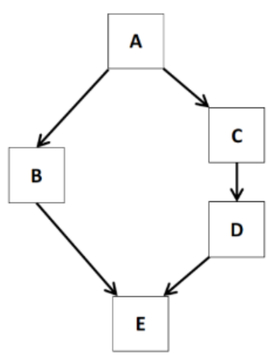
Asymptotic Bounds Formal Definition
o(g(n)) = { f(n) |c > 0, 0 > 0, n 0 : 0 f_{(n)} < cg(n) }
(()) = { () | > 0, 0 > 0, 0: 0 () () }
Θ(()) = { () | 1,2 > 0, 0 > 0, 0: 0 1()
() 2()
Ω(()) = { () | > 0, 0 > 0, 0 0 () ()



Simplified master theorem
T(n) = a * T(n/b) + (nd)
a>bd -> T(n)= (n ^ logb(a))
a=bd -> T(n)= (nd log(n))
a<bd -> T(n)= (nd)
Masters theorem
T(n) = a T(n/b) + f(n),
Constant > 0
f(n) = O(n ^ (logba -))
-> T(n)= (n ^ logba).
f(n) = (n ^ logba) ->
T(n)= ((n ^ logba) * lg n)
f(n) = Ω(n ^ (logba +))
if a f(n/b) c f(n)
for c < 1 -> T(n)= (f(n))

Logs
log_a(x*y) = log_a(x) + log_a(y)
log_a(x/y) = log_a(x) - log_a(y)
log_a(x^{y}) = y * log_a(x)
log_a(a^{r}) = r
log_a(a) = 1
log_a(1) = 0
log_a(1/b) = - log_a(b)
log_b(x) = θ(log_{a}(x))
log_{b}(x) = θ(log_{a}(x))
(a^(b^n))
(a^{b})^n
(x^y)^z = x^{(y*z)}
a^(log_a(b)) = b
a^(log_b(n)) = n^{(log_b(a))}
log_b(a) = log_{x}(a)/log_{x}(b)
log_b(n)=x -> b^{x} = n

Loop Invariant: At the start
Initialization: base case of loop
Maintenance: function of loop
Termination: state at end

Harmonic Numbers
Hn = 1/1 + ½ + ⅓ + ... 1/n
Hn ~ ln(n)
(Asymptotically Equivalent)

BinaryTree as Array
root i = 1
parent(i)= i/2
left(i) = 2i
right(i)= 2i+1

MAKE-SET(x)
[x] = x
rank[x] = 0

UNION(x,y)
LINK(FIND(x), FIND(y))

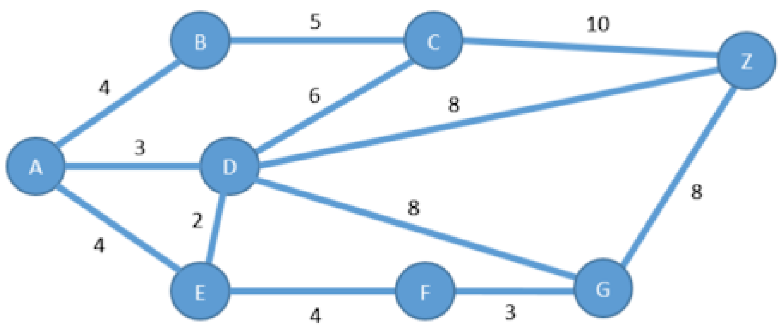
L'Hopital's Rule | Calc reminders
lim n f(n)/g(n) = lim n f′(n) / g′(n)
y=loga(x) if ay=x logb(x) = (loga(x))

True Statements About Asymptotic Limits
If lim n f(n) / g(n)
= 0, then f(n) = o(g(n)). (converse true)
= L, where 0 L < , then f(n) = O(g(n))
= L, where 0 < L < , then f(n) = (g(n))
= L, where 0 < L , then f(n) = Ω(g(n))
= , then f(n) = (g(n)). (converse true)
f(n) is asymptotically equivalent to g(n):
f(n) ~ g(n) only if lim nf(n)/g(n) =1

