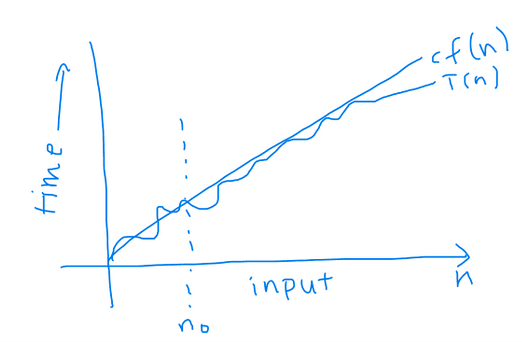
Algorithm efficiency and Big-oh

Jeopardy  [jeopardylabs.com/play/2019-01-22-76](https://jeopardylabs.com/play/2019-01-22-76).

s18 details: Got through the examples on page 4, but only had maybe 20 mins at that point. did not do linear search. did not show graph picture on p5. raced through 3 examples on p6.

* **Introduction/motivation** for the idea:
* In the last class we looked at our first ADT, the vector ADT.   
  + Remember that the main description of the vector ADT is that it maintains a collection of elements, and each element has a unique position, which is denoted by an integer.
  + Furthermore, the vector ADT is used whenever we want **random access** to the elements, because vectors support accessing any position in the vector at any time (that’s what random access means).
  + We also talked about the most natural **implementation** of the vector ADT, which would be using an array. However, we also said there’s no reason why you couldn’t use something like a linked list.
  + However, just because you could implement an ADT in a certain way doesn’t mean you should. Choosing an array-based vector implementation over a linked-list based implementation usually will work out better for reasons of efficiency. That is, the operations that the vector does --- in particular the random access ones --- will run more efficiently with an array-based implantation than with a linked-list implementation.
* So what do we mean by efficiency?  
  + We can measure the efficiency of an algorithm in two ways.
    - Space efficiency – the amount of memory it takes to run the algorithm.
    - Time efficiency – the amount of time it takes to run the algorithm.
  + In the early day of computing, space was more important, because memory was much slower than it was now, and it was much more expensive. Now, often we focus on time efficiency because storage is so cheap and plentiful these days. However, it is still a consideration, especially in certain cases where you often can’t physically put a lot of memory, such as in embedded systems (wireless sensors), space probes, digital cameras/phones.
  + Today we’re going to focus on time, though much of the discussion today will be applicable to space as well.
* In 141/142 (esp with me) we often talked about how we measure the running time of an algorithm, and we came to the conclusion that the best way of doing this was to count the number of “basic operations” that an algorithm does. The reason for this is it eliminates a lot of the problems of considering variations in programming language or type of computer.  
  + A “basic operation” is something the computer can do in one step.
* Consider the following algorithm
  + Computing Time Example
  + //find mean a0...an-1
  + //pseudocode
  + double mean(int a[], int n){
  + sum=0; (line 1)
  + i=0; (line 2)
  + while (i < n) { (line 3)
  + sum+=a[i]; (line 4)
  + i++; (line 5)
  + }
  + return sum/n; (line 6)
  + }
* If we change n, how many times does each line run?
  + line 1 – 1
  + 2 – 1
  + 3 – n+1
  + 4 – n
  + 5 – n
  + 6 – 1
  + total = 3n+4
* T(n) = 3n+4
* We can do this T(n) calculation for any algorithm, where n represents the “size of the input” to the algorithm.
* Now suppose we rewrite the last line of the algorithm as avg = sum/n. Return avg. Now it takes two steps, so the T(n) is now 3n+5.   
  + Because algorithms can often be written in various ways that are all just slight variations on a theme, there are two things we usually do as computer scientists to make our lives easier.
  + 1. We usually consider the behavior of this function T(n) as n gets “really big.” In other words, when n = 1, the difference between 3n+4 and 3n+5 is 7 vs 8, which is like a 12.5% difference.
  + But when n = 100000, the difference is negligible.
  + 2. Because certain kinds of differences become negligible when n becomes big, we often group these functions into categories. Where within a category, the differences between the T(n)’s are negligible, but between categories, the T(n)s differences are very big.
* We do this through a process called BIG-OH NOTATION.
  + You may have seen big oh before, but today we’re going to tackle it in a 100% formal manner.
  + Present big-oh definition.
  + T(n) is O(f(n)) if and only if  
      
    there exists some constant C such that T(n) <= C\*f(n) for all sufficiently large values of n.
  + Even more formally if

exists c, n0 forall n > n0 T(n) <= c\*f(n)

