How Long Does Computation Take?
Draft Notes on the Efficiency Algorithms for MA/CS 109
Leo Reyzin with the help of Nick Benes

We will now stop talking about what we can’t do with computers and start talking about what we can do. In the homework we are looking at how to give a computer directions to do a certain task. We call a set of instructions like this an algorithm (not to be confused with a logarithm).

Definition: An algorithm is a set of sufficiently precise instructions for getting some output from some input.

Algorithms can be written in a variety of ways: as text, in a programming language, as bits. A competent programmer can convert a text description of an algorithm into a program in some programming language; special programs called compilers (or, in some cases, interpreters) convert programming language programs into bits your computer actually executes.

The Giving Change Algorithm
Here’s an example (essentially from the homework problem on giving directions to cashiers on how to give change in binary):

• Input: A positive integer $N$.
• Output: A set of coins from $\{1, 2, 4, 8, 16, 32, 64, \ldots\}$ that add up to $N$, with no coin used twice.

Consider the following two methods, i.e. two algorithms, for determining the set of coins we need.

• (Method 1:) Count out the change using $N$ pennies. Then replace pairs of pennies with 2 cent coins until you have 0 or 1 penny left (i.e. there are no more pairs of pennies left). Then replace pairs of 2 cent coins with 4 cent coins, then pairs of 4 cents with 8 cent coins, etc. Repeat until there are no pairs of any coin left, so you have no more than one of each coin.

• (Method 2:) Make a list of powers of 2 (i.e. of coin values) until the first one that exceeds $N$. Go down that list starting from the largest coin value. Then
  1. Give the largest coin whose value is not more than $N$; subtract the value of this coin from $N$.
  2. If $N > 0$, go back to step 1.

When we’re comparing possible algorithms for a program, the main thing we look at is their running times, i.e., how long they will take. One of the nice things is that we don’t even have to write the program to get a good idea for how long it will take. We do this by
counting the number of operations involved. Let’s compare the two methods for the input 109:

**Method 1:**

1. Take 109 pennies.
2. Make 54 replacements to get one penny left
3. Make 27 replacements to get no 2 cent coins left
4. Make 13 replacements to get one 4 cent coin left
5. Make 6 replacements to get one 8 cent coin left
6. Make 3 replacements to get no 16 cent coins left
7. Make 1 replacement to get one 32 cent coin left, plus the new 64 cent coin.

This gives us a total of 104 replacements, plus counting the initial 109 pennies.

**Method 2:**

1. Make the list of powers of 2 by adding, and stop when we get above 109:

   \[
   
   \begin{align*}
   1 + 1 &= 2 \\
   2 + 2 &= 4 \\
   4 + 4 &= 8 \\
   8 + 8 &= 16 \\
   16 + 16 &= 32 \\
   32 + 32 &= 64 \\
   64 + 64 &= 128
   \end{align*}
   \]

   7 additions to get past 109.

2. 64 is the biggest power of 2 less than 109: 109 − 64 = 45
3. 45 − 32 = 13
4. 16 is bigger than 13, so we skip it.
5. 13 − 8 = 5
6. 5 − 4 = 1
7. 1 is on the list.
So how many operations? We started with 7 additions, we did 4 subtractions, and we made 7 comparisons with the list along the way.

We have a bit of a problem here. We have replacements and counting as our operations in the first method, but addition, subtraction, and comparison in the second. These different operations may take different amounts of time, and we might worry about comparing them. But for our purposes now they take roughly the same amount of time, so we’ll just count operations. We have a total of 213 operations for the first method, but only 18 for the second (which is why it doesn’t matter if it takes a little longer to add than to count — we just care about the order of magnitude, a rough estimate).

But we might want to use this algorithm for lots of other inputs, and we don’t want to repeat all of these calculations for every possible input. Instead, we generalize our counting for the case when our input is some number $N$:

Method 1: Between $N$ and $2N$. (In fact, it’s clear that it will take at least $N$ steps to just count all the pennies; we will not show here why all the exchanges will not add more than another $N$ steps—it’s a good exercise for the reader.)

Method 2: Between $2 \log_2 N$ and $3 \log_2 N$. (Because we need $\log_2 N$ additions, $\log_2 N$ comparisons, and at most $\log_2 N$ subtractions.)

Think back to the function zoo for a minute. We have an analogy: log is to linear as linear is to exponential. So from the point of view of log, a linear function is HUGE! So method 1 takes much longer. For example, suppose we want to figure out how to write down the population of the whole world ($\approx 6$ billion people). Now

$$
\log_2 1,000 \approx 10 \\
\log_2 1,000,000 \approx 20 \\
\log_2 1,000,000,000 \approx 30 \\
\log_2 \text{human population in 2009} \approx 33
$$

So now you’ve seen examples of algorithms, as well as how we can do the same thing with different algorithms and how a shorter running time can make one algorithm considerably better than another.

Search algorithms

Let’s consider a different problem now. We’ve talked a bit about how much information, how many bits, there are in the world now. There are a lot. Let’s talk about algorithms for finding information. How do you find information when you look at a book? You go to the index. The same happens with computers. When you search for something using a search engine (such as Google), it goes to an index that it has already compiled and tells you what pages match.

Finding things in an index is essentially the following problem:

**Problem:**

- **Input:**
  - Index of $N$ keywords
  - Search word
Consider the following two algorithms.

- **Linear Search**: Go through the index one word at a time comparing against the search word. (Note that the list doesn’t have to be sorted for this to work.)

- **Binary Search**: Usually when you look for something in an index, you don’t go through the index word by word. You know the index is in alphabetical order, so you go to about where you expect the first letter to be and you go from there. But that requires some intuition that computers don’t have, and it can be hard to translate that intuition for a computer. So let’s think about how a computer can look for a keyword in an ordered index. We can have it do something like this:
  
  - Look in the middle of the list
    * If that’s your search word, done!
    * If it’s bigger than your search word, repeat on the left half.
    * If it’s smaller than your search word, repeat on right half.
  