Union find
Union-Find data structure, also called a Disjoint-Set data structure Used for storing collections of sets Supports:
• makeSet(u): Create a set {u} O(n)
• find(u): Return the set that u is in O(1)
• union(u,v): Merge the set that u is in with the set that
Let R be a binary relation on a set S. Then R is an Equivalence Relation on S iff:
• R is Reflexive on S: R(x,x)
• R is Symmetric on S: If R(x,y) then R(y,x)
• R is Transitive on S: If R(x,y) and R(y,z) then R(x,z)
Example: Integers mod m Two integers are equivalent if they have the same remainder mod m. IF R is an Equivalence Relation on S, you can partition S into equivalence Classes induced by R, where all elements x and y in each Equivalence Class satisfy R(x,y).
Let S = { a, b, c, d, e}
Let R = { (a,a), (b,b), (c,c), (d,d), (e,e) (a,b), (b,a), (a,c), (c,a), (b,c), (c,b), (d,e), (e,d) }
This R can be partitioned into two subsets, such that within each subset, all the elements are related (to themselves and each other) in both directions.
P = { {a, b, c}, {d, e}} These two subsets of S are the Equivalence Classes induced by R.

Stirling's Approximation:
n! = sqrt(2πn) * (n/e)n
Stirling's Formula:
n!= sqrt(2πn) * (n/e)n * (1 + Θ(1/n))
Corollaries:
n! ~ sqrt(2πn) * (n/e)n
1) n! = o(nn)
2) n! = (bn) for any b>0
3) log(n!)= (n log(n))

FIND(x)
```
if x [x]
  [x] = FIND([x])
return [x]
```



Find:
Let's assume that we have an Equivalence Relation R, and a set of induced Equivalence Classes: {E1, E2 , E3 , ... , Ek}
• Each Ei will have a representative element.
• FIND(x) for x in S, returns the representative element for the Equivalence Class containing x
FIND -- an Example
Let S = { a, b, c, d, e}
P = { {a, b, c}, {d, e}}
E1 = {a, b, c}, E2 = {d, e}
Let a and d be the representative elements of E1 ,E2, respectively.
FIND(c) -> a
FIND(d) -> d
FIND(e) -> d

Insert MaxHeap
```
// Increase heapsize by 1, putting in new value "key", and
// maintaining the heap property for the resulting array A.
HeapInsert(A, heapsize, key)
  heapSize = heapSize + 1
  A[heapSize] = key
  j = heapSize
  // Propagate change further up tree if parent(j) value is smaller than A[j].//
  while j > 1 and A[ parent(j) ] < A[j]/.
    Swap A[j] with A[ parent(j) ]
    j = parent(j)
```

PARTITION (A, p, r)
```
//Quicksort and select
x = A[p]
q = p
for j = p + 1 to r
  if A[j] x
    q = q + 1
    exchange A[q] and A [j]
exchange A[p] and A[q]
return q
```

Recursion Tree Table:
for a * T(n/b) + dn,
Problem | size | work
Level t and bottom
at | n/bt | d*(at*n/bt)
alg(n) | n/blg(n) | (n^lg(a)

LINK(x, y)
```
if rank[x] > rank[y]
  [y] = x
else [x] = y
  if rank[x] = rank[y]
    rank[y] = rank[y] + 1
```

Answer 4: This is Case 3 of the SMT, where the bulk of the work is done at the root of the recursion tree, although you didn't have to say that.

To simplify notation, let lgbn = log$_b$n.

The work at level t of the tree is is O($n^d (a/b^d)^t$), so the total work in the tree is:
$\sum_{k=0}^{lgbn}$ O($n^d (a/b^d)^t$)

We'll prove that T(n) = O(n^d), and then prove that T(n) = Ω(n^d).

a) Proof that T(n) = O(n^d): Let r = (a/b^d). We're told that r < 1.

From the Recursion:
T(n) = $\sum_{t=0}^{lgbn} n^d (a/b^d)^t = n^d \sum_{t=0}^{lgbn} r^t$

< $n^d \sum_{k=0}^{\infty} r^t$ — Taking the summation further
= n^d * 1/(1-r) — Sum of infinite geometric progression with r < 1
= O(n^d) — Since r is a constant, so is 1/(1-r).

which is the first part of what we needed to prove.

b) Proof that T(n) = Ω(n^d): Let r = (a/b^d). We're told that r < 1.

