

Announcements (0:00-3:00)

- Quiz 0 this Wednesday
 - Don't fret, it's just another way for us to assess how well you're absorbing material and for you to measure your own progress
 - We'll be spread out between three lecture halls based on last name. Check the website for details.
- This week's section notes and sections will be jam-packed with review material and sample questions!
- Monday is add/drop deadline, but if you're having trouble, don't drop without talking to David or one of the TFs.

Phone Number Lookups (3:00-8:00)

- Two volunteers get phone books and have to look up numbers.
- Volunteer #1 will execute linear search. Volunteer #2 will execute binary search (divide and conquer).
- Their task: find a plumber. Whoever finishes first wins. 1, 2, 3, go!
- Of course, volunteer #2 wins because binary search is sweet like that.
- In particular, binary search uses the strategy of repeatedly shrinking a problem into something more manageable.
- Let's put this into more concrete terms. Suppose the phone book had 1024 pages.
- How many steps would binary search take in the worst case? Just 10, because you can only divide 1024 in half 10 times.
- How many steps would linear search take? It could take up to 1024 because what you're looking up might be on the last page. Even in this case it would probably take at least 750 to get to the P's in the yellow pages.
- The moral of the story: the design of your algorithm influences how much time the developer takes to write a program and how much time the user takes to run it.
- Remember when you counted yourselves by repeatedly merging with another person and sitting down?
- Although that algorithm was different, it had the same spirit of repeatedly dividing the problem in half.
- Given n students, we can only divide them in half so many times. (In particular, you can only divide n in half $\log_2 n$ times). This puts a nice cap on the time it will take to run the algorithm.

Run Times (8:00-13:00)

- Let's talk about more deeply run times. What does it mean for an algorithm to be linear or logarithmic?
- We will use Excel to understand the meaning of these terms.

- n will represent the “size” of the problem (e.g., how many pages in the phone book, how many students in the class)
- $T(n)$ is the run time of the algorithm, how many steps it takes to complete as a function of the size of the input
- First, suppose that your algorithm is linear.
- A good example is counting students one at a time. If we count students one at a time, we will have to go through as many steps as there are students.
- Then we let $T(n) = n$.

- We have the following runtimes:

n	$T(n) = n$
1	1
20	20
40	40
60	60
80	80
100	100
120	120

- Now, suppose we wisen up and count by twos. Then we let $T(n) = n/2$
- We have the following runtimes:

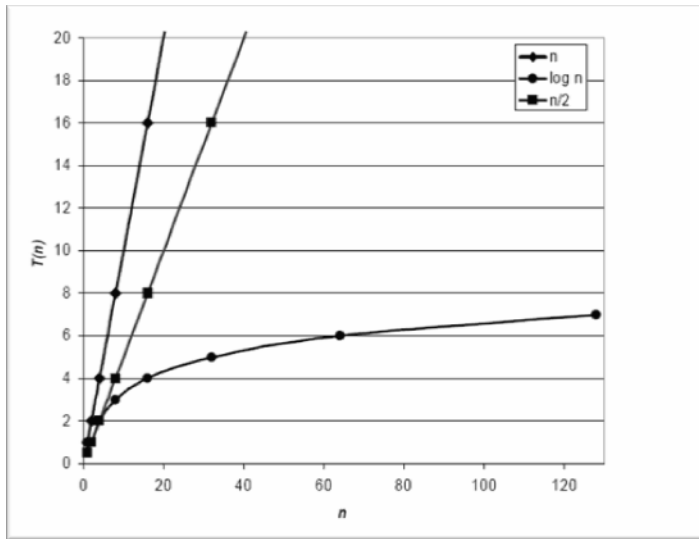
n	$T(n) = n/2$
1	.5
20	10
40	20
60	30
80	40
100	50
120	60

- Finally, suppose that we use parallel processing to have students count themselves using the algorithm from the first day (see scribe notes from Week 0 Monday for pseudocode).
- $T(n) = \log_2 n$ because you are repeatedly dividing the number of students in half, and you can only divide n by two $\log_2 n$ times.
- We have the following runtimes:

n	$T(n) = \log_2 n$
1	0
20	4.32
40	5.32
60	5.91
80	6.32
100	6.64
120	6.91

- Clearly, a *logarithmic* algorithm is much more efficient than a linear algorithm, especially as n grows.

