Recurrences from the very beginning:
A way of defining functions inductively. Define the small values for a base case. Then show what they do for n given n-1.

\[ T(0) = 1 \]
\[ T(1) = 2 \]
\[ T(n) = 2*T(n/2) + 5n + 3 \text{ for } n \geq 2 \]

So for some \( n \geq 3 \) you can define the function recursively by backtracking to your base case(s).

Defining MergeSort

```cpp
Mergesort(a) {
  if (|a| = 1) return a, it is already sorted
  if (|a| >= 2)
    Split into 2 equal size arrays
    Recursively sort the subarrays
    Merge the results
}
```

What is the time, \( T(n) \), to sort an array of length \( n \)?

- You do a test to see if \(|a| = 1\)
  \[ T(1) = c, \text{ some constant independent of } n \]

Otherwise, if \( n \geq 2 \)
Do another test, constant amount of time \( O(1) \)
Split into two arrays, which may be constant time it could also be linear time \( O(1) \)
Merging into one array, of length \( n \), so \( O(n) \) to merge

Everything is accounted for except for the recursive call:
\[ T(n) = c + cn + 2*T(n/2) \] -> this is the time needed for the recursive call
\[ T(n) = cn + 2*T(n/2), \text{ merge the constants they are unimportant-ish} \]
\( O(nlogn) \)
Typically if you are dividing something over and over again like this, you usually get \( O(nlogn) \)

if
\[ T(1) = 1, \text{ and } T(n) = 2*T(n-1), \text{ this will turn out to be exponential} \]

HOW TO SOLVE THESE RECURRENCE RELATIONS: Two ways
Take the mergesort example:
(a) Draw the recursion tree

```
  n
  / \
n/2  n/2
2cn/2  \  \  \  \n/2  n/4  n/4  n/4
```

At each level the order will be $cn$

You will eventually get to the depth $n/(2^k)$, and the depth of the recursion tree is going to be $\log(n)$
Which becomes $cn\log(n)$

(b) Unwind -> Dexter’s favorite: Try and unwind a few steps and look for a pattern

$T(n) = 3*T(n/3) + cn$ --> will also be $n\log(n)$, but a different constant

$T(n) = cn + 3*T(n/3)$
  = $cn + 3*($plug in $n/3$ for $n$ in the first equation, because it works for all large $n$)
  = $cn + 3*(c(n/3) + 3*T(n/9))$
  = $cn + 3*c(n/3) + 9*T(T(n/9))$
  = $cn + cn + 9*T(n/9)$
  = $2cn + 9*(c(n/9) + 3*T(n/27))$
  = $2cn + 9*c(n/9) + 27*T(n/27)$
  = $3cn + 27*T(n/27)$

after k steps, you will get:
  = $kcn + ((3^k)*T(n/3^k))$

after k= $\log_3(n)$ steps
  = $(\log_3n)cn + nc$ ---> $O(n\log(n))$

THIS IS NOT A PROOF! In order to prove this works formally, you MUST use INDUCTION. Once you have the answer, working out the inductive proof you will not have too much trouble solving the inductive proof.

The main thing I want you to get is a good intuition of what order an algorithm is, what are the characteristics of something that is linear, or quadratic, etc. time.

Good things to keep in mind:
Any time you have a polynomial: $O(3n^3 + 4n^2 - 18)$ ---> $O(n^3)$ {Bounded by}
The smaller order terms get absorbed for large values of $n$

Any power of a log: $(\log n)^{1000000}$ grows SLOWER than $n^{1/1000000}$

Any time that you have: $n^{1000000}$ grows SLOWER than $2^{n^{(1/1000000)}}$

$(\log n)^n$ vs $n^3$
remember that logs and exponents are monotonic functions: $x <= y$, $\log x <= \log y$, $2^x <= 2^y$
take the log of both sides: \( n \log \log n \) grows FASTER than \( 3 \log n \)

\[
O(\log_a n) = O(\log_b n)
\]

Because \( \log_a n = \log_b n / \log_b a \), and \( \log_b a \) is just a constant so it can be ignored

\[
\log(n^a) = a \log n
\]

\[
\log(xy) = \log x + \log y
\]

\[
\log(x/y) = \log x - \log y
\]

Also with logs you have to ignore small values because the log function is negative for small values \(<1\)

Note, \( \ln x \) means that the slope of the function at 1 is 1, and \( \ln(1) = 0 \)

Compare:

\[
n^{1/2} \log n \] compared to \( n \)

\[
\log n \] compared to \( n^{1/2} \) because you can cancel the \( \sqrt{n} \) because \( n = \sqrt{n} \sqrt{n} \)

\[
3^n \] compared to \( 2^n \) ---> you cannot just dismiss these as constants because they are not multiplicative constants, they are the base

Look at the ratio between them and you will see it grows unboundedly:

\[
3^n / 2^n = (3/2)^n
\]

TOPOLOGICAL SORT USING DEPTH FIRST SEARCH:

*see diagram in notes*

Remember, a Topological sort is a numbering/ordering of the verticies so that the edges of the graph go from smaller numbers to larger numbers.

