Computer Science 50
Fall 2008
Scribe Notes

Week 3 Wednesday: October 8, 2008
Andrew Sellergren, Anjuli Kannan

Contents

Computer Science 50
Fall 2008
Scribe Notes

Week 3 Wednesday: October 8, 2008
Andrew Sellergren, Anjuli Kannan

1 Announcements (0:00–13:00)

- Take a look at the Hacker Edition of Problem Set 2 even if you don't plan on doing it!
- David has bad handwriting.[1]
- "Lunch with David" on Fridays: e-mail `rsvp@cs50.net` if you'd like to join.
- Use the Bulletin Board! Your question might already be answered, or, if not, it *will* be answered by a member of the staff or perhaps even a fellow student.
- Slashdot: can computers think? Check out the article about a new attempt to pass the Turing test.

2 Searching and Sorting (13:00–50:00)

2.1 Bubble Sort

- Recall from last time that binary search enables us to vastly improve search time ($O(\log n)$). By this method, we could search a list of ∼4,000,000 students in about 32 steps, worst-case scenario.
- What's the catch? We have to sort the list beforehand. Some of you might have attempted this with the dictionary we provided for the Hacker Edition of Problem Set 2. How do we do this? One way is called bubble sort:

```
Repeat n times:
    For each element i:
        If element i and its neighbor are out of order:
            Swap them.
```

- So why isn't bubble sort $O(n)$? Remember, we're imagining the worst-case scenario. Because we can only move an element by one position each time we walk through the list, we have to repeat the entire process above n times in order to move an element from the very front of the list to the very end of the list (i.e. if the list were *perfectly* out of order). Thus, bubble sort is $O(n^2)$.

[1] And not nearly the note-taking abilities of, for example, me. Should I use more exclamation points, though?

2.2 Selection Sort

- What other options are there? Recall selection sort:

```
Let i := 0.
Repeat n times:
    Find smallest value, s, between i and list's end, inclusive.
    Swap s with value at location i.
    Let i := i + 1.
```

- So we yank the smallest value out of the list and we want to put it at the front of the list. Should we shift every element of the list by one position? This would defeat our purpose! Instead, we'll swap the element at the front of the list with the smallest value we just yanked.

- After all this, what do we get, though? If the smallest value in the list is at the very end (worst-case scenario), then we have to walk through all n elements of the list to grab it. Then we have to walk through $n - 1$ elements to grab the second smallest value. And so on, repeating n times. In the end, it's still $O(n^2)$.[2]

- To see these sorting algorithms in action (and compare their speed), check out this wobsite.[3]

- So what improvements can we make? Consider the case where the list is already sorted before we begin. As it's written now, we're still going to take n^2 steps to figure that out. What if we declare a variable to keep track of the number of swaps we make? That way, if we walk through the list and make 0 swaps, we know that the list is fully sorted and we can stop.

- How do we quantify this improvement? Previously, the worst-case scenario was n^2. But, in fact, so was the best-case scenario since we would still take n^2 steps to figure out that the list was already sorted. Now, we're dealing not with O, but with Ω. By implementing this swaps variable, we've gone from $\Omega(n^2)$ to $\Omega(n)$. That is, we'll still have to walk through the list at least once (n steps) to find out that it's already sorted.

[2] Orly? Yarly.
[3] You can also read about it on my blag.

