Week 3, continued

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Reminder

- CS50 Lunch is this Friday as usual, 1:15pm. RSVP at http://cs50.harvard.edu/rsvp.

Sorting

- At the end of the semester, final exams are usually given on so-called "blue books." Let's look at how we can solve problems and think about problems more carefully, with 26 such blue books.
  # Each blue book has a letter from A-Z written on each of them. We mixed them up randomly, and now a volunteer from the audience, Mary Beth, is going to sort them.
  # She formed several piles, with letters close together in the alphabet sorted in the same small pile, and once she was done those piles were merged into one final large pile.
- This demonstrates that there are many approaches and techniques that we can use and combine.

Bubble Sort

- Last time we demonstrated bubble sort by switching adjacent pairs and moving larger values toward the right. With 8 people, we needed to make 7 comparisons, and so with `n` people we made `n - 1` comparisons the first time we passed through the line of people:

```
(n - 1)
```

- Let's use the number of comparisons as a unit of measure. So in the first pass we made `n - 1` comparisons, but we know that the largest value is now furthest on the right, so we only have to make `n - 2` comparisons in the second pass:

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

- Following this logic, we eventually get to 1 final comparison, for a total number of:

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

- To simplify, we can use the back of our old high school math book to eventually figure out that this series is equivalent to:

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

- Let's multiply it out:

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

- But what happens if we have, say, 1,000,000 people?

```
n^2/2 - n /2
1,000,000^2/2 - 1,000,000/2
500,000,000,000 - 500,000
499,999,500,000
```

  - # So what does this mean? It means that if we were to sort 1,000,000 people, it would take 499,999,500,000 comparisons, and that as `n` gets larger and larger, the `n`^{2} becomes much larger than `n`.
    - # In fact, even a certain senator from Illinois can comment[1] on the inefficiencies of bubble sort.

Big # Notation

- To generalize more effectively, we look at the underlined terms below, and realize that n^2 is so much bigger than n that it dominates:

```
n²/2 - n /2
1,000,000²/2 - 1,000,000/2
500,000,000,000 - 500,000
499,999,500,000
```

- So we're going to take just the biggest term that matters the most:

```
n²/2 - n/2
O(n²)
```

- and say that bubble sort is on the **order**, **big** O, **of n^2**. We drop the smallest term, and even the division by 2, as that's still only a constant factor. We care about the part that gets bigger fastest as our input gets bigger.
- Big #; is the notation we'll use to compare the running time of algorithms, which counts generally the number of comparisons or swaps you're making, rather than the time or memory it takes. Indeed, big # is an upper bound on the running time of bubble sort; in other words, bubble sort will take no more than n^2 steps to complete.
- What if we think about an upper bound of #(n)? An algorithm like finding the biggest number in a list would take no more than n steps in finding that number. In the worst case, it might be the very last number, taking n steps, so n is the upper bound.
- What about #(log n)? The phone book problem required #(log n) time to solve, which really just means it's approximately decreasing the problem in half every step, such that the problem becomes increasingly small. Binary search, shown with the doors on the screen from last lecture, takes #(log n), but it assumes that the input is already sorted.

[1] http://youtu.be/FmJxGw0O5wA?t=9m37s

- #(1) means that the algorithm takes a constant time or number of steps, as opposed to a single step. Maybe it's 1 or 10 or 1000, but it's independent of the size of the problem or how big *n* is. Adding two numbers takes #(1), as does finding the first (or any!) element in an array. (As an aside, an array is a type of random-access memory, in which elements can be randomly accessed in constant time.)

    # Thinking back to week 0, the `[ say ]` block in Scratch also took constant time. (Actually, a better example of operations that take constant time would be Scratch's `[ show ]` or `[ hide ]` block, since the performance of Scratch's `[ say ]` block is arguably dependent on the length of the phrase that you have it say.)

Big Ω Notation

- The flip side is **big Omega**, Ω, notation, which is the lower bound running time for our algorithm.

- An algorithm that has a lower bound of *n* steps, $\Omega(n)$, is bubble sort, since we would only need *n* steps to verify that a list is already sorted. All sorting algorithms, in fact, should have $\Omega(n)$ running time. If we looked at fewer than *n* elements, then we wouldn't know if they're all sorted.

- What about an algorithm with $\Omega(1)$? `printf` is suggested by someone in the audience, but we have to be careful since it does take input of a certain length, and would take longer to print a longer string than a shorter one. Let's consider binary search. In the best case, we might open a phonebook and find Mike Smith in the middle, taking only one or two or a constant number of steps to find him on the first try. (Ajay from Monday also demonstrated this lower bound when he got lucky twice and found the number 50 on the first try both times.)

- As an aside, if # and Ω happen to be same for some algorithm, then we say it is in **Theta**, or Θ, of some value, as in $\Theta(n)$.

Analyzing Running Time

- Let's look at selection sort. We start by going through the list to find the smallest element, taking *n* - 1 steps by making comparisons between two elements at a time. The next pass will take *n* - 2 steps, and like we've seen in bubble sort, result in #(n^2) for selection sort. But since we've defined selection sort to keep going through the list taking one smallest element at at time until we have an entire sorted list, it also has $\Omega(n^2)$.

- In insertion sort we took each person and placed them in the correct place, and though we moved linearly down the list one at a time, inserting them required the other elements to move. So in the worst case there might be *n* - 1 shifts to the right, *n* - 2 shifts, etc, with #(n^2) for insertion sort, and also $\Omega(n^2)$ since we also continue one element at a time without any optimization. **[Actually, David misspoke. Insertion sort is actually in Ω(n). We'll fix the video asap.]**

- Bogosort is fun to look at, since it takes its input, like a deck of cards, randomly shuffles them, checks if it's sorted, and repeats until it is. (One of its nicknames[2] is actually stupid sort.)