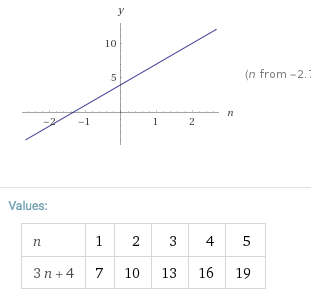
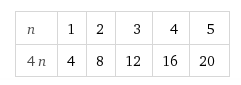
* draw picture
* 

Prove T(n)=3n+4 is O(n)

First graph 3n+4

USE WOLFRAM ALPHA:

graph 3n+4, n=0 to 10

* 
* graph 3n+4, n, n=0 to 10
  + illustrates that n is not above 3n+4. But we can multiply by a constant.
* Graph 3n+4, 4\*n, n=0 to 10
* 
* So we can choose c = 4, n0 = 4 (or 5, 6, ...)

More examples with 3n+4:

* Ask , is 3n+4 = O(2n). YES
* Is 3n+4 = O(n+17) YES
* Is 3n+4 = O(n^2) YES
* Is 3n+4 = O(sqrt(n)) NO

Rules of big-oh

* Since every T(n) has multiple big-oh functions that could be assigned to it, which one do we use?
* Remember big-oh specifies an upper bound on T(n). Remember if we say T(n)=O(f(n)), then that means if we go far enough to the right on the graph, c\*f(n) will always be ABOVE T(n).
* So what we usually try to do is make f(n) as simple & small as possible, so that c\*f(n) will be just barely above T(n).
* How does this work? Turns out we have rules.
* Start with T(n) = O(T(n)). This is always true. A function is always bounded above by itself.
* Rule #1 – We can drop coefficients inside O(...). This is because whatever goes inside big-oh is allowed to be multiplied by a coefficient c anyway. So instead of writing O(2n) or O(4n), we just write O(n). Simple!
* Rule #2 – If you have multiple terms added together inside big O(..), only keep the one that grows the fastest.

Why do we do this?

* Why not just use a stopwatch?
  + Computers are different. faster/slower
* Why not just report T(n)?
  + T(n) is counting "basic" steps, and the way we count basic steps differs from person to person and computer to computer.
  + Example: sum += a[i]; This might be one step, or it might be 2 (one for +=, one for the array access) or it might be 3 (+, =, array access)
* Why drop coefficients and only keep the term that grows the fastest?
  + We are interested in long-term behavior? How does this T(n) behave when n gets "really big"?
  + As we move to the right, the coefficients don't matter as much as the term that grows the fastest, which tends to dominate the rest of the terms.
  + ALSO: These big-oh rules end up grouping the functions into categories based on their fastest-growing term. For instance, 3n+4, 2n+7, and 16n+589 are all big-oh of O(n).

[draw table on slide], 3 rows, 4 columns(formula, O(1)?, O(n)? O(n^2))

