Computer Science 50
Fall 2010
Scribe Notes

Week 4 Monday: September 27, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 4 Monday: September 27, 2010
Andrew Sellergren

1 Announcements and Demos (0:00–8:00)

- This is CS50.
- 1 new handout.
- Now that you're officially in CS50, you should let everyone know it by wearing apparel from the CS50 Store!
- Check out this receipt for a real world example of floating-point imprecision.
- Version 3 of HarvardEvents is slicker than ever! If you've ever wanted to search or browse for events on campus, especially ones with free food, this is the tool for you. You can even add your own campus group's calendar by clicking "add calendar" in the top left corner underneath the logo.
- Even Barack Obama knows that bubble sort is suboptimal! Of course, there's no perfect way to sort 1 million 32-bit integers, as Eric Schmidt asked, because there are always tradeoffs between memory and runtime. In the real world, as David experienced while writing Perl scripts to detect worms across several gigabytes of data, a script that takes 10 to 15 minutes to write might take 8 hours to execute.
- Microsoft is hosting a NERD Party this Thursday!
- Asus is releasing a new tablet soon. Find out here how you can play with one before it goes public.

2 From Last Time (8:00–14:00)

- Selection sort, whereby the smallest number is found while traversing an array and is swapped to the beginning of the array, seemed like a reasonable alternative but bubble sort but turned out to be in $O(n^2)$ just as bubble sort was.
- Big O notation is used to characterize the worst-case running time of an algorithm. In the context of sorting, the worst case is when the array is in reverse order.
- Because selection sort takes no aggregate view of the entire array, its best-case running time is also n^2. Thus, we say that selection sort is in $\Omega(n^2)$.
- Thankfully, as this demo illustrates, merge sort presents a faster alternative to both selection sort and bubble sort.

3 Problem Set 3 (14:00–19:00)

- The first part of this problem set will have you implement one of the sorting algorithms we've looked at so far.
- The second part will have you finish building out The Game of Fifteen. Whereas for the previous problem sets, you started from scratch, for this problem set, you will start with distribution code, a skeleton framework of files and functions. Our goal is to get you accustomed to designing larger programs and working with someone else's code. HarvardEvents, for example, makes use of several JavaScript libraries, including YUI, which has a built-in calendar widget so that you don't have to implement it yourself.
- Before you begin coding, play around with the staff solution. The program takes a single command line argument, the dimensions of the board, and proceeds to print the game board with a `_` representing the blank tile. You are then continually prompted for a tile to move and you can play the game as you normally would.
- In the Hacker Edition, you are tasked with implementing the game as well as a solver mode. When asked for a tile to move, you should be able to enter `GOD` and the tiles will start moving automatically until the game is solved.

4 More Sorting (19:00–30:00)

- Last time, we sorted a list of numbers using a few different algorithms. This time, to emphasize that sorting boils down to a finite number of comparisons, we'll be sorting 8 cups by their weight using a standard balance scale.
- With a list of numbers, we as humans can observe the entire list at once and immediately identify the smallest number. The computer, however, cannot identify the smallest number without walking through the entire list once and making $n - 1$ comparisons. With a series of differently weighted cups, human and computer are evenly matched. Neither a human nor a computer can glance at the series of cups and know *a priori* which is the lightest. Thus, we must use the balance and compare 2 cups at a time, making $n - 1$ comparisons before we find the lightest cups.
- When we counted the number of students in Sanders and searched the phonebook for a specific name, we achieved a result most efficiently when we used a "divide and conquer" approach. So it is with sorting.
- Let's reduce our sorting problem to a more manageable size. We have 8 cups. If we divide them into two groups and focus on one group, we've immediately cut the problem in half. If we divide the cups twice more,

we're left with a single cup. When we consider this single cup by itself, it is already sorted, so we're done with the first "divide" step. In fact, the other cup that we singled out, when considered by itself, is also sorted. In order to create a list of size 2, however, we need to merge these two sorted cups. This is where the first comparison is made. We put the lighter cup on the left and the heavier cup on the right and we're done sorting a list of size 2.

