CS 50: Introduction to Computer Science I
Harvard College
Fall 2007

Scribe Notes
Week 3 Friday
Anjuli Kannan

Awards! (0:00-9:00)

- Cutest – Music video to "I Will Follow Him" – Jeremy Steineman '08
- Funniest – Dancing Malan – Vivek Sant '11
- Most Amazing –Scratch, Scratch Revolution – Ann Chi '08

Game of Fifteen (9:00-12:00)

- Everyone should have picked up a little plastic Game of 15 on the way in
- These have come all the way from Connecticut courtesy of David's mom!
- For those of you following along at home, this is what we're talking about:

1 2 3 4
5 6 7 8
9 10 12
13 14 11 15

- Give it to your neighbor to shuffle up the tiles
- Now try to put the tiles back into numerical order
- In a couple weeks, you'll be implementing the Game of 15 in C!

Sorting (12:00-26:30)

- How many steps will this take to sort?
  4 2 6 8 1 3 7 5
- First, let's think about how a human would sort it
- Bring down 8 volunteers to hold up pieces of paper with these numbers on them
- Have 1 more volunteer sort the 8 numbers by telling people where to move:
  - Move 8 to end: 4 2 6 1 3 7 5 8
  - Switch 1 and 4: 1 2 6 4 3 7 5 8
  - Switch 5 and 7: 1 2 6 4 3 5 7 8
  - Switch 3 and 6: 1 2 3 4 6 5 7 8
  - Switch 6 and 5: 1 2 3 4 6 5 7 8
- How did he do this? Switched low numbers with numbers that were higher than them but to the left of them.
- How many steps did it take? Seems like it only took 5 steps.
- Sweet! Does that mean we can sort n elements in runtime O(n)?
- No!!! If we were actually to code this up, we wouldn't have the convenience of human intelligence. Unfortunately, the computer is much more difficult to work with, and that adds time cost.

- For starters, we would not be able to take this kind of high level view to immediately see what is out of place. Just figuring out what is out of order will also take steps.
- As for switching numbers, we'd have to think about arrays and memory locations
- But what our sorting volunteer did in this example is actually a lot like a sorting algorithm called bubble sort…

Bubble Sort (18:45-26:30)

- In bubble sort, we repeatedly walk through the array and compare elements that are next to each other. If they are out of order, we swap them.
- Here's the pseudocode:
```
repeat n times
   for each element i:
      if element i and its neighbor are out of order
         Swap 'em.
```
- By this logic, we are guaranteed to get the largest element in the right place in the first round. In the second round we are guaranteed to get the next to largest element in the right place, and so on.
- Let's look at this example: 7 1 4 3 1 5 8 6
    - In the first round this would take the following transformation over the course of the for loop:
      7 1 4 3 1 5 8 6 → 1 7 4 3 1 5 8 6 → 1 4 7 3 1 5 8 6 → 1 4 3 7 1 5 8 6 → 1 4 3 1 7 5 8 6 → 1 4 3 1 5 7 8 6 → 1 4 3 1 5 7 8 6 → 1 4 3 1 5 7 6 8
    - Each time, we compare the ith element to the (i+1)th element and swap them if necessary
    - As you can see, the higher elements "bubble" up to the end
    - The end of the first round has put 8 in the correct place
    - The end of the next round will put 7 in the correct place
- At this rate, it will take us at most n rounds to put all n elements in the correct place.
- In the first round, we must look at n-1 elements (the rightmost has no neighbor). In the second round we can save time by looking at n-2 elements (because the last one is properly situated). In the third round we look at n-3 elements.
- So the total number of steps comes out as $(n-1) +(n-2) + (n-3) + … 1 = n(n-1)/2 = n^2/2 – n/2$
- Asymptotically, the only important term in this expression is $n^2/2$, and when we talk about running times we generally discard constant factors.
- So in the longrun, we say the runtime of bubble sort is order n square or $O(n^2)$
- For sorting, this is a pretty poor runtime. Can we do better?
- Idea #1: What if we just move 6 to the sixth slot and then try to move everyone else forward to take up the empty space?
    - Well, this might work here, but in general, our n items to be sorted won't be numbers 1 to n. That would just make life too easy.