Example: Is T(n)=57…

1. O(1)? Yes
2. O(n)? Yes
3. O(n2)? Yes

Example: Is T(n)=50n + 10000…

1. O(1)? No
2. O(n)? Yes
3. O(n2)? Yes

Example: Is T(n)=100005n2+2576n+4000

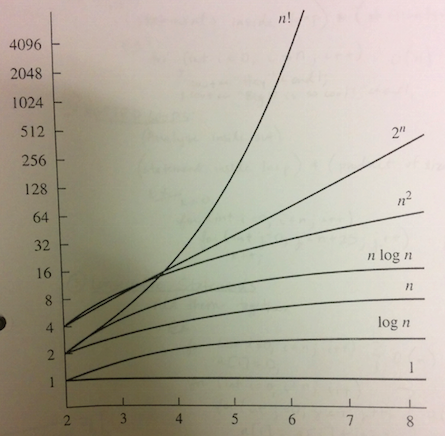
1. O(1)? No
2. O(n)? No
3. O(n2)? Yes

T(n)=100005n2+2576n+4000 <= 100005n2+2576n2+4000n2= 106581n2

c=106581, n0=1

**CATEGORIES**

* This naturally gives us a bunch of categories based on simple function growth patterns. O(1), n, n^2, n^3.... 2^n, 3^n, 4^n... n! (n^n beyond that), log n, n log n [Notice you can't drop the n from n log n, b/c not a constant]
* Name the ones that have names.



General Rules for Running t=Time **SHORTCUTS**

1. Loops:

(statements inside loop)\*(# iterations)

Ex: O(n)

for (int i=0; i<n; i++){

cout<<"Hey"<<endl;

cout<<"Big O is so cool!"<<endl;

}

1. Nested Loops: **[PRODUCT RULE]**

(Analyze inside out)

(statement inside loop)\*(product of size of all loops)

Ex: O(n2)

k=0;

for(int i=0; i<n; i++)

for (int j=0; j<n+25; j++)

k++;

1. Consecutive statements

Add them together

Ex: O(n2)

for (int i=0; i<n; i++)//O(n)

a[i]=0;

for (int i=0; i<n; i++)//O(n^2)

for (int j=0; j<n; j++)

a[i]=a[j]+i+j;

Logarithm (most confusing)

An algorithm is O(logn) if it takes constant time [O(1)] time to cut the problem by a fraction. (if it’s just c and not fraction time it’s O(n)).

(Just for completeness, don't go over this code.)

// C program to implement iterative Binary Search

#include <stdio.h>

// A iterative binary search function. It returns

// location of x in given array arr[l..r] if present,

// otherwise -1

int binarySearch(int arr[], int l, int r, int x)

{

    while (l <= r)

    {

        int m = l + (r - l)/2;

        // Check if x is present at mid

        if (arr[m] == x)

            return m;

        // If x greater, ignore left half

        if (arr[m] < x)

            l = m + 1;

        // If x is smaller, ignore right half

        else

            r = m - 1;

    }

    // if we reach here, then element was

    // not present

    return -1;

}

What is the tightest Big-O?

|  |  |
| --- | --- |
| sum=0;  for (int i=0; i<n; i++)  sum++; | O(n) |
| sum=0;  for (int i=0; i<n; i++)  for (int j=0; j<n; j++)  sum++; | O(n2) |
| sum=0;  for (int i=0; i<n; i++)  for (int j=0; j<n\*n; j++)  sum++; | O(n3) |
| sum=0;  for (int i=0; i<n; i++)  for (int j=0; j<i; j++)  sum++; | O(n2) |
| sum=0;  for (int i=0; i<n; i++)  for (int j=0; j<i\*i; j++)  for (int k=0; k<j+100; k++)  sum++; | O(n5) |
| x=n;  sum=0;  while(x>0){  sum++;  x=x/2;  } | O(logn) |
| sum=0;  for (int i=1; i<n; i\*=2)  sum++; | O(logn) |
| for (int i=0; i<n; i+=2){  sum++;  }  for (int j=0; j<n; j++){  sum++;  } | O(n) |

* **Worst case/avg case/best case.**

There's one important control statement in every computer language that we haven't seen yet in any of the code we've looked at with big oh.

IF statements!

But most algorithms usually include some if statements. So how do we analyze these?

The reason this is complicated is because if statements can change which lines of code execute. Which makes it harder to count the number of operations, because we may not know ahead of time if each if statement will be true or false.

