Algorithms, Data Structures

This is CS50. Harvard University. Fall 2015.

Cheng Gong

Table of Contents

1. Programming

- Conditions are an if-then statement, like a fork in the road.
- Boolean expressions are expressions that can be true or false, used to make decisions within a condition.
- Variables are used to store some value or piece of data, like a counter that stores a number.
- Loops are parts of code that happen again and again.
- A function is a group of code that does some action and is used in a bigger program to accomplish a larger goal.
  - # For example, Uber might use functions implemented by Google to display a map in their app.
- A multithreaded program can do multiple things "at the same time," even if it only has one CPU.
- Events are like messages sent between threads or programs when something happens, so they can respond to them.
- If you want to see classmates' Project 0's, we have a folder here[1].

[1] https://scratch.mit.edu/studios/1944849/

2. Data Structures, Algorithms

2.1. Sorting

- Today we move from lower-level programming details to higher-level ways to solve problems more generally.
- For example, when we looked for Mike Smith in the phone book we assumed that the phone book was already in alphabetical order.
  - # And if the phone book wasn't sorted, we'd have to look page by page.
- So we have a deck of cards, a volunteer shuffles them, and a volunteer comes up to sort them.
  - # Our volunteer is separating the deck into suits first, and then sorting them, because the piles are smaller and more manageable.
- Let's start with the numbers `1-8` in some random order:

```
6 3 4 8 7 1 2 5
```

- If we were to sort this as humans, we might start by looking at the first two numbers and fix them by swapping them if they're out of order:

```
3 6 4 8 7 1 2 5
```

- Now let's look at the next two, and swap the `6` and `4`

```
3 4 6 8 7 1 2 5
```

- We repeat this until we get to the end:

```
3 4 6 7 1 2 5 8
```

- But we're not done yet, so we'll keep going until all the numbers are sorted. The biggest number will be swapped to the right every time, until eventually all the numbers are sorted.
- For example, the next pass will bring us to:

```
3 4 6 1 2 5 7 8
```

- So now we can ignore the last two elements, and repeat this until we're done. And we'll know that we're done when we've passed through the list once without making any swaps.
- What we just did is called bubble sort, and it is on the order of *n*^2 for running time. There are *n* elements (8 in this case), and we want to consider the worst case (where the elements are in reverse order, so we have to do the most work). So we need to make *n* - 1 swaps the first pass, and *n* - 2 swaps the next pass (if we make the optimization of skipping the last element, then the last two elements, and so on, since we know they'll be in the right place), until we finally make the last one swap. So the total number of swaps will be (*n* - 1) + (*n* - 2) + … + 1, which, we can look up, adds up to (*n* - 1)*n*/2. And as *n* increases in size, the biggest term will be the *n*^2 part, so the algorithm is on the order of *n*^2, #(*n*^2).
- We can look at a different approach. Let's make another list:

```
18 16 11 14 15 17 13 12
```

- We'll go down the list and remember what the smallest element is, in this case `11`. But since we have a fixed number of spots in our list, we'll just swap it with the first element in our list:

```
11 16 18 14 15 17 13 12
```

- Now we can start in the second spot, and look for the smallest element from there, and repeat what we did:

```
11 12 18 14 15 17 13 16
```

- But now it seems we're moving `16` a bit too far to the right. But remember that if the list is randomly sorted at the beginning, then the swaps we make will average out to make no difference.
- Eventually we'll have our sorted list, but this approach too is on the order of *n*^2 since we're making *n*, then *n* - 1, then *n* - 2, etc. checks to find the smallest element on each pass, and doing that *n* times. This sorting algorithm is called selection sort.
- Let's look at yet another approach with a new list.

```
7 4 3 1 2 8 6 5
```

- Perhaps we can fix things as we go. So `7` is sorted if we consider it by itself. So now let's look at the next number, `4`. So we can move the `7` to the right and put `4` in front.

```
4 7 3 1 2 8 6 5
```

- Then we look at the `3`, and we can remove the `3` and shift the `4` and `7` down:

```
3 4 7 1 2 8 6 5
```

