Announcements

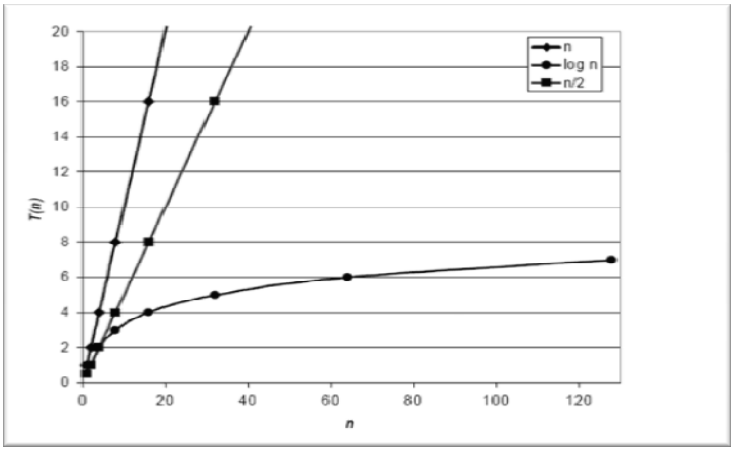
- Please return Scratch boards.
- We told you in the past to use cppreference.com. However, this site includes C++ reference, which may confuse you. To simplify things, please use the "C reference" site linked from the website.

Algorithm Run Times (13:30 – 20:00)

- Recall our algorithm for counting (or see week 0 Monday scribe notes)
- This algorithm is called logarithmic because, at each iteration, we reduce the size of the problem by a factor of one half
- Contrast this with the original counting algorithm.
- That algorithm would be linear because, at each iteration, we reduce the size of the problem by a difference of one.
- For small n, the improvement of a logarithmic algorithm over a linear one may not be obvious. Let's look at large n.

$\log_2 \log_2 n$	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
—	0	1	0	1	1	2
0	1	2	2	4	8	4
1	2	4	8	16	64	16
1.58	3	8	24	64	512	256
2	4	16	64	256	4096	65536
2.32	5	32	160	1024	32768	4294967296
2.6	6	64	384	4096	2.6×10^5	1.85×10^{19}
3	8	256	2.05×10^3	6.55×10^4	1.68×10^7	1.16×10^{77}
3.32	10	1024	1.02×10^4	1.05×10^6	1.07×10^9	1.8×10^{308}
4.32	20	1048576	2.1×10^7	1.1×10^{12}	1.15×10^{18}	6.7×10^{315652}

- As this chart shows, for n=1024, a linear algorithm will take 1024 steps, while a logarithmic one will take just 10.
- We can visualize the difference between run times by graphing T(n), runtime, against n, size of problem.



- As you can see, runtime rapidly increases for algorithms that are quadratic or, worse, in n, when compared to algorithms that are just linear or logarithmic in n.
- Worse still are cubic and exponential algorithms. See slides for additional comparisons.

Asymptotic Notation (20:00-24:30)

- Worst case runtime is indicated by O(.)
- For instance, counting students one by one (not using our logarithmic version) is O(n)
- For formal definitions, see slide 8.

Number-Hunting (24:30-33:00)

- On the board are two 8-member arrays of integers covered by pieces of paper.
- Bring down a volunteer and ask him to find the value 7 in the top array.
- Volunteer looks behind pieces of paper in order from left to right and finds 7 in the last one.
- What is the method? Linear search.
- In the worst case, this will take 8, or O(n) steps.
- Can we do better?
- Actually, we cannot do better as long as the numbers are randomly placed behind the papers. O(n) is the best we can achieve.
- But what if we are given a sorted array? Now look for 7 in the bottom array, which may be assumed sorted.
- This time, we can get O(log n)! We'll implement a similar algorithm to what we did with the phone book: look right in the middle and find 11, so throw away the right side of the array and repeat.

- This algorithm can also tell us in O(log n) whether or not the element is in the array.
- This is binary search, and is the same algorithm we used to search for a number in the phone book.