Recursion and Merge Sort

- Let's have two volunteers from the audience, Peter and Alina, sort a deck of cards. We see them separating the cards by suits, with diamonds together, and hearts together, etc. Then they sort by number, and finally merge the suits together.
  # The ways they're sorting are algorithms, or processes, that we take for granted as humans and that we've long had intuition for. But once we formalize them, we can solve more interesting problems and more quickly.
  # This is also an example of dividing a problem into smaller ones with the same requirements, solving them simultaneously, and combining the results.

- Thinking back to the pseudocode from week 0:

[2] http://en.wikipedia.org/wiki/Bogosort

```
pick up phone book
open to middle of phone book
look at names
if "Smith" is among names
    call Mike
else if "Smith" is earlier in book
    open to middle of left half of book
    go to line 3
else if "Smith" is later in book
    open to middle of right half of book
    go to line 3
else
    give up
```

# Notice how we have an iterative, or looping approach with lines 8 and 11, where we go back to line 3 and repeat the process again and again.

- But let’s simplify this and change the nature of our program:

```
pick up phone book
open to middle of phone book
look at names
if "Smith" is among names
    call Mike
else if "Smith" is earlier in book
    search for Mike in left half of book ❶

else if "Smith" is later in book
    search for Mike in right half of book ❷

else
    give up
```

# Now we seem to have an incomplete instruction. How do we `search for Mike` in lines 7 and 10? Well, we just use this exact same program, starting at the top. Lines 4 and 12 makes sure that the program will end. With lines 7 and 10, we have a **recursive** algorithm that "calls" itself.

- A complicated concept at first, let’s look at another example with merge sort:

```
On input of n elements
```

```
if n < 2
    return
else
    sort left half of elements
    sort right half of elements
    merge sorted halves
```

# We have the check at the beginning, since a list with one or zero elements is already sorted. That's our **base case** (earlier with the phonebook it was either calling Mike or giving up) which ensures that the algorithm will eventually stop.

# We also have `sort left half` and `sort right half`, but the key step is `merge sorted halves` where we reassemble the sorted lists.

- Let's see this with 8 volunteers from the audience, numbered in the following order:

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

- So let's run through merge sort. There are 8 elements, so we mentally divide the list in two halves.

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

- Then we start over, and again proceed to the `else` block, which means we are sorting the left half of the left half:

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

- We call the algorithm again, but this time we have just one element, 4, which is sorted. We move on to 2, which is also sorted by itself:

```
4    2    6    8    1    3    7    5

4    2    6    8    1    3    7    5
```

- Now the left half is sorted and the right half is sorted, so we need to merge the two halves, taking whichever element comes first and placing it on the left:

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

4         6    8    1    3    7    5    // bring the 2 forward
```

```
2                                       // we need a temporary location
 for the new, merged list

          6    8    1    3    7    5    // and then the 4
2    4

2    4    6    8    1    3    7    5    // now the left half of the left
 half is sorted
```

# Note that in merge sort we need some form of "scratch paper", or place to put these numbers, every time we merge them, meaning we need another resource, space, in addition to time.

- The next step is to mentally rewind and remember that we need to sort the right half of the left half, since we just sorted the left half of the left half. So the same process takes place (though for cleanliness we've omitted the shuffling to and from a temporary location as above):

```
2    4    6    8    1    3    7    5    // 6 is sorted

2    4    6    8    1    3    7    5    // 8 is sorted

2    4    6    8    1    3    7    5    // the right half of the left half
 is sorted

2    4    6    8    1    3    7    5    // the entire left half of the
 list is now sorted
```

- We return to the right half and quickly repeat the same process (and again remember that we need to bring merged elements forward before putting them back in the list):

```
2    4    6    8    1    3    7    5

// sort the left half of the right half
2    4    6    8    1    3    7    5

// sort the left half of the left half of the right half
2    4    6    8    1    3    7    5

// sort the right half of the left half of the right half
2    4    6    8    1    3    7    5
```

```
// merge the left half of the right half
2    4    6    8    1    3    7    5

// sort the left half of the right half of the right half
2    4    6    8    1    3    7    5

// sort the right half of the right half of the right half
2    4    6    8    1    3    7    5

// merge the right half of the right half
2    4    6    8    1    3    5    7

// merge the right half
2    4    6    8    1    3    5    7
```

- Now that both halves are sorted, we start with the first element on each half, and take whichever comes first and places it in the next spot on our "scratchpad". We repeat this until the entire list is merged:

```
list:       2    4    6    8    1    3    5    7
scratch:

list:       2    4    6    8         3    5    7
scratch:    1

list:            4    6    8         3    5    7
scratch:    1    2

list:            4    6    8              5    7
scratch:    1    2    3

list:                 6    8              5    7
scratch:    1    2    3    4

list:                 6    8                   7
scratch:    1    2    3    4    5

list:                      8                   7
scratch:    1    2    3    4    5    6

list:                      8
scratch:    1    2    3    4    5    6    7
```

```
list:
scratch:    1     2     3     4     5     6     7     8
```

- Note that we are only moving down the list now, and never backwards. Though this feels slow, if we take a look at another sorting algorithm demo[3], without the human movement and explanation, we see how much faster merge sort is, especially given a larger data set.
- We divided our input in half log(n) times, but each time we did that, we had to merge n elements together. So merge sort has a total of #(n log n) running time.

[3] http://cg.scs.carleton.ca/~morin/misc/sortalg/