The Master Theorem, Introduction to Graphs, and Review of Proofs by Induction

Another way to solve recurrence relations, and an introduction to graphs.

The Master Theorem

Recall: Recurrence relations are a way to model the runtime of a recursive piece of code and may be solved to produce a closed form. We studied one method for solving these recurrence relations, the method of backwards substitution.

The Master Theorem is another method for finding the closed form of a recurrence relation. It is easy to apply the Master Theorem but it may only be used if certain conditions are met. Luckily, this is indeed the case in the analysis of many algorithms.

The Master Theorem – Simplified

If your recurrence relation fits this form:

any number for number of recursive calls

chop the input multiplicatively by at least half

$$T(n) = a * T(n / b) + O(n^d)$$

polynomial additional work

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c > 0$ and $d \geq 0$,

on the quiz, algorithm given, who would be able to have the master theorem applied to it; you must remember the right shape it must have so that you'll know it's right ig

then the closed-form of T(n) has this form:

given $$T(n) = \begin{cases} O(n^d) & \text{when } a < b^d \\ O(n^d \log(n)) & \text{when } a = b^d \\ O(n^{\log_b(a)}) & \text{when } a > b^d \end{cases}$$

the polynomial

exactly equal

The Master Theorem: Example

To demonstrate, let's use the recurrence relation for MergeSort the we had derived earlier. Through backwards substitution, we found the runtime of MergeSort to be *O*(nlog(n)), so this should also be the result produced by the Master Theorem.

Here is the recurrence relation for MergeSort:

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2T(n/2) + n^1 & \text{when } n > 1 \end{cases}$$

Notice that the recurrence relation matches the form needed to use the Master Theorem. In this case: a = 2, b = 2, c = 1, d = 1.

The equation $a \overset{?}{=} b^d$ evaluates to $2 = 2^1$, so the closed form of this equation is:

$$n^d\log(n) = n^1\log(n) = n\log(n)\text{, the same as before!}$$

Practice Quiz: Recurrence relations and runtime of recursive algorithm

quiz like this next tuesday 2/22

```
function helper (array A)
    s = size of A; sum = 0;
    for i = 1 to sqrt(s)      // sqrt(s) = $s^{0.5}$
        sum = sum + A [i];
    return sum;
 end function helper
```

a. *Give a tight upper bound for the runtime of function **helper**.* s^(1/2)
b. *Write a recurrence relation that models the runtime for function **solver.***
c. *Give a tight upper bound for the runtime of function **solver** as a function of n, the size of its array parameter (if you use the Master theorem, provide the values of a, b, etc).*
d. *How does **solver** compare to a cubic algorithm?*

```
function solver ( array A )
    if the size of A is 5 or less
        return the largest value in A;
    else
        Let A1, A2, A3, A4, A5 and A6 be six
            contiguous "equal size" parts of A;

        // recursive calls
        x1 = solver (A1);
        x2 = solver (A3);      recursive
        x3 = solver (A5);

        x4 = helper (A);

        return x1 + x2 + x3 + x4;
    end if
end function solver
```

next week, take a picture and turn in a pdf for the quiz

(see slides 3 for how to come up with recursive)

how to study: change a, b, d in the form and solve and come upw ith an answer. (review giving upper bound)

b. solver is recursive, so T(n) = $\begin{cases} \text{1, for n<= 5} \\ \text{3T(n/6) + O(n^(1/2)), otherwise} \end{cases}$

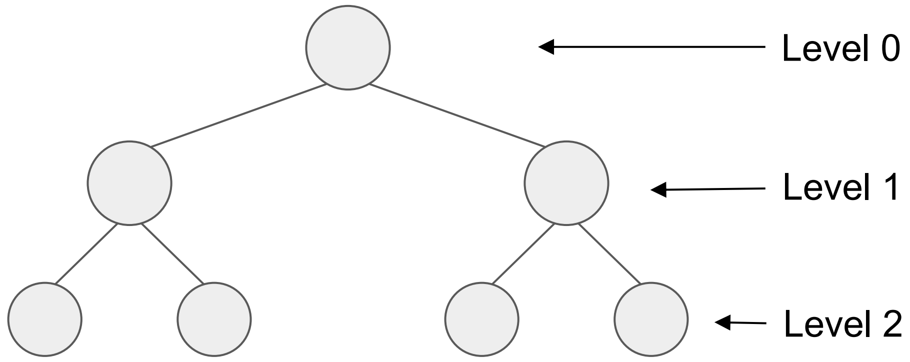
c. this form is the right shape for the master theorem. thus, by the master theorem with a = 3, b = 6, and d = 1/2, 3 > 6^(1/2), [sqrt(6) is going to be 2. something since 4 < 6 < 9] then T(n) = n^(log base 6 of 3).

d. n^(log base 6 of 3); 6 to the what is 3? something less than 1 (cuz 6^1 = 6 > 3), which < n < n^3. So, solver is more efficient than the cubic algorithm for large n.

it's important to say "for large n" since we're blowing off the smaller terms.