From the Recursion:
T(n) = $\sum_{t=0}^{lgbn} n^d (a/b^d)^t = n^d \sum_{k=0}^{lgbn} r^t$

> n^d * 1 — Taking just the term with t = 0
= Ω(n^d)

which is the second part of what we needed to prove. **QED**

Strong Induction instructions:
Base Step: Prove that the proposition P(n0) is true.
Induction Step: Prove for all n > n0 that if P(k) is true for all k such that n0 k < n, then P(n) must also be true.
To do this, take an arbitrary n > n0, and assume that P(k) is true for all k such that n0 k < n then show that P(n) must also be true.
When the Base Step and the Induction Step have been proved, we conclude that P(n) must be true for all n n0
Weak Induction instructions
Base Step: Prove that the proposition P(n0) is true.
Induction Step: Prove for all n > n0 that if P(n-1) is true, then P(n) must also be true.
To do this, take an arbitrary n > n0, and assume that P(n-1) is true for that n then show that P(n) must also be true.
When the Base Step and the Induction Step have been proved, we conclude that P(n) must be true for all n n0

$$\frac{d}{dx} c = 0$$ Constant Rule
$$\frac{d}{dx} x^n = nx^{n-1}$$ Power Rule
$$\frac{d}{dx} \sin(x) = \cos(x)$$
$$\frac{d}{dx} \cos(x) = -\sin(x)$$ Trigonometric Rules
$$\frac{d}{dx} b^x = b^x \ln(b)$$ Exponential Rule
$$\frac{d}{dx} \ln(x) = \frac{1}{x}$$ Logarithmic Rule

QuickSort(A, p, r):
```
If p < r
  q = PARTITION(A, p, r)
  // Values in A[p..q] are less than or equal to A[q]
  // Values in A[q+1..r] are greater than A[q]
  QuickSort(A, p, q-1)
QuickSort(A, q+1 ,r)
To sort entire array A:
QuickSort(A, 1, A.length)
```

Fast Prims:
```
Until all vertices are reached:
  Activate the unreached vertex u with the smallest key.
  For each of u's neighbors v:
    k[v] = min( k[v], weight(u,v) )
    if k[v] updated, [v] = u
  Mark u as reached, and add ([u],u) to MS
```

Connected Components Example
V = {a, b, c, d, e}
E = { (a, b), (a,c), (d,e) }
Let V = {a, b, c, d, e} and E = { (a, b), (a,c), (d,e) }
• Start with an initial partition into singleton sets.
• P = { {a}, {b}, {c}, {d}, {e} }
• Process one edge at a time. (a, b)
• Perform FIND operations on each element, then UNION on the results: UNION(FIND(a), FIND(b))
• Continue until all the edges have been processed

Variation of strong induction
Base Step: Prove that the proposition P(n0) is true. Induction Step: Prove for all n >= n0 that if P(k) is true for all k such that n0 k n, then P(n+1) must also be true.
To do this, take an arbitrary n >= n0, and assume that P(k) is true for all k such that n0 k n then show that P(n+1) must also be true.
When the Base Step and the Induction Step have been proved, we conclude that P(n) must be true for all n n0
HeapSort Sorting Strategy: O(n log n)
1. Run BuildMaxHeap(A,n) on an unordered array A[1..n], and Set heapSize = n.
2. Find max element A[1].
3. Swap elements A[heapSize] and A[1]. Now the max element is at the end of the array!
4. Discard node heapSize from heap (by decrementing heapSize variable).
5. New root may violate MaxHeap Property, but its children must be MaxHeaps! So run MaxHeapify(A, heapSize, 1) to fix this.
6. Go back to Step 2 if heapSize is not 0.

Lower Bound of Ω(n log(n))
Any deterministic Comparison-Based Sorting Algorithm must take Ω(n log(n)) steps (Worst-Case Running Time).
Any randomized Comparison-Based Sorting Algorithm must take Ω(n log(n)) steps in Expectation (Average Running Time).