Take any node with NO INCOMING EDGES. Then delete that node and delete those outgoing edges. Then you can take any one of the next nodes that has no incoming edges, and delete that node and all of its outgoing edges. And you can continue this pattern forever and ever. Remember, Topological sort is only possible with a graph that has no cycles. Is linear time with adjacency lists.

Using DEPTH FIRST SEARCH:

Do a DFS starting from a node with no incoming edges.

A e f g, then back out, c d and back out, b.

Build the tree and then traverse it in REVERSE PRE-ORDER (traverse from right to left)

Can also do Topological sort using DIJKSTRA’s Algorithm
*load factor
the ratio of the number of elements in the table to the size of the table, when it gets
too big then you would probably want to allocate a bigger table and rehash. You
MUST rehash before the table gets full in linear probing.
A good load factor is .75

*static vs dynamic types of objects
Expressions in the language HAVE static types.
Objects have dynamic types.
It does not make sense to talk about the dynamic type of some variable x. You are
really looking at the thing occupying x.

X + y - 3 is an expression, it has a static type of int.

Foo x= new Foo(); new Foo() is an expression, with a static type of Foo. The static
type of the variable x is Foo. The compiler sees all of this at Compile Time, it does
not need to run the program to figure this out.

When you run your program, during RunTime, x gets a dynamic type object of Foo.
The object that is created at run time has the dynamic type.

Suppose I do this:
Integer x= new Integer(5);
Object y= x;

New integer is an expression, static type Integer.
X is static type Integer in all places. Y is static type Object.

At Run Time, new Integer(5) creates an object and the object integer man runs on
over to jump inside x, and x has a dynamic type of Integer. Y has a static type of
Object, but a dynamic type of Integer.

Integer z= (Integer) y;
Although integer man is also in y, the compiler may not know that y has a dynamic
type of Integer, for example if y was the parameter of the method like foo(Object y)
{...}. At compile time, the JVM puts a little check in and at Run Time JVM goes
through and makes sure that y is indeed a dynamic type of Integer. This CANNOT be
done at Compile Time.

If it was Object x= new Object(); foo(x)
Foo(Object y) {
    Integer z= (Integer) y;
}
At Run Time this will produce a ClassCastException.
*KEEP IN MIND:
The dynamic types of objects NEVER changes, the cast does not change the dynamic
type of any object.
The static type of an expression NEVER changes. This always remains the same.

The invariant between static types of expressions and dynamic types of objects is:
Whenever you evaluate an expression, at any point, and you get a value, the dynamic
type of that value(object) is a subtype of the static type of that expression.

--- Abstract classes and interfaces can be casted to as well

Some casting errors can be caught at compile time:
Integer x= new Integer();
String s= (String) x;

Interface I1 {}
Interface I2 {}
Interface I3 extends I2 {}
Class C1 implements I1 {}
Class C2 implements I2 {}
Class C3 extends C1 implements I2 {}

(a) I1 i1= new C1();
(b) I1 i1= new C3();
(c) C3 c3= new I2();
(d) I3 i3= new C2();
(e) I2 i2= new C3();
    C1 c1= (C1)i2;

(a) ok because it is an up-cast
(b) ok because it is an upcast
(c) NO GOOD...you cannot say new INTERFACE, instantiation of interfaces fails. This
would be a compile time error.
(d) NOPE. This will also be a compile time error because the compiler knows that
C2 is not a subtype of I3.
(e) The first line is completely ok because it is a cast up. Now you have a dynamic
type of a C3 and a static type of an I2 variable trying to cast it into a, but since the
dynamic type is a subtype of the class we want to cast it too.

DFS and BFS when represented with adjacency lists:
Linear time: O(m+n), m is #edges and n #nodes
Matrix will always be more
For most graphs, m is typically on the order of n, and n^2 is the worst case for the
adjacency list and THE case for the matrix.
TABLE DOUBLING:
When the load factor gets too big, typically around 75% (this is what Java uses), you will have longer and longer chains, which means that searching is no longer $O(1)$. Now create a new table (e.g. if $n$ is the size of the original table, do $2n + 1$), and rehash all of the values and hopefully when you do so there will be shorter chains.

KEY POINTS OF THREADS AND CONCURRENCY:

What is a thread?
A sequence of some entity that is executing instructions.

If you have a shared data structure that multiple threads are accessing, you need to lock the data structure and therefore block out all other threads, do what you need to do with the data structure, then unlock it.

Synchronized(queue) {...}

In the synchronized block, your thread is the only thread that has access that particular object. This block is known as a critical section.

Thread.yield() --> a way to be nice and let other Threads run either of the same priority or lower priority if no other same-priority Threads exist / need to do things.

Thread.wait() and Thread.notify():

Suppose the main thread starts the GUI, then a Frame object gets created and displayed making dialogs and menus and all that nonsense, and the user clicks the mouse before the initialization to finish

A Function $f$ is $O(g)$ if there exists an $n_0 > 0$ and there exists a $c > 0$ for all $n \geq n_0$, and $f(n) \leq c g(n)$

Doing a Witness Pair Problem:
Log($n + 42$) is $O(\log n)$ (hint try using 2 and 7)

$n_0 = 7$, $c = 2$
For all $n \geq 7$, log($n + 42$) $\leq 2\log n$
True, because now you are comparing $n+42$ and $n^2$, which for sufficiently large $n$ as well as the base case, is true.