Review of Recursion

1. In a “completely full” tree, each level that is present is completely full. This is a diagram of a “completely full” binary tree with a height (number of edges from root to deepest leaf) of 2.



Compute the number of nodes at level l in a completely full binary tree.

Review of Recursion

```
// returns the number of nodes in level l of a completely full binary tree
getNumNodesInLevel(int l) { // l is a level, l >= 0
   if (l == 0)
      // handle base case from scratch: level 0 only has the root of the tree
      return 1
   else
      // every node in level (l - 1) has two children in level l
      return 2 * getNumNodesInLevel (l - 1)
}
```

we have faith in the algorithm will give the answer for l - 1

number of parents at level l - 1 and each of them have 2 children exactly; notice this comes from the problem definition

Introduction to Graphs

Graphs in computer science are used to model many problems, and are typically described as

a set of Vertices (or nodes), V,
connected by a set of Edges, E,
written as G = (V, E).

N = number of nodes

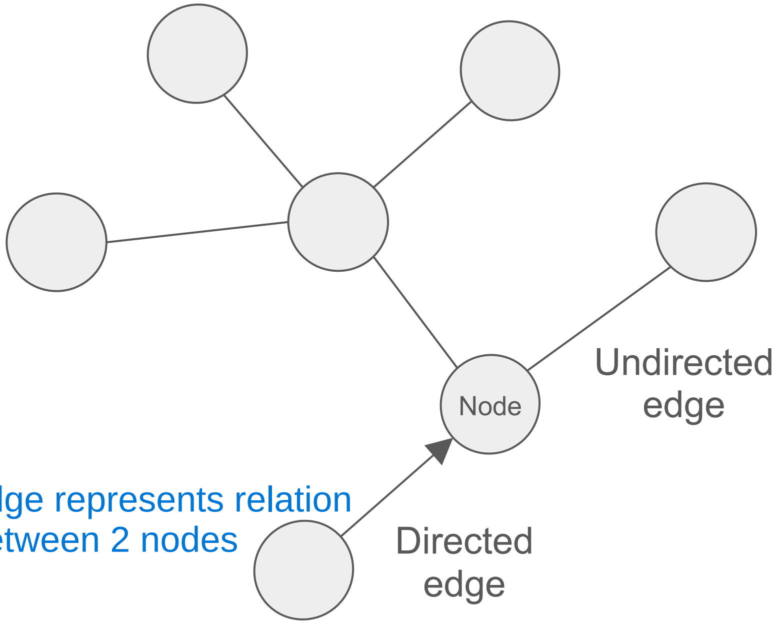
tree is an acyclic graph

Graphs allow cycles (loops), but trees don't.

Graphs may be **directed or undirected**, based on whether the graph's edges are directed or undirected. Directed edges may only be traversed one way, undirected edges may be traversed in either direction.

Weights (costs) by be associated with a graph's edges.

A graph is called "**complete**" or **fully-connected** if every vertex has an edge to every other vertex. A fully-connected undirected graph is called a **clique**.



edge represents relation between 2 nodes

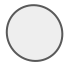
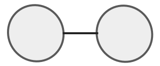
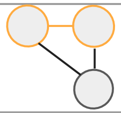
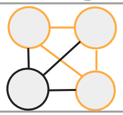
Used to model direct flights between airports or fiber optic cables between buildings.

Introduction to Graphs

We want to compute **the number of edges in a fully-connected undirected graph with n vertices**. Let's start by imagining a few small graphs to see if we can find a pattern.