* Consider algorithm for linear search. [SHOW SLIDE]
* // Linearly search x in arr[]. If x is present then return its
* // location, otherwise return -1
* int search(int arr[], int n, int x)
* {
* int i;
* for (i=0; i<n; i++)
* if (arr[i] == x)
* return i;
* return -1;
* }
* Analyze with T(n).
  + Assuming x is in arr[0], arr[n-1]. arr[n/2].
  + O(1), O(n), and O(n/2) = O(n).
* For a different algorithm, we might have a completely different set of scenarios that we care about. So it's difficult as algorithm designers to anticipate all the different cases in which people would use this. So normally what we do is we consider three different situations that cover the vast majority of what people care about.
* These are: the BEST case, the WORST case, and the AVERAGE case.
  + the best case asks – what is the fastest possible big-oh for this algorithm?
  + the worst case asks – what is the slowest possible big-oh for this algorithm?
  + avg – on "average" what is the big-oh?
* analyze linear search w/ these 3 ideas.

If time, talk about big oh of VECTOR operations.

If time, talk about why ignoring constants can get you into trouble. e.g., 2n versus 200000n. OR

2000000n versus 0.000001n^2.

**Day 2**

**Notes from s18 – too much to get through.**

Major goals:

* Review
* worst case/avg case/best case
* More examples
* recursive function T(n) calculation [use Fibonacci, binary search]
* Analyze vector operations in terms of big oh
  + make table.
  + talk about how we do this for all ADTs + implementations.

Review

* Last time we talked about how we analyze algorithms in terms of efficiency.
  + Time efficiency and space efficiency.
  + We talked about how we can take an algorithm, written in actual computer code or pseudocode, and come up with a function T(n), in terms of the size of the input to the algorithm, which is traditionally what n represents.
  + Then we talked about how express T(n) in terms of big-oh notation, which is a way to group these T(n) measurements into categories based on how fast T(n) grows when n gets “really big.”
  + Review def’n of big-oh:

**T(n) is O(f(n)) if and only if  
there exists some constant C such that T(n) <= C\*f(n) for all sufficiently large values of n.**   
  
Draw graph, showing T(n) and c\*f(n). T(n) must always be “below” the cf(n) function after some value n\_0.

**LIMIT DEF:**

**T(n) = O(f(n)) iff lim(n->infinity) |T(n)|/f(n) < infinity.**

* + In general, we care about the component of T(n) that grows the fastest, and we disregard any coefficients. For example, if your T(n) is a polynomial, we care about the term with the highest power. If you have an exponential component like 2^n, that will dominate any polynomial component. Any polynomial dominates any logarithm. Anything dominates a constant.
* In general, we end up with a number of common big-oh categories, namely
  + O(1) constant
  + O(n) linear
  + O(log n) – logarithmic
  + O(n log n) – log-linear.
  + O(n^2) – quadratic
  + O(2^n) – exponential.
* We “like” constant, linear, log, and log-linear. Algorithms that fall into those categories are generally tractable for most values of n that you will encounter, or at least their growth is slow enough to be manageable.
* Quadratic and higher algorithms are to be avoided if possible. Sometimes it’s not possible. There are plenty of really common algorithms where it can be proven you can’t operate faster than some of these higher categories. For instance, matrix multiplication. It’s relatively simple to prove that you can’t multiply matrices in faster than quadratic time. And the reason for that is when you multiply two matrices, you will necessarily have to have to look at every number within both matrices. So if each matrix is n-by-n, then that’s an O(n^2) algorithm at a bare minimum, before you even start multiplying any of the numbers together.

