

Complexity Theory and Algorithms

The study of algorithms is often associated with devising *efficient* methods for finding a solution for tasks. (Indeed, the online Merriam-Webster dictionary defines “algorithm” as (1) “a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation” and (2) “a step-by-step procedure for solving a problem or accomplishing some end especially by a computer”.)

Adding to this, we might demonstrate that, for some problems, it is *unlikely* that there is an *efficient* algorithm that will work *all the time*. (This is generally considered part of “complexity theory” and deals partially with the so-called theory of what it means for a problem to be “NP-complete”.)

This begs the question “What does it mean for an algorithm to be *efficient*?” One of the goals of this course is to try to answer this question, to help you to recognize efficient algorithms, and to give you familiarity with how to design (efficient) algorithms, and, to some extent, to recognize that there are some problems for which efficient (general) algorithms may not exist.

For examples of the kinds of problems we will consider in this course, I give the following list. By the end of this course, we will likely have discussed methods for finding solutions for most of these problems, as well as examining the efficiency of those algorithms in terms of the *running time* of the algorithms, i.e. how many operations will be necessary to find the solution, expressed in terms of the *input size* of the problems.

1. Bridge crossing (at night)

(You probably saw this problem in COMP 108, but not the generalized version of it.)

Four people want to cross a bridge at night. A group of *at most two* people may cross at any time, and each group must have a torch so they can see clearly on the bridge. Only one torch is available, so some sort of shuttle arrangement must be arranged in order to return the torch so that more people may cross.

Each person has a different crossing speed. One person takes one minute to cross, another takes two minutes, the third takes five minutes, and the last one takes ten minutes to cross. The speed of a group is determined by the speed of the *slower* member.

Your job is to determine a strategy that gets all four people across the bridge in the *minimum time*. (Find a *schedule* to get the people across in the minimum time. And what is the minimum time required?)

How about if you have six people, whose crossing times are 2, 3, 4, 6, 9, and 10 minutes? Can you generalize your procedure to handle n people (some of the crossing times might even be equal, but this causes no difficulties in terms of finding a solution)?

2. Item Testing

Suppose that you’re involved in a Health and Safety testing operation for a new glass jar. There is a ladder that has $n > 1$ rungs, and you wish to

determine the highest rung from which you may drop this new jar without it breaking (up to the maximum of n , at which point you stop if it hasn't broken after that test).

If you're given only one jar for testing purposes, you might need to perform n different tests. In other words, the only way to determine the maximum (ladder) height is by starting on the bottom rung and dropping the jar from that height. If it survives, then you try it from the second, third, etc. until you find the point at which it survives a drop from rung k but breaks at rung $k+1$. (We're assuming that the jar is unaffected if it doesn't break, i.e. it's not "cracked" or otherwise harmed if it doesn't completely break apart.)

Suppose now that I give you two jars with which to perform this testing. Design a procedure to perform this experiment using the *fewest number* of tests. In other words, given two jars, what is the smallest possible number of tests you might need to determine the maximum height (i.e. which rung on the ladder) from which the jar can be dropped without breaking? How about if I give you three jars you may use for the tests?

(Hint: In this case, it's very important that the maximum height n is specified *beforehand*.)

3. Buying postage

Consider a post office that sells stamps in three different denominations, 1p, 7p, and 10p. I don't like the taste of the glue on the stamps (these aren't the self-adhesive type), and there's no faucet nearby to moisten the stamps.

So given the value of postage, i.e. I need to mail a package that requires postage of N pence, I want to find the *minimum number* of stamps necessary for my package. (And how many of each type do I need to buy?) Help me out with my difficulties here by finding a procedure to determine this.

4. Making change

(This is somewhat related to, yet different from, the preceding problem.)

American coins come in five different values, namely 50-cent, 25-cent, 10-cent, 5-cent, and 1-cent coins. Therefore, if we wanted change for 11 cents (or 11¢), we can do this with one 10¢ coin and one 1¢ coin, or two 5¢ coins and one 1¢, or one 5¢ and 6 1¢ coins, or with eleven 1¢ coins. So there are four ways to give change for 11¢.

How many ways are there to make change for N ¢? (And how can you compute this number *efficiently*?)

5. Majority finding

Suppose the L is a list of integers. A *majority element* for L is an integer that appears more than half the time, i.e. if L has n integers in the list, a majority element appears strictly *more than* $n/2$ times in the list.