We can build each subsequently larger graph by adding one more node and connecting this new node to all the older nodes. (yellow parts)

n nodes $\longrightarrow \displaystyle\sum_{i=0}^{n-1}$ (n - 1 + 1) * [(n - 1)/2] = [n*(n-1)]/2, hypothesized true for every n > 0

n	Fully-connected undirected graph	# edges
1	 0	0
2	 +1	1
3	 +2	3
4	 +3	6
5	? +4	? 10

More Review of Recursion

```
// returns the number of edges in fully-connected undirected graph with n nodes
getNumEdgesInFUG(int n) { // n is the number of nodes, n >= 1
  if (n == 1)
    // handle base case from scratch: single node graph has no edges
    return 0
  else
    // the n-th node must connect to all other (n-1) nodes, and
    // those (n-1) nodes must all be connected to each other
    return (n-1) + getNumEdgesInFUG (n - 1)
}
```

prove base case, prove the recursive case
up to n = 0, it is also true for the next value

Review of Proofs by Induction

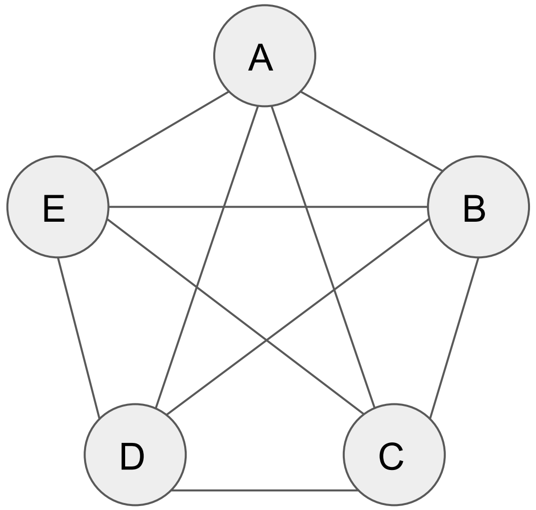
We want to compute **the number of edges in a fully-connected undirected graph with n vertices**. Lets start by looking at a few small graphs to see if we can find a pattern.

n	# edges
1	0
2	1
3	3
4	6
5	10

We do indeed find a pattern: It appears that the number of edges in a fully connected graph with n vertices is

$$\frac{n(n - 1)}{2}$$

A fully-connected, undirected graph with 5 vertices had10 edges.



Although this holds true for these specific cases, we would like to prove that it holds true for fully connected, undirected graphs of any size.

Introduction to Graphs

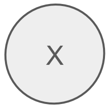
Claim: The number of edges in a fully connected, undirected graph with n vertices, for all n >= 1 is

trying to prove numEdgesinFUG(n) = $\frac{n(n-1)}{2}$, so LHS = 0, is RHS = 0, then LHS = RHS by transitivity of equality?

Proof: We want to prove the claim for all n >=1. We will do this by induction on n.

Step 1: Base case when n is 1.

Every complete, undirected graph with 1 vertex looks like this:



think of this as the method numEdgesinFUG(n)

By inspection, we observe that it has 0 edges. Per the claim, when n = 1, the number of edges is

this is the formula

$$\frac{n(n-1)}{2} = \frac{1(1-1)}{2} = \frac{1(0)}{2} = \frac{0}{2} =$$ 0, indeed as observed.

Introduction to Graphs

if we can call recursively for M - 1, you can tweak and make it true for M as well, and M + 1 respectively

Step 2: The inductive case.

We will begin by assuming that the claim is true for n up to and including some accepted maximum, lets call this maximum M. This means that, in the remaining part of our proof, we are allowed to use the "fact" (assumption) that

any fully-connected, undirected graph with M (or fewer) vertices indeed has (M * (M - 1)) / 2 edges.

We will try to prove the claim is also true when n is (M + 1). In other words, that

a fully-connected, undirected graph with (M + 1) vertices indeed has

$$\frac{\mathbf{(M + 1)((M + 1) - 1)}}{\mathbf{2}} \quad \textbf{edges.}$$

each node has m edges connected from it on an m + 1 nodes graph; if we take a subset of it and take one out, the subset is still FUG with m nodes

Introduction to Graphs

Lets start with any fully-connected undirected graph with (M + 1) vertices. From this starting graph, choose one vertex v, and mark v and all its edges.

The unmarked portions of the graph are a “sub-graph” with (M + 1 - 1) vertices.

Since the “sub-graph” has M vertices, we can apply the induction hypothesis to it, to obtain that it has (M * (M - 1)) / 2 edges. These edges are also in the starting graph.

Since the number of marked edges from v is M, the total number of edges in the starting graph with (M + 1) vertices is

using previous case of M based on our assumption in the beginning

$$\frac{M * (M - 1)}{2} + M$$

+ how many edges from the node taken away from the (M + 1) case (because the node cannot have an edge with itself, so it has M edges)

Introduction to Graphs

Now that we figured out the total number of edges for a graph with (k + 1) vertices, we have to show that it is precisely what our claim says it should be:

$$\frac{M(M - 1)}{2} + M = \frac{(M + 1)((M + 1) - 1)}{2} \Longleftrightarrow$$

$$\frac{M(M - 1)}{2} + \frac{2M}{2} = \frac{(M + 1)M}{2} \Longleftrightarrow \frac{M^2 - M + 2M}{2} = \frac{M^2 + M}{2} \Longleftrightarrow$$

$$\frac{M^2 + M}{2} = \frac{M^2 + M}{2}$$

So, we have proven the inductive step. By the principle of mathematical induction, the number of edges for a fully connected, undirected graph with n vertices is $\frac{n(n - 1)}{2}$ for all n >= 1.

