Week 3

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Announcements

- CS50 Lunch this Friday, 9/25 - RSVP if interested at cs50.harvard.edu/rsvp[1].
- Wednesday's lecture is online only.
- Problem set 2 is out, and we strongly suggest that you start as early as possible! You never know when you'll encounter a bug or something that will block you, and much better to have plenty of time to get unblocked.

2. Searching and Sorting

- Volunteer Alan repeats David's phone book example, tearing the problem in half repeatedly to find Mike Smith.
- This method is much faster than reading one page at a time (linear search), but requires us to leverage the assumption that the phonebook is sorted!
- How do we get something like the phonebook in sorted order?
- Volunteer Caroline sorts a stack of blue books from A-Z by taking each book off the pile and placing it in the first or second half of the alphabet, then merging the two piles.
- Volunteer Trevor is presented with seven "doors" and tasked to find the number `50` behind one of them.

[1] http://cs50.harvard.edu/rsvp

    # Because the inputs (the numbers behind the doors) were in a random order, the best we can do is to pick doors randomly (or sequentially) to find the number `50`.
    # If the inputs are sorted, Trevor notes that `50` is the largest number in the list and so it'll be at one end or the other (depending on whether we've sorted small-to-big or big-to-small).

- Last year, volunteer Ajay found `50` in the unsorted list in just one click, which didn't quite convey the advantage of sorting…
- In a video from quite a few years ago, we see Sean, another volunteer, search for the number `7` in a much more linear fashion.

3. Sorting Algorithms

- The algorithms we'll use for sorting are fundamentally the same whether we have eight, a thousand, millions, or billions of inputs.
- Eight volunteers come up to represent the numbers `1` through `8`, initially standing in the following order:

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

    # We'll try to formalize how to get them in numerical order.
    # We start with a greedy approach (like from Problem Set 1!), solving the local problem that `4` and `2` are out of order:

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

    # Continuing by going down and comparing numbers one pair at a time, swapping them as necessary:

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

    # We got to the end of the list, but it's not quite sorted. It is closer than before, though, because on each swap, a smaller number is moved (or "bubbles") to the left, while a larger number "bubbles" to the right.

Let's try running through again:

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

We might need to do this a few more times:

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

And again:

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

Because we made a swap on this last pass, let's run through and do a sanity check that the numbers are in fact all in order:

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

We made no changes on the last pass, so we can be certain that we're done and the list really is sorted!

- This algorithm is called **bubble sort**, in which we sort pairwise until we can make an entire pass through the array without making any swaps.
- Let's try another algorithm, in which we go through the array and pick out the element we want (starting with the smallest):

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

We don't find anything smaller than `1`, so we move `1` to the front:

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

We swap the existing value in the first position with the position where we found the smallest element.

We know that `1` is definitely in the right spot, so we can ignore `1` and only consider the rest of the list when looking for the smallest element

We continue building up our sorted list at the front, finding the smallest element and moving it into position:

```
1    2    6    8    4    3    7    5    // 2 is already in position
1    2    3    8    4    6    7    5    // 3 is moved toward the front
1    2    3    4    8    6    7    5    // 4 is moved toward the front
1    2    3    4    5    6    7    8    // 5 is moved toward the front
```

Running through the rest of the list, every time we look for the smallest element we find that it's already in position, so we don't need to move them - just add them to the part of the list we've already sorted.

This algorithm is called **selection sort**, because we repeatedly "select" the smallest element.

- Let's reset one more time:

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

If we restrict our list to just the first element, we have just the list `[4]`, which is trivially sorted (since it only contains one element).

Let's go down the list and move each element from the "unsorted" to the "sorted" portion of the list, making sure to place it correctly in the sorted portion:

```
4    2    6    8    1    3    7    5    // [4] is sorted
2    4    6    8    1    3    7    5    // we move 2 to in front of 4;
 [2,4] is sorted
2    4    6    8    1    3    7    5    // [2,4,6] is sorted
2    4    6    8    1    3    7    5    // [2,4,6,8] is sorted
1    2    4    6    8    3    7    5    // we move 1 to the beginning;
 [1,2,4,6,8] is sorted
1    2    3    4    6    8    7    5    // we move 3 to its location;
 [1,2,3,4,6,8] is sorted
1    2    3    4    6    7    8    5    // we move 7 to its location;
 [1,2,3,4,6,7,8] is sorted
1    2    3    4    5    6    7    8    // we move 5 to its location;
 the whole list is sorted
```

    # When we move `1` to the beginning, `2`, `4`, `6`, and `8` all need to shift over (with similar effects when we move `3`, `7`, and `5` into position), so even though we pass through the list only once, it still takes quite a few steps.
  # This algorithm is called **insertion sort**, because we take the elements one at a time and insert them in their correct place in the sorted part of the array.

