SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2018

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*.

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. You will practice your technical writing. As part of this, you may wish to gain experience with the mathematical typesetting software LaTeX.

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 looks 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.

Counting Basic Operations: Bubble Sort

A sorting algorithm so simplistic, you might not have even studied it previously is called *bubble sort*. Here's a version.

ALGORITHM BUBBLESORT(A)
  //Input: an array $A[0..n-1]$
  for $i \leftarrow 0..n-2$ **do**

for $j \leftarrow 0..n-2$ **do**
  if $A[j+1] < A[j]$ **then**
    swap $A[j+1]$ and $A[j]$

Let's simulate it!

The comparison and swap operations are examples of *basic operations* of the algorithm. We often wish to count the number of times these basic operations occur to help us analyze the cost of an algorithm. We will focus here on the comparison, since it happens every time the inner loop iterates.

(To be computed in class: each `for` loop becomes a summation!)

Counting Basics

Let's recall some old favorites from Discrete Math:

$$\sum_{i=5}^{8} 1 =$$

$$\sum_{i=l}^{u} 1 =$$

$$\sum_{i=5}^{8} N =$$

$$\sum_{i=l}^{u} N =$$

Also recall that summations can take advantage of associativity:

$$\sum_{i=l}^{u} (a+b) = \sum_{i=l}^{u} a + \sum_{i=l}^{u} b$$

And a couple others worth remembering/thinking about:

$$\sum_{i=1}^{N} i = 1 + 2 + 3 + ... + N =$$

$$\sum_{i=l}^{u} i =$$

Improving Bubble Sort

When we simulated the version above, there were some comparisons made that we know never would result in any swaps.

Let's think about how we can change the bounds of the inner loop to avoid making those comparisons:

ALGORITHM IMPROVEDBUBBLESORT(A)
  //Input: an array $A[0..n-1]$
  for $i \leftarrow 0..n-2$ **do**
    for $j \leftarrow 0..$ [ ] **do**
      if $A[j+1] < A[j]$ **then**
        swap $A[j+1]$ and $A[j]$

In the lab exercise for today, you will analyze the improved version.

Example: Greatest Common Denominator

We first consider a very simple but surprisingly interesting 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.