    - And even still, moving everyone down a slot might seem like a pretty simple task for humans, but when we're working with a computer, that would translate to order n shifts!
- Idea # 2: Can we just look for small elements and put them at front?
    - Well we don't know a priori what the smallest value is, so how do we know what "small elements" are?
    - We'd have to start looking at the elements themselves, which is basically what we're doing in bubble sort
- Idea # 3: Maybe we could do a min search, using variables. The first iteration would look for the smallest element in the whole array and put it in the first slot. The second iteration would look for the min the in the rest of the array and put it in the second slot, and so on. Each time we find the min, we could just do a swap which is a constant number of operations.
    - This is pretty tight.
    - Actually, we call this selection sort…

Selection Sort (26:30-34:00)

- In selection sort, we first look for the minimum element over the entire array (n steps) and put it in the first slot
- Then we look for the minimum element over the rest of the array (n-1 steps) and put it in the second slot
- This process continues until we have filled the full array
- Here's the pseudocode:

```
let i = 0
repeat n times:
   find smallest value s between I andn list's end, inclusive
   swap s with value at location i
   i++
```

- The total number of steps comes out to n + (n-1) + (n-2) + … + 1 which is once again $O(n^2)$
- Can we do better?
- Well, binary search was O(log n), but we definitely can't do that well. Why not? Because log n steps isn't even enough to look at all the elements!
- But surely we can do better than n square…

Merge Sort (34:00-52:00)

- Let's use recursion!
- Here's the pseudocode:

```
On input of n elements:
      If n < 2, return.
```

```
Else
      Sort left half of elements.
      Sort right half of elements.
      Merge sorted halves.
```

- Okay, this kind of looks like a cop out because it assumes you can sort the right and left halves. But we can just call the merge sort function again, making this algorithm *recursive*.
- There's another tricky thing about this algorithm: how are we going to "merge" sorted halves?
- Well, if we have two sorted lists, we can combine them by repeatedly looking at the first element of each and taking the smaller one.
- Example: Suppose we run Mergesort on [4 2 6 8 1 3 7 5]
- Sort the left half 4 2 6 8 → run Mergesort([4 2 6 8])
    - Sort 4 2 → run Mergesort ([4 2])
        - Sort 4 → run Mergesort ([4])
            - Return [4] because size < 2
        - Sort 2 → run Mergesort ([2])
            - Return 2 because size < [2]
        - Merge halves
            - We have two lists: [4] and [2]. To merge, we look at the first elements of each and decide that 2 is smaller. So 2 is first. Then 4 is remaining so it comes next.
            - Return [2 4]
    - Similarly, run Mergesort(6 8) to get [6 8]
    - Merge [2 4] and [6 8]
        - We just keep comparing first elements.
        - Look at 2 and 6. 2 < 6, so 2 will be the first element in the new list, and we are left with [4] and [6 8] to merge
        - Now look at 4 and 6. 4 < 6, so 4 will be the second element in the new list and we are left with [6 8].
        - Since the first list is exhausted, we just take the entire second list [6 8] and append it to the end of our merged list. Return [2 4 6 8].
    - So our left half is sorted
- Now sort the right half.
    - Sort 1 3 → run Mergesort([1 3])
        - Sort 1 → run Mergesort([1]) → [1]
        - Sort 3 → run Mergesort([3]) → [3]
        - Merge [1] and [3] to get [1 3]
    - Sort 7 5
        - Sort 7 → run Mergesort([7]) → [7]
        - Sort 5 → run Mergesort([5]) → [5]
        - Merge [7] and [5] to get [5 7]

- Merge [1 3] and [5 7]
  - Smallest element between 1 and 5? 1, so 1 goes in the first slot
  - Remaining lists: [3] and [5 7]. Smallest element between 3 and 5? 3, so 3 goes in the next slot.
  - Now first list is exhausted so just append the remainder of second list
  - Return [1 3 5 7]
- Now merge [2 4 6 8] and [1 3 5 7]
  - Compare 1 and 2. Smallest is 1, so 1 goes in first slot.
  - Remaining lists are [2 4 6 8] and [3 5 7]. Compare 2 and 3 to find that 2 goes in next slot.
  - Remaining lists are [4 6 8] and [3 5 7]. Compare 3 and 4 to find that 3 goes in next slot.
  - Remaining lists are [4 6 8] and [5 7]. Continue with process.
  - Eventually get [1 2 3 4 5 6 7 8]
- What is the running time of this? O(n log n). Why?
  - We divide the list in half until we can divide no more. The number of times you can divide n half is equal to $\log_2 n$. But since we disregard constant factors, we just say log n.
  - On each division we do maximum n comparisons during the merge step (because we do one comparison for each slot of the array).
  - So we have log n divisions and n steps for each $\rightarrow$ O(n log n)
- More formally, for 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)!
- This is a recurrence!
- Example: Suppose we want to find T(16)
  - $T(16) = 2T(8) + 16$
  - $T(8) = 2T(4) + 8$
  - $T(4) =2T(2) + 4$
  - $T(2) = 2T(1) +2$
  - $T(1) = 0$
  - Putting this altogether: $T(16) = 2(2(2(2(0 +2)+4)+8)+16)$
- Sweet running time!
- What's the trade off?
  - Space – you need more RAM to do this because you need temporary storage to do the "scratchwork"
  - Complexity, clarity