Practice Quiz: Proof by induction

In a tree, the ***height of a node*** is the number of edges from the node to its deepest leaf. The ***height of a tree*** is a height of its root. A tree with single node (this node is the tree's root and also its only leaf) has a height of 0.

In a ***completely full binary tree***, every level that is present in the tree is completely full. For example,

- every completely full binary tree that has a level 0 has 1 node at level 0 (the root level); and
- every completely full binary tree that has a level 1 has 2 nodes at level 1 (the children of the root); and
- every completely full binary tree that has a level 2 has 4 nodes at level 2.

.

Practice Quiz: Proof by induction (continued)

Begin by drawing the shape of a completely full binary tree of height 4.

0. How many nodes does this tree have at level 0?

1. How many nodes does this tree have at level 1?

2. How many nodes does this tree have at level 2?

3. How many nodes does this tree have at level 3?

4. How many nodes does this tree have at level 4?

[pretend there's a drawing here]

Practice Quiz: Proof by induction (continued)

5. Your friend claims that *the number of nodes in level $\boldsymbol{l}$ in a completely full binary tree that has a level* $\boldsymbol{l}$ is 2^l, for $\boldsymbol{l}$ >= 0. Verify that this claim is indeed correct for small $\boldsymbol{l}$ by completing the following table.

$\boldsymbol{l}$	Number of nodes counted in the completely full binary tree at level $\boldsymbol{l}$?=	$\boldsymbol{2^l}$
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4

Practice Quiz: Proof by induction (continued)

6. Let's use ***numNodesLevel(l)*** to refer to the ***number of nodes in level l in a completely full binary tree that has a level l***. Prove by induction on l that ***numNodesLevel(l)*** = 2^l, for all l >= 0.

6.a. Base case: when l is __0__.

We will show that ___numNodesLevel(0)___ indeed __= 2^0__ .

A completely full binary tree at level 0 only has the root node, so numNodesLevel(0) = 1.
We note that the right hand side, 2^0, also equals 1.
So by transitivity of equality (RHS = 1 = LHS), the base case is true.

Practice Quiz: Proof by induction (continued)

6.b. Inductive case:

We begin by assuming that the claim is indeed true for all l <= M (we think of M as the accepted max).

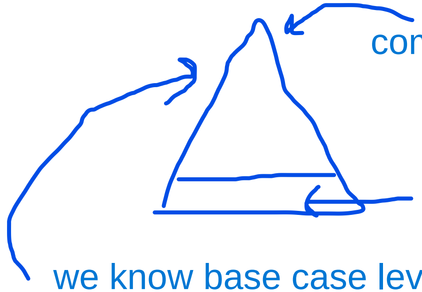
For the specific claim that we are proving here, this means that we accept that

(assume it holds for finite number l)

numNodesLevel(l) = 2^l ______ for all l <= M.

this is because we need to prove that the case holds for past the accepted max

Now we need to prove that ______ numNodesLevel(M + 1) $\overset{?}{=}$ 2^(M + 1) ______.



completely full binary tree

The nodes in level (M + 1) have 2^M parents in level M.Eac

we know base case level 0 has 2^0 nodes
level 1 has 2^1 nodes
level M has 2^M nodes

true for beginning of base case and anything beyond

Thus, since we proved the base case and the inductive case, ***numNodesLevel(l)*** = 2^l, for all l >= 0.

Additional practice for proofs by induction

These will make your brain buff.

1. The Postal Service at country Z has hired you to verify that every (integer) postage cost >= \$6 could be achieved using only \$2 and \$7 stamps. How do you confirm this?

2. Show that the number of nodes in a binary tree with height h is < 2^{h+1}.

sample inductive case from quiz accordance to practice quiz:

- assume claim true for all l <= Max - 1
- accept that numNodesLevel(l) = 2^l for all l <= Max - 1
- now we need to prove that numNodesLevel(Max) indeed = 2^Max

The nodes in level Max have 2^(Max - 1) parents in level Max - 1.
Each of those 2^(Max - 1) parents has exactly 2 children in level Max because the tree is binary and completely full.
So, the numNodesLevel(Max) = 2^(Max - 1) * 2 = 2^Max.