Search (33:00-35:00)

- Search algorithms will tell us if a particular value is in our array or not.
- Here is the pseudocode for what we call Linear Search:

```
on input n:
   for each element i:
      if i == n:
         return true.
   return false.
```

- Just look at every element and see if it's the right one. If you never find it, return false.
- What is it about this program that makes this O(n)? The loop, which repeats n times.
- Here is the pseudocode for Binary Search:

```
on input array[0], … , array[n-1] and i:
   let first = 0, last = n-1
   while first <= last:
      let middle = (first+last)/2
      if i < array[middle], then let last = middle - 1
      else if i < array[middle then let first = middle + 1
      else return true.
   return false.
```

- This is just a written out version of what we were doing with the phone book: look in the middle and go left or right.
- If we get to the point where first is no longer less than last, we can be sure that the desired value is not in the array

Recursion (35:00-64:00)

- Recursion is a technique for writing algorithms in which a function calls itself.
- Here's an example of a function that computes the sum of all the numbers from 1 to n (sigma.c):

```
int
sigma(int m)
{
    int i, sum = 0;
```

```
    /* avoid risk of infinite loop */
    if (m < 1)
        return 0;

    /* return sum of 1 through m */
    for (i = 1; i <= m; i++)
        sum += i;
    return sum;
}
```

- Notice the use of hierarchical decomposition in separating this functional component from main.
- In this function we just use iteration, i.e., a loop.
- The loop runs m times, each time adding i to the running sum and then incrementing i.
- Now let's introduce the idea of recursion. The basis of recursion is assuming that your function can deal with smaller inputs. So if we want to find sigma(n), we will just return (n+ sigma(n-1)).
- Then the computer will call sigma(n-1) and that will return (n-1 + sigma(n-2)), and the computer will call sigma(n-2) and so on.
- But after a certain point we don't want to subtract 1 anymore. So we add in a base case that stops the computer with the input gets so small that we know the answer.
- This occurs when n=0. At that point, we know that the sum of all the numbers from 0 to 0 is 0.
- Check out the code for sigma2.c:

```
int
sigma(int m)
{
    /* base case */
    if (m <= 0)
        return 0;

    /* recursive case */
    else
        return (m + sigma(m-1));
}
```

- This captures the spirit of taking a large problem and breaking off a piece we can handle, leaving a smaller problem to deal with.
- Is this faster or slower than the original version of the function?
- It takes the same number of "steps" but the steps in each version are different. We'd have to know details about the operating system, etc., to know how long it would take to process the different lines of code.

- Beware, however, of a drawback to recursive functions. Each call to the function adds a stack frame, and eventually there are too many to fit in the allotted space for the program. At this point, you get a segmentation fault.
- So, if it's not clearly faster, and takes up more space, what is better about it? Elegance.
- Also, recursive implementations can sometimes be faster and use less space, even if they don't in this case.
- As another example, we can write recursive code for binary search. See slide 15.

Introduction to Sorting (64:00-81:00)

- So binary search was a lot faster than linear search, but we had to make an assumption about the array: it was sorted.
- We can't always assume this will be true. Maybe we could sort the elements and then do our search. But how fast can we sort?
- Can we sort in constant time? No, because you have to touch every element at least once. This puts a lower bound on sorting: it must be at least linear.
- 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
- How did he do this? Repeatedly walked through the list, searching for the minimum remaining element each time, and then swapping them with the desired position.
- How long will this take in the worst case? In the first iteration he searches for 1, and, in the worst case it is at the end of the 8-number array, so that takes 8 steps. The next iteration will similarly take 7 steps in the worst case, and so on.
- This gives a run time of $8 + 7 + \ldots + 1 = 36$. More generally, $n + (n-1) + \ldots + 1 = n(n+1)/2 = (n^2 + n)/2$, which we will consider $O(n^2)$, or quadratic.
- Can we do better?
- One idea: at each pass, look for min and max. This reduces run time by factor of one half. However, $n/2 = O(n)$, so we have not done fundamentally better. (See formal definitions.)