  (Note that the list must be sorted for this to work.)

On average, linear search will reach your word around half way through the list, so the running time is about \( N/2 \) comparisons. Binary search, on the other hand, will work much faster. You cut the list in half each time. And the number of times you can cut \( N \) in half before you reach 1 is \( \log_2 N \). So binary search will take only \( \log_2 N \) comparisons. Thus, if you have a list with a billion keywords, linear search will take about 500,000,000 comparisons whereas binary search will take about 30. If you are running a search engine with many queries, this difference is crucial in reducing the amounts of computers and power you have to buy.

There is lots more we can do with binary search. For example, we can use binary search to find the square root of 109:

- 55 is about halfway to 109. But \( 55^2 > 109 \), so we need something less than 55.
- 27 is about halfway between 0 and 55. But \( 27^2 > 400 > 109 \), so we need something smaller still.
- 13 is about halfway between 0 and 27. And \( 13^2 = 169 > 109 \), so we need something smaller again.
- 7 is about halfway between 0 and 13. \( 7^2 = 49 < 109 \), so we want something bigger.
- 10 is halfway between 7 and 13. \( 10^2 = 100 < 109 \), so we need a bigger number.
- Now we’ll start getting more precise (actually we could have been doing this all along, but these numbers were easier). \( 11.5 \) is halfway between 10 and 13, and \( 11.5^2 = 132.25 > 109 \). So we’ll go smaller and try 10.75 and keep going until we reach whatever precision we want (say, 10.44). And it works fairly quickly.
Before we leave the topic of binary search, remember that it requires that we already have an ordered index. Sorting a list, i.e. putting it in order, is another challenging problem in computer science. We tackle it next.

Sorting

We now consider the following problem: given a list $L$ of $N$ words, sort them alphabetically. Again, consider two algorithms.

- **Selection Sort**
  1. Initialize $R$ to be an empty list.
  2. Find the alphabetically first element in $L$, remove it from $L$ and place it at the end of $R$.
  3. If $L$ is not empty, go back to step 2.

- **Merge Sort**
  1. If $L$ has just one element, you are done sorting it.
  2. Split $L$ into two halves of equal size (if $L$ has an odd number of items, then one half will, of course, have one more word than the other).
  3. Sort the left half using Merge Sort.
  4. Sort the right half using Merge Sort.
  5. Merge the two sorted halves together into a single sorted list.

We will analyze both sorting algorithms by counting how many comparisons they perform. This number dominates all the other operations.

First let’s analyze selection sort. Finding the alphabetically first element (step 2) is done as follows. Let $m$ denote the alphabetically first element known so far. Start by setting $m$ to the first element on the list. Then go through the list in order, comparing each element with $m$, and replacing $m$ if you find something alphabetically earlier than $m$. At the end of the trip through the list, $m$ will have the alphabetically first element. Thus, the first time through $L$ we will make $N - 1$ comparisons. Then $L$ gets shorter (because one element is removed), and so the second time we will need only $N - 2$ comparisons. And so on. Adding up all the comparisons needed, we will get $(N - 1) + (N - 2) + \cdots + 1 = N(N - 1)/2 \approx N^2/2$ comparisons.

Now let’s analyze merge sort. First, our description may seem a bit strange, because it describes merge sort in terms of merge sort. But note that the description is in terms of sorting a smaller list—and eventually, the list will get down all the way to size 1, which we know how to sort. (It would also be ok to stop at size 2 and then sort the two elements by swapping them if necessary. Stopping at size 1 just makes for a more concise description.)

Next, we need to understand how to merge two sorted lists. The procedure is as follows. Given two sorted lists $A$ and $B$, to produce a merged list $C$, simply compare the two elements at the head of $A$ and $B$. Whichever is smaller is removed from its list and goes to the end.
of \(C\). This process is repeated until either \(A\) or \(B\) runs out—and whichever doesn’t run out is moved to the tail of \(C\).

Counting how many comparisons merge sort takes is tricky. Note that every element will participate in a merge about \(\lceil \log_2 N \rceil\) times (once for lists of size \(N\), once for lists of size \(N/2\), etc.). And each time an element participates in a merge, it moves to the merged list exactly once. (Note that it may get compared a lot—some elements get compared just once and move, while other may get stuck as heads of their lists for a while and get compared with many elements on the other list. However, it will be moved exactly once, no matter how many times it’s compared.) Since there are \(N\) elements, the total number of element moves is \(N \cdot \lceil \log_2 N \rceil\). And there cannot be more comparisons than moves, since every comparison results in a move. Thus, merge sort takes at most \(N \cdot \lceil \log_2 N \rceil\) comparisons.

To understand the difference of the two running times, consider a possible task facing the Social Security Administration: sort the list of all Americans (\(N \approx 300,000,000\) people) by name (or by social security number, it doesn’t matter). Selection sort will take about \(N^2/2 = 45,000,000,000,000,000\) (that’s 45 quadrillion) comparisons. Even if they can do a million comparisons a second, it will take about \(45,000,000,000,000,000/60/60/24/365/365/2 = 1,427,000,000,000\) ≈ 1,427 years.

On the other hand, merge sort will take about \(N \cdot \lceil \log_2 N \rceil = 300,000,000 \cdot 29 \approx 9,000,000,000\) comparisons. 9 billion is not a small number, but at least its feasible: doing a million comparisons a second, we can be done in \(9,000,000,000/60/60/1 = 2.5\) hours; doing a hundred thousand comparisons a second, we can be done in a bit over a day.