Your task is to write pseudo-code for an algorithm that will find a majority element for a list L , if one exists.

As stated, this problem isn't very interesting. The following restrictions make the problem much more interesting, namely

- (a) You can only use a constant amount of memory, i.e. you can only store three or four (or some small number of) intermediate values, counters, etc. (So you *can't* have a counter for each possible value that appears in L , and don't even know beforehand what the smallest/largest elements on L might be.)
- (b) You can only read through the list L twice.

Under these restrictions, write a majority finding algorithm that will either find a majority element for L , or prints out the conclusion that no such element exists.

(Hint (maybe?): The idea is that after you've read the list once, you then have a *candidate* value, x , for a majority element. The second time you read L , you're then checking to see if x is actually a majority element or not.)

6. Increasing/decreasing subsequences

If I give you a sequence of $n^2 + 1$ distinct integers, it is a fact that there is either an *increasing subsequence* of (at least) $n + 1$ integers or a *decreasing subsequence* of (at least) $n + 1$ integers (or perhaps both).

For example, in the sequence of ten numbers below

$$6, 4, 3, 7, 1, 10, 2, 5, 8, 9$$

we see there is an increasing subsequence of length four (namely 4, 7, 8, 9 but 4, 5, 8, 9 or 6, 7, 8, 9 also work; we also see that 1, 2, 5, 8, 9 is an increasing subsequence of length five). Note also in this case that 6, 4, 3, 2 is a decreasing subsequence of length four.

How can you *find the length* of a longest increasing (or decreasing) subsequence in a given sequence of numbers?

(Note: The opening statement can be generalized as follows.

If I give you a sequence of $m \cdot n + 1$ distinct integers, there is either an increasing subsequence of length (at least) $m + 1$ integers, or a decreasing subsequence of length (at least) $n + 1$ integers (or perhaps both).

The mathematical proof of this fact relies on the *Pigeonhole Principle*, and is a proof by contradiction.)

7. The Game of Fif

(First described by mathematician Martin Gardner, given its name by Alex Bogomolny. Less of an algorithmic question, and one of recognition of this game as another, likely more familiar, game. This isn't really an algorithmic problem, but more of a puzzle of sorts.)

Fif is a two player game. The "board" consists of a row of integers 1 to 9. Players take turns selecting integers by checking the box below them (an integer can only be selected by one player).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | | | | | | | | |

The goal of the game is to select integers such that, amongst those selected by a single player, there will be a *triplet* of numbers that sums to 15 (hence the name of the game). The player who achieves this first wins the game.

Does the player who goes first always win this game? Or can the second player win? Can there be draws? Devise a strategy to play this game (either if you are the first player or the second player).

8. Fast Exponentiation

(You may have seen this kind of question elsewhere, say in COMP 108 or COMP 109.)

In some applications of cryptography (secret-sharing, if you like), we need to compute expressions like $x^{37} \bmod 60$ repeatedly for different values of x . (Recall $x^{37} \bmod 60$ means the *remainder* when x^{37} is divided by 60.)

You can do this with 36 multiplications, namely compute $x^2 = x \cdot x$, then $x^3 = x \cdot x^2$, then $x^4 = x \cdot x^3$, etc. (Or, more precisely, compute $x^2 \bmod 60$, then compute $x^3 \bmod 60$, etc.) However, 36 multiplications, while not “very many” if you have to do it once, quickly becomes very many when you have to do this several thousand or several million times.

How can you compute $x^{37} \bmod 60$ using (many) *fewer than* 36 multiplications? (And don’t say you can do it with “zero multiplications” by performing a whole lot of addition operations, as this would be even slower.)

Can you generalize your procedure so that you could find $x^k \bmod n$ for given values of x, k , and n ?

We will talk about one encryption/decryption scheme later in this course, the so-called RSA method, that uses this exact type of operation, so it’s important to find a quick way to do “modular exponentiation”.

9. Deal or No Deal (sort of, well not really...)

(We will likely *not* talk about the solution to this problem as it’s more mathematical than algorithmic in nature...)

I shuffle and deal out a set of ten cards facedown, each of which shows an amount of money (on the hidden side of the card). You can turn over as many cards as you like, stopping at any time you want, but you only win the amount of money shown on the *final card* that you turn over.

What is your best strategy to play this game?