ANALYZING BIGOH FROM CODE – RULES

1. Product rule (loop rule)
   1. In general, with nested loops, when the # of loop iterations of the inner loop isn’t controlled by the outer loop, then if T1(n) = O(f(n)) and T2(n) = O(g(n)), then

you multiply the times and obtain T1(n)T2(n) = O(f(n)\*g(n))

1. Sum rule
   1. If you have one piece of code that runs in time T1(n), and another piece of code that follows it that runs in time T2(n), then the total time is T1+T2(n).   
      AND if T1(n) = O(f(n)) and T2(n) = O(g(n)), T1(n)+T2(n) = O(max(f(n), g(n))
2. Constant rule
   1. If you have a piece of code that runs in time T(n), and you run it k (fixed!) times, then that takes time k\*T(n). Suppose T(n) = O(f(n)). Then kT(n) = O(f(n)). AKA DROP CONSTANT MULTIPLIERS.

New Stuff For Today  
  
**Hidden coefficients.**

* Big-oh comes with a caveat, sometimes a large caveat. Recall that we had two rules for big oh—always take the term that grows the fastest, and drop coefficients. Sometimes this can lead to intentional consequences.
* Take for instance, T1(n) = n^2 + 2. A reasonable T(n). What is the big-oh? n^2.  
  And now, T2(n) = 1,000,000n + 1,000,000. big oh? O(n). So we have a quadratic algorithm in T1, and a linear-time algorithm in T2. By our rule of thumb, we should prefer T2 over T1.
* But for all n>= 0, clearly T1(n) < T2(n). so in the real world, we’d probably prefer T1.
* Now these numbers are made up, but here are some that are not.
* Let’s go back to the matrix multiplication problem.
  + Explain how the Naïve matrix mult algorithm is O(n^3).
    - New matrix is n by n. go through each square (n^2 operations).
    - For each square, you need to mult & add an entire row and column.
  + So one of the big breakthroughs in matrix multiplication was called Strassen’s algorithm (1969), which works in O(n^2.81). Only faster for matrices greater than 100 by 100.
  + Coopersmith-Winograd alg (1990) O(n^2.376). Coefficients are so big only worthwhile for matrices that are so big they can’t be stored on modern computers anyway.

**Best Case/Avg Case/Worst Case**

* So in general, the sequence of steps for getting the big-oh of an algorithm is calculate T(n) and get the big-oh of that.
* One of the issues we encountered last time was what do we do about IF statements. What if there’s an if statement in the code that gives you one version of T(n) sometimes and a different version of T(n) a different time?

Let’s examine the append algorithm.   
**void** IntVector::append(**int** value)  
{  
 **if** (thesize == capacity) 🡨 1 operation  
 expand(); 🡨 ???  
  
 data[thesize] = value; <- 1 operation  
 thesize++; <- 1 operation  
}

* But what is expand?
* **void** IntVector::expand()  
  {  
   **int** newcapacity = capacity + SIZE\_INCREMENT; 1 op  
   **int** \*newdata = **new int**[newcapacity]; 1 op  
    
   *// copy over* **for** (**int** x = 0; x < thesize; x++) x=0 🡪 1 op  
   { x<thesize -> n+1 ops  
   newdata[x] = data[x]; inside loop = n ops  
   }  
    
   **delete**[] data; 3 things here.  
   data = newdata;  
   capacity = newcapacity;  
  }
* Total = 1 + 1 + 1 + (n+1) + (n) + 3 = 2n + 7.  
  We said expand was O(n). So append is either T(n) = 3 OR T(n) = 2n + 7.
* These are two different big-ohs.
  + We are usually interested in 3 cases.
  + Best case O(1).
  + Worst case O(n). [USUAL ONE WE CARE ABOUT WHEN UNSPECIFIED]
  + Average case. how do we figure this out?
    - How often do we do these things?
    - We call expand() every 5 times.
    - So 4/5 times, it’s T(n) = 3.
    - 1/5 times, it’s T(n) = 2n+ 7.
    - So average T(n) = (4/5)3 + 1/5(2n+7). Still O(n).
* Review big-oh loop rules.