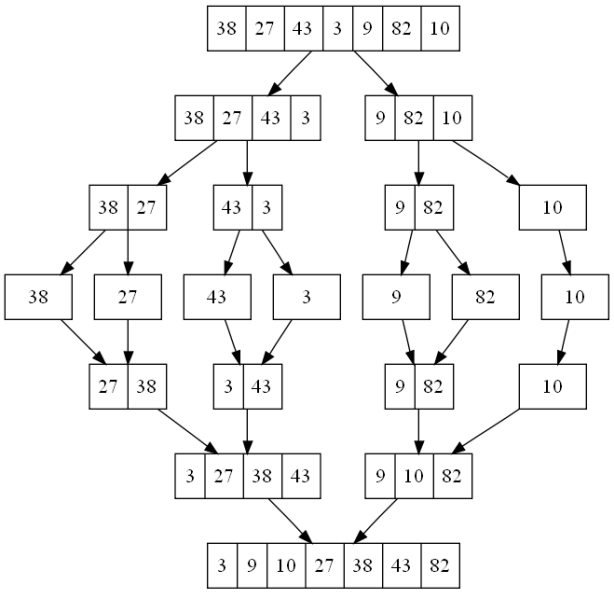
2.3 Merge Sort

- So are we stuck with bubble sort and selection sort?[4] Thankfully, no. Let's take a look at another algorithm called 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.
```

  Notice that as with binary search, we're attacking the problem by successively cutting it in half. Notice also that merge sort is a *recursive* algorithm.

- How does merge sort break out of its recursion? The first line of the pseudocode is the key. If $n < 2$, or in other words, if we only have one element left, then we've split the list in half as many times as possible and we're done sorting.



http://en.wikipedia.org/wiki/Merge_sort

[4] Yes. In fact, this is how Google sorts web pages. Once they're sorted, there's this guy (I know him personally) who goes through them by hand. Finally, your search results are returned to you by IPoAC.

- Take a look at this diagram from Wikipedia. At the top, we're beginning with an unsorted list of seven elements. Let's use merge sort to put them in order from least to greatest:
  1. Our input of n elements is the entire list; $n = 7 > 2$, so sort the left half of elements (i.e. split the list in half and move left in our tree diagram).
  2. Our input of n elements is the four numbers 38, 27, 43, and 3; $n = 4 > 2$, so sort the left half.
  3. Our input of n elements is the two numbers 38 and 27; $n = 2$, so sort the left half.
  4. Our input of n elements is now simply the number 38; $n < 2$, so return.
  5. Returning completed the step "Sort left half of elements," where the number 38 was the "left half of elements." Now, we move on to the "Sort right half of elements" step without recursing.
  6. Our input of n elements is now simply the number 27; $n < 2$, so return.
  7. Returning completed the step "Sort right half of elements," where the number 27 was the "right half of elements." Now, we move on to the "Merge sorted halves" step without recursing.
  8. In merging, we put 27 first. Note that we're going to need a second array of size n so that we have some place to store the numbers while we merge them. This is a space cost of merge sort!
  9. Finally, the numbers 27 and 38 are in order. This completes the step "Sort left half of elements," where the numbers 38 and 27 were the "left half of elements." Now, we move on to the "Sort right half of elements" step, where the numbers 43 and 3 are the "right half of elements."
  10. And so on...
- How do we merge sorted halves when each half has more than one element? Consider the lists {27, 38} and {3, 43} as our sorted halves:
  1. Point to the start of the left half with your left hand and to the start of the right half with your right hand.
  2. Compare the two numbers you're pointing to.
  3. Choose the lesser number and place it in your temporary array.
  4. Move the hand that was pointing to the lesser number one to the right.
  5. Return to step 2 and repeat until both hands are pointing to the ends of their respective lists.

- That's it! We repeat all these steps until we've sorted all the halves and merged all the sorted halves. Pretty simple, huh?[5]
- What's the big O notation for the merge sort algorithm? Well, we divided the list in half $\log n$ times. And to merge the sorted halves, we walked through n elemnts. Combine the two and we get $O(n \log n)$. Let's try to represent this formulaically:
  - Let $T(n)$ = running time if list size is n.
  - $T(n) = 0$ if $n < 2$
  - $T(n) = T(n/2) + T(n/2) + O(n)$ if $n > 1$
  - That is, we have to sort the left half, which takes $T(n/2)$, sort the right half, which takes $T(n/2)$, and merge, which takes $O(n)$!
- Ex: Suppose we want to find $T(16)$:
  - $T(16) = 2T(8) + 16$
  - $T(8) = 2T(4) + 8$
  - $T(4) = 2T(2) + 4$
  - $T(2) = 2T(1) + 2$
  - $T(1) = 0$
  - $T(16) = 2(2(2(2(0 + 2) + 4) + 8) + 16) = 64$
- If you want to see what this looks like in real terms, check out this wobsite. Try running bubble sort, selection sort, and merge sort side by side.

3 GDB (50:00–62:00)

- Recall that GDB is a very powerful debugging program.
- To use GDB, we must compile with the flag `-ggdb`, which tells GCC to add to our binary file certain markers which GDB will be able to recognize. This has a certain space cost which you can observe if you run `‘‘gcc’’` (un-aliased version of the command) to compile, followed by `ls -l` to view file sizes. The binary file will be slightly smaller than if you had run `make`, which includes the `-ggdb` flag by default.
- Let's return to `sigma1.c` and `sigma2.c`. Recall that we were able to crash `sigma2` by supplying very large numbers because it uses a recursive algorithm. After numerous function calls, we ran out of stack space and got a segmentation fault.
- Can we use GDB to debug `sigma1`. Run the command `gdb sigma1`. Once GDB starts up, if we type `run`, `sigma1` executes as normal, but within our GDB environment.

[5] No.

- If we type `break main` and then `run` when GDB starts up, `sigma1` will execute, but pause as soon as `main` is called. Now, we can step through each line of our code by typing `next`. We can also view the lines surrounding our current line by typing `list`.
- What if we look at the contents of variables before we've stored anything in them? Try typing `print answer` and notice that we get garbage. This is why you should initialize your variables!
- If we type `step`, we walk to the next line of code but also enter any functions that are called. If we `step` into the `sigma` function and `print m`, we get out the number we inputted as user.
- A more complete list of GDB commands is available here!