=60 by t=17 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=16, proceed to Step 2

Step 2 Divide m=60 by t=16 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=15, proceed to Step 2

Step 2 Divide m=60 by t=15 and check the remainder. It is 0, so we proceed to Step 3

Step 3 Divide n=24 by t=15 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=14, proceed to Step 2

Step 2 Divide m=60 by t=14 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=13, proceed to Step 2

Step 2 Divide m=60 by t=13 and check the remainder. It is not 0, so we proceed to Step 4

Step 4 Set t=12, proceed to Step 2

Step 2 Divide m=60 by t=12 and check the remainder. It is 0, so we proceed to Step 3

Step 3 Divide n=24 by t=12 and check the remainder. It is 0, so we return t=12 as our gcd

However, it does not work if one of our input numbers is 0 (unlike Euclid's Algorithm). This is a good example of why we need to be careful to specify valid inputs to our algorithms.

Another method is one you probably learned in around 7th grade.

Step 1 Find the prime factorization of m

Step 2 Find the prime factorization of n

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd(m,n)

So for our example to compute gcd(60,24):

Step 1 Compute prime factorization of 60: 2, 2, 3, 5

Step 2 Compute prime factorization of 24: 2, 2, 2, 3

Step 3 Common prime factors: 2, 2, 3

Step 4 Multiply to get our answer: 12

While this took only a total of 4 steps, the first two steps are quite complex. Even the third is not completely obvious. The description lacks an important characteristic of a good algorithm: *precision*.

We could not easily write a program for this without doing more work. Once we work through these, it seems that this is going to be a more complicated method.

We can accomplish the prime factorization in a number of ways. We will consider one known as the *sieve of Eratosthenes*:

```
Sieve(n) {
  for p = 2 to n { // set array values to their index
    A[p] = p
  }
  for p = 2 to floor(sqrt(n)) {
    if A[p] != 0 { //p hasn't been previously eliminated from the list
      j = p * p
      while j <= n  {
        A[j] = 0  //mark element as eliminated
        j = j + p
      }
  }
  // nonzero entries of A are the primes
```

Given this procedure to determine the primes up to a given value, we can use those as our candidate prime factors in steps 1 and 2 of the middle school gcd algorithm. Note that each prime may be used multiple times.

So in this case, the seemingly simple middle school procedure ends up being quite complex, since we need to fill in the vague portions.