General Rules for Running t=Time

1. Loops:

(statements inside loop)\*(# iterations)

Ex: O(n)

for (int i=0; i<n; i++){

cout<<"Hey"<<endl;

cout<<"Big O is so cool!"<<endl;

}

1. Nested Loops:

(Analyze inside out)

(statement inside loop)\*(product of size of all loops)

Ex: O(n2)

k=0;

for(int i=0; i<n; i++)

for (int j=0; j<n+25; j++)

k++;

1. Consecutive statements

Add them together

Ex: O(n2)

for (int i=0; i<n; k++)//O(n)

a[i]=0;

for (int i=0; i<n; i++)//O(n^2)

for (int j=0; j<n; j++)

a[i]=a[j]+i+j;

(Continued run time rules)

1. If… else

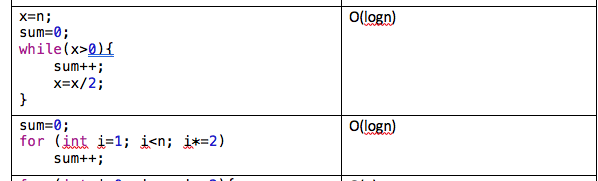
if (test condition)

S1

else:

S2

cost of test condition + larger of S1/S2



How do we analyze this first one?

Look at sequence of values of x.

time(0), First time x = n

time(1) next time x = n/2

time(2) n/4, (time3) n/8

At time step k, we have n/2^k.

When does the loop stop. When n/2^k = 1.

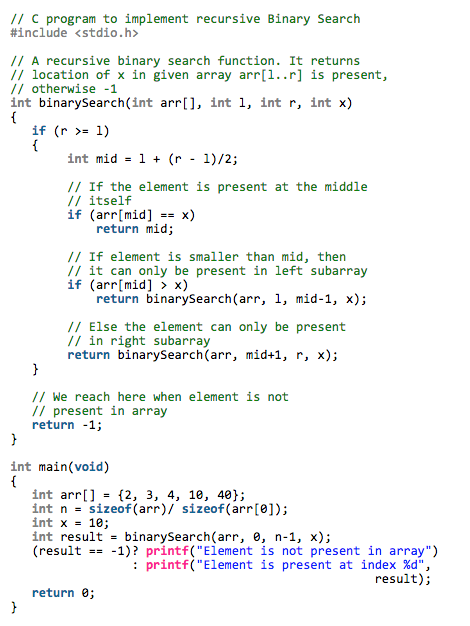
Solve this equation.

log\_2(x) = n. So this loop runs log2(n) times.

Next one.

Same thing. What’s the pattern. I = 1, 2, 4, 8, 16. So after k steps, I = 2^k.

Stops when I = n. So 2^k = n. Same thing.

log(2)(n) = k.  
  


Binary Search time: T(n) = T(n/2) + 1. T(1) = 1

Show analysis for this.

reduces to T(n) = k\*T(n/2^k)+k.

n/2^k = 1

logn = k

= T(1) + logn.

**Naïve Fibonacci:**

long long fib(int n)

{

if (n <= 1)

return 1;

else

return fib(n - 1) + fib(n - 2);

} Fake this analysis with T(n) = 2T(n-1) + 1. Real analysis is O((1+sqrt5)/2)^n = O(1.61)^n = golden ratio.

Better Fibonacci:

long long fib(int n)

{

vector<long long> cache(n);

cache[0] = 1;

cache[1] = 1;

for (int x = 2; x <= n; x++)

{

cache[x] = cache[x - 1] + cache[x - 2];

}

return cache[n];

}

Best Fibonacci:

long long fib(int n)

{

if (n <= 1)

return 1;

long long a = 1, b = 1, curr;

for (int x = 2; x <= n; x++)

{

curr = a + b;

a = b;

b = curr;

}

return curr;

}

Also analyze T(n) = 2T(n-1)+1, T(0)=1. [this is Fibonacci]

Table of Vector operations:

Worst case Best? Average

Get O(1) 1

Set O(1) 1

Append O(n) 1 n

Prepend O(n) n

[linear] Search O(n) 1 n/2 = O(n)

size O(1) 1

insert an element

front n 1

back n

middle n

remove an element

front n

back 1

We do this for all ADTs.