4. Algorithmic Efficiency

- Let's see how efficient these algorithms are, starting by looking at the simplest:
  # When we do **bubble sort**, each pass through the list takes *n* - 1 steps.
  # For **selection sort**, because we select one element on each pass through the list and remove it from the part of the list we need to consider, each successive pass takes one step fewer:

```
(n - 1) + (n - 2) + ... + 1
```

  # This is equal to $n(n - 1)/2 = (n^2 - n)/2 = n^2/2 - n/2$
- The largest term in this expression is the n^2 part - as *n* gets larger, this term will dominate.
  # If we plug `1,000,000` in for *n*, we get $1{,}000{,}000^2/2 - 1{,}000{,}000/2 = 500{,}000{,}000{,}000 - 500{,}000 = 499{,}999{,}500{,}000$
  # `499,999,500,000` is pretty much equal to `500,000,000,000`!
  # Therefore, we say that this formula is on the **order** of n^2, or **$O(n^2)$**.
- This **big-O** notation is used in computer science to denote an **upper bound** on how long an algorithm takes to run.
  # Some examples of common running times for the kinds of algorithms we've seen or will see soon in this course are $O(n^2)$, $O(n \log n)$, $O(n)$, $O(\log n)$, and $O(1)$.
- So **selection sort** is **$O(n^2)$** - and it turns out that **bubble sort** and **insertion sort** have the same worst-case sorting time.
  # What's the best case for insertion sort? If our list is already sorted, we only need to pass through the list once to check that everything is already sorted - so only *n* steps are required. But in the worst case, if the list is reversed, we have to slide everything over for every new element we add to the sorted list:

```
1 + 2 + ... + (n - 2) + (n - 1)
```

    # This is equivalent to our expression for the efficiency of selection sort above, so insertion sort is also O(n^2).

    # As mentioned before, each pass through the list for bubble sort takes n - 1 steps, and if the list is completely reversed, it takes n passes through the list to sort it, so bubble sort is also O(n^2).

- What's an algorithm we've seen that takes O(n) steps? Our initial, one-page-at-a-time algorithm for finding Mike Smith in the phonebook is O(n), or **linear time**. (Note that two, ten, or more pages at a time would still be O(n), even if faster in real time, because running time would still grow linearly with the number of pages in the phonebook!)

    # Taking attendance by counting everyone in the room individually is also O(n).

- Our divide-and-conquer method of finding Mike Smith in the phonebook, which we'll call **binary search**, is O(log n), or **logarithmic time**.

- Any algorithm with a fixed number of steps regardless of the length of input is O(1), or **constant time** (not necessarily just one step, but a constant number of steps that doesn't depend on n).

- Another metric besides big-O that we'll care about is Ω, or **big-omega**, which refers to the **lower bound** on the running time of an algorithm.

    # Selection sort always takes on the order of n^2 steps, even if the list is already sorted, so it's both O(n^2) and $\Omega($$n^2$).

    # Bubble sort and insertion sort both only need to pass through the array once if it starts out sorted, so although they're also O(n^2), they're both $\Omega($$n$).

- This online demo[2] shows various sorting algorithms, such as bubble sort, moving bars in order of length. The longer bars "bubble" to the right, and the shorter bars "bubble" to the left. (It may or may not work on your computer, unfortunately, as browser support for Java applets like this is now pretty rare.)

- Another demo[3] compares how long various algorithms take, and shows that although these O(n^2) sorting algorithms may feel pretty fast, they're much, much slower than

[2] http://cs.smith.edu/~thiebaut/java/sort/demo.html
[3] http://cglab.ca/~morin/misc/sortalg/

merge sort, an algorithm with a better big-O running time that we'll discuss in more detail later.

- Finally, we watch what different sorting algorithms sound like[4] to get a feel for them in another way.

[4] http://youtu.be/t8g-iYGHpEA