- To make this even more clear, look at the following graph



- As you can see, $\log n$ is always less than n , but as n grows the difference becomes extremely large.
- This trend continues as n grows. The advantage of a logarithmic algorithm over a linear algorithm becomes even greater.
- Also, notice that improving by a constant factor—one half—doesn't really make much of a difference when we start talking about huge inputs. To a computer scientist, n and $n/2$ are more or less equally slow.
- Today, having algorithms that can handle large n is very important. Websites like Google and Facebook must serve large numbers of users and search large databases quickly enough to satisfy their users.
- To do this, they must design efficient algorithms. Finding places where, for instance, linear algorithms can be replaced with logarithmic algorithms that are just as effective is extremely important to the functioning of these websites.

A Bit about Grading (13:00-16:00)

- Your programs will henceforth be graded according to:
 - Correctness: Does it do what it is supposed to do?
 - Design: How well is your program written? Can I understand how it works if I try to read?
 - Style: Purely aesthetics. Are parentheses, indentation, comments, etc., used appropriately?

More on Run Times (16:00-24:00)

- The following chart compares runtimes of different categories of algorithms:

$\log_2 \log_2 n$	$\log_2 n$	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}

- The third column lists various values of n ranging from 2^0 to 2^{20} .
- The other column list the values of various functions of n . These functions are all commonly seen as runtimes of algorithms.
- For instance, look at the seventh row, where $n=64$.
 - If we had a linear algorithm to count 64 students, it would take 64 steps to complete, as shown by the third column
 - But if we had a logarithmic algorithm to count the same number of students, it would take just 6 steps to complete, as shown by the third column
 - Logarithmic runtime, as you can see, is a great improvement over linear runtime
- As we will see, sometimes we will have algorithms that cannot be written in such a way as to yield a linear runtime.
- In these cases, we may end up with quadratic runtimes, or $T(n) = n^2$.
- Although it is often not possible to reduce the runtime of such algorithms to linear, we will see how we can sometimes rewrite them to have a runtime of just $n \log n$.
- This doesn't sound like a big improvement, but compare the fourth and fifth columns for an input of 2^{20} . The difference is 10^5 !
- To indicate worst-case runtimes, we use so-called "big O notation"
- A linear algorithm is $O(n)$. A logarithmic algorithm is $O(\log_2 n)$.
- Whatever is in the parentheses is a function of n indicating an upper bound on the runtime of the algorithm.
- In the worst case, for instance, the entry we are looking for in the phone book will be on the last page and we will have to flip through all n pages!
- To indicate best-case runtime, we use theta notation
- Whatever is in the parentheses is a function of n that expresses a lower bound on the runtime of the algorithm.
- Consider linear search again. In the best case, the entry is on the first page. So linear search is $\Omega(1)$.
- So we can talk about algorithms in terms of their worst-case scenario or best-case scenario.

- Notice that we don't take into account the number of steps involved in turning each page. These details can vary from one computer to another (or one page-turner to another) so we don't take them to account. We just look at the bigger picture.

Number-Hunting (24:00-34:00)

- On the board are two arrays of integers covered by pieces of paper.
- Bring down a volunteer and ask her to find the value 17 in the top array.
- Merrit looks behind pieces of paper "randomly" and finds 17 on sixth try.
- What is the method? Random search with memory.
- In the worst case, this will take 8, or $O(n)$ steps.
- Can we do better?
- Suggestion: Look behind two pieces of paper at once.
- Yes, this would halve the running time, which in this case might make a pretty big difference. But in general, we don't really take constant factors into account when considering asymptotic runtimes. So $O(n/2)$ is considered to fall in the category $O(n)$.
- 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 17 in the bottom array, which we know is 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 23, 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.
- Both of these algorithms are search algorithms...

Search (34:00-38:00)

- 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 (38:00-48:00)

- 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;
}
```

- 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 $\text{sigma}(n)$, we will just return $(n + \text{sigma}(n-1))$.
- Then the computer will call $\text{sigma}(n-1)$ and that will return $(n-1 + \text{sigma}(n-2))$, and the computer will call $\text{sigma}(n-2)$ 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.