- But in the worst case scenario, we would have to shift 1, then 2, then 3, and eventually *n* - 1 elements for each of the *n* elements, which would end up taking the same order of steps as do our other approaches so far. This approach is called insertion sort.
- Just to recap, bubble sort, selection sort, and insertion sort are all on the order of *n*^2, #(*n*^2).
- But if we knew how close our list was to being sorted, some of them might do better.
- When we talked about the upper bound of running time, we used "big #" notation. We can also talk about the lower bound, or how fast an algorithm is in the best case, using "big Omega" notation with the symbol Ω.
- The worst case scenario for these sorts is the case where the list is backwards, and the best case is when the list is already sorted. Then, insertion sort will still have to look at the list at all *n* elements, but not need to do any shifting. But selection sort would have to check the entire (unsorted part of the) list to find the smallest element each time, for *n* times, and have the same running time in the best case as in the worst case. And bubble sort still needs to compare all the pairs the first time, but once it goes through the list without making any changes, it would stop and be done.
- So to summarize, bubble sort has Ω(*n*), selection sort Ω(*n*^2), and insertion sort Ω(*n*). So bubble sort and insertion sort are asymptotically the same, whereas the algorithms are roughly the same for really big *n*.
- We take a look at this visualization[2] to get a sense of how these, and other algorithms, sort.
- Merge sort, on the order of *n* log *n*, is an asymptotically faster algorithm whereby the list is divided in half over and over again, and each half is sorted before being merged back together.

[2] https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

- Let’s look at how bubble sort might be implemented in a programming language called Java:

  # We have some variables at the top, and it looks like most of the code is in a `while` loop, which will repeat as long as the variable `flag` is set to true.

  # Within the loop, we first set the flag to `false`, to indicate that we haven’t made any swaps yet. Then we have a `for` loop, which is a loop that repeats some number of times, in this case for the length of the list.

  # Within the `for` loop, we have a condition where we’re comparing two items, and it looks like we’re swapping elements and then marking that we made a swap.

  # So we see two loops, one within another, where the `for` loop runs *n* times if there are *n* items in our list, and the `while` loop could run for up to *n* times in the worst case, which means that there would be roughly a total of *n*^2 steps before this algorithm finishes.

2.2. Data Structures

- Inside a computer, we have a CPU that performs calculations, and two main types of memory, RAM and hard disk. RAM is temporary but faster, but hard disks can store data more permanently.
- When we open an application on our computer, the data is loaded from the hard disk into RAM so we can work with it more quickly.
- At the lowest level, memory can be thought of as a grid of storage locations, where each rectangle in the grid represents one byte of space. And if we have a billion of those rectangles, then we have a billion bytes, or a gigabyte of memory.
- Each of those rectangles are labeled with numbers, from 0, 1, 2, 3, etc.
- The term "32-bit integer" means that every integer takes up 4 of those rectangles in memory (and will overflow if it’s too big to fit into that space).
- So imagine we are trying to sort a list of 8 numbers, and we only have 8 bytes of space in memory to hold the list because the rest is being used by other programs. But if we want to sort these numbers with the help of extra memory, we need to allocate more memory. And this happens with a concept called virtual memory, where the operating

system (like Windows or Mac OS) copies some memory, that isn’t being used, from RAM to hard disk. Then that area of memory can be used by our program. And if we switch to another program that needs this memory back, that program’s memory will be copied back from hard disk to RAM. This process of swapping memory is what causes our computers to slow down when we have lots of programs running, because hard disk is much slower than RAM.

- In any case, a continuous block of rectangles (like our list of 8 numbers) is called an array in many programming languages. We ask for a set number of bytes in advance, since we need to find either free memory of that size or to move other programs' data to disk.
- So if we don’t know how much space we need in advance, we might ask for a small chunk of memory first. We’ll fill the small chunks with our data, and then connect them to each other by remembering where each chunk is. All the chunks might be in different parts of memory, but we can link them by also storing the address of the next chunk. In many programming languages, these addresses are called pointers, since they point to another location in memory that contains data. The upside here is that we can flexibly extend and shrink how much memory we’re using in this data structure, which is called a linked list, but the downside is that we don’t know where our data is unless we start of the beginning each time and follow the pointers to the next location one at a time.
- We’ll cover hash tables and trees next time, but conclude by watching this video[3] on what sorting algorithms sound like!

[3] https://www.youtube.com/watch?v=t8g-iYGHpEA