- To backtrack a bit, we began with a list of size 8 and divided it into two lists of size 4. One of those lists of size 4, we divided into two lists of size 2. Finally, we divided one of those lists of size 2 and then merged it back together in sorted order. Now let's consider the other list of size 2.
- Repeating the process as before, we divide this list of size 2 into two separate single cups which, when considered individually, are sorted. When we merge these single cups back together, we make a single comparison to create a sorted list of size 2.
- We now have two sorted lists of size 2. To merge them, we'll need to walk through both, comparing cups along the way. Let's call the lists A and B for clarity's sake. The first cup in list A is lighter than the first cup in list B, so we place it farthest to the left. Then we compare the second cup in list A to the first cup in list B. The first cup in list B is lighter, so we place it second farthest from the left. Finally, the second cup in list A is lighter than the second cup in list B, so we place it third farthest from the left. The second cup in list B is the heaviest in this list of size 4, so we place it at the rightmost.
- Although this algorithm may seem confusing and slow, we'll soon find that it is much faster than either selection sort or bubble sort. We'll also see that it's actually quite simple and elegant to implement using recursion.

5 Merge Sort (30:00–52:00)

- Let's summarize in pseudocode the cup-sorting algorithm we used above:

```
On input of n elements:
  If n < 2
    Return.
  Else
    Sort left half of elements.
    Sort right half of elements.
    Merge sorted halves.
```

When we say "Sort left half," what we really mean is to start this entire algorithm over using a list of size $n/2$ rather than the original list of size n. You can think of all of the lines of pseudocode above as belonging to

a function called `Sort` which calls itself if it is passed 2 or more elements. The fact that this algorithm calls itself is what makes it recursive.

- The `n < 2` is the *base case* of this recursive algorithm. To prevent it from looping infinitely, we check if there are fewer than 2 elements in the list. If `n < 2`, then there is only a single item in the list and, as we said before, a single item considered by itself is already sorted.
- Consider the following array: 4, 2, 6, 8, 1, 3, 7, 5. Using merge sort, we're going to recursively chop it in half until we're left with two lists of size one: 4 and 2. Each of these lists is already "sorted," so now we need to merge them. How do we merge? Well, 2 is less than 4, so we want to put the 2 list before the 4 list. Where will we store these? In reality, we're going to need more memory to store the sorted total list.
- Now that 2 and 4 are in order, we step back for a second. 2 and 4 comprise the left half of a list of four: 4, 2, 6, 8. So now that we've sorted the left half of this list, we need to sort the right half, namely 6, 8. Again, we divide this list of length two into two lists of length one: 6 and 8. Then, we merge them, putting 6 before 8.
- One of the key takeaways to keep in mind with this algorithm is that we're only looking at each number once. Because it makes fewer comparisons than bubble sort or selection sort, merge sort is much faster.
- Now, again we're at the merging step. The left half is 2, 4 and the right half is 6, 8. Let's point our left hand at the first element of the left half and our right hand at the first element of the right half. $2 < 6$, so we put 2 in first. Now we'll advance our left hand one step so that it's pointing at 4. Our right hand is still pointing at 6 because we haven't merged that number yet. We compare 4 and 6 and then put 4 into our sorted array. Then we put 6 and 8 in because there's nothing to compare them against.
- At this point, we've completed the very step in our first call to merge sort: we've sorted the left half of the entire original list. Now we'll sort the right half: 1, 3, 7, 5.
- Cutting out a few of the intermediate steps, we end up with two sorted lists of size two: 1, 3 and 5, 7. When we merge them we end up with 1, 3, 5, 7. Finally, we'll merge this right half with the original left half, 2, 4, 6, 8. Obviously, we'll end up with 1, 2, 3, 4, 5, 6, 7, 8. We'll do so by iterating over both lists comparing the current left number to the current right number, selecting the smaller, and advancing to the next number in the list we just took a number from.
- Each merging step will take n steps because we must iterate over both lists of size $n/2$. But what about the division steps? We executed $\log n$ divisions, so we must execute $\log n$ merges. Our total merge sort algorithm, then, is in $O(n \log n)$. This is an improvement on $O(n^2)$!

- Let's try to represent merge sort's running time, $T(n)$, 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 $T(16)$:

  $T(16) = 2T(8) + 16$
  $T(8) = 2T(4) + 8$
  $T(4) = 2T(2) + 4$
  $T(2) = 2T(1) + 2$
  $T(1) = 0$
  $T(16) = 2(2(2(2(0 + 2) + 4) + 8) + 16) = 64$

  We add 16 in the first step because it takes 16 steps to merge both lists of 8. Eventually we boil down to $T(1)$, which is 0 because a list of size one is already sorted. Does our final result, 64, agree with our original determination of $O(n \log n)$. Well, $16 \times \log 16 = 64$, so yes. Compare this to $O(n^2)$, which would take $16^2 = 256$ steps. Already we're reaping the benefits.

- Have a listen to What different sorting algorithms sound like! They all sound like Pac-Man to me.