MaxHeapify(A, heapSize, parent) pseudocode
```
left_child = left(parent)
right_child = right(parent)
largest = parent
if (left_child <= heapSize and A[left_child] > A[largest])
largest = left_child
if (right_child <= heapSize and A[right_child] > A[largest])
largest = right_child
if largest parent
Swap A[parent] and A[largest]
MaxHeapify(A, heapSize, largest)
```

slowPrim(G = (V,E), starting vertex s):
```
//Let (s,u) be the lightest edge coming out of s.
  MST = { (s,u) }
  verticesVisited = { s, u }
  while |verticesVisited| < |V|:
    Find the lightest edge (x,v) in E such that:
      x is in verticesVisited, and
      v is not in verticesVisited
    Add (x,v) to MST
    Add v to verticesVisited
  return MST
```

DynamicBinomialCoefficient(,)
```
[0] = 1
for = 1 to
  [] = 0
  for = 1 to
    for = down to 1
      [] = [ - 1] + []
return []
```

Worst-Case Time Using "Basic" Up-Trees
• FIND(x): O(path length from x to root of x's tree) = O(n)
• UNION(x,y): O(1)

SELECT(A, k, p, r)
```
If r-p+1 <= n0 // MergeSort for "small" n
  MergeSort(A, p, r)
  return p + k -1
pivotPstn = CHOOSEPIVOT(A, p, r)
q = PARTITION(A, pivotPstn, p, r)
Lcount = q - p
if Lcount = k-1 // k'th smallest value is in position q
  Return q
else if Lcount > k-1 // k'th smallest value is on the Left
  return SELECT(A, k, p, q-1)
else // Lcount < k-1, so k'th smallest value is on the Right
  return SELECT(A, k-(Lcount+1), q+1, r)
```

Application of UNION-FIND
Let G = (V, E) be an Undirected Graph. We want to find the Connected Components of G.
• Two vertices u and v are in the same Connected Component if there is a path between them.
This defines an Equivalence Relation on the Graph.
Which means they are Reflexive, Symmetric, Transitive
• Two vertices are equivalent if they are in the same Connected Component.
Theorem 1: (The Treeness Theorem) Let G = (V,E) be a graph. The following are equivalent.
a) G is a tree (i.e. connected and acyclic).
b) G contains a unique x-y path for any x, y in V
c) G is connected, but if any edge is removed from E, the resulting graph is disconnected.
d) G is connected, and |E| = |V| -1
e) G is acyclic, and |E| = |V| -1
f) G is acyclic, but if any edge is added to E (joining two non-adjacent vertices), then the resulting graph contains a unique cycle

Decision Tree Theorems
Theorem 1: Let T be a binary tree with n leaves and height h. Then necessarily *h* lg .
Theorem 2: Suppose a problem P has () possible outputs on input of size n. Then no algorithm for P that uses only k-ary probes of the input data can perform fewer than log () such probes on input of size n, in worst case. Thus log () is a lower bound for the (worst case) runtime of such an algorithm. IThis does not mean an algorithm that solves P in only log () steps exists, only that none do in fewer. Also note that the theorem is restricted to the class of algorithms doing only k-ary probes. Changing k changes the lower bound. Since all log functions are asymptotically equivalent, the theorem implies a single asymptotic lower bound Ω(log ()) for any k.
Theorem 3: Any algorithm that performs only array comparisons must do at least 3/2 - 2 such comparisons in order to determine both the minimum and maximum of an array of length n
The UNION Operation
• Let's assume that we have an Equivalence Relation R, and a set of induced Equivalence Classes: {E1, E2 , E3 , ... , Ek}
• UNION(x, y) changes the partition so that the sets that x and y belong to are merged into one set. For example, if x E2 and y E3, then after UNION(x, y) we have: {E1, E2 U E3 , ... , Ek } {E1, E2 U E3 , ... , Ek}

Greedy Algorithm(Activity selection):
```
Suppose the activities are sorted by finishing time
  If not, sort them. Okay, now they're sorted.
  myActivityList = []
  for k = 1,...,n:
    if we can fit in Activity k after the last thing in myActivityList:
myActivityList.Append(Activity k)
  return myActivityList
  O(n) If list is already sorted,
O(n log n) else
```

Graph Handout:
Lemma 1: If T is a tree with n vertices and m edges, then m = n -1
Lemma 2: If G is an acyclic graph with n vertices, m edges, and k connected components, then m = n - k
Lemma 3: If G is a connected graph with n vertices and m edges, then m >= n - 1
Lemma 4: Lemma 4 If G is a graph with n vertices, m edges, and k connected components, then m >= n - k
Lemma 5: Let G be a connected graph with n vertices and m edges. Suppose also that m = n -1. Then G is acyclic, and hence a tree.
Lemma 6: Let G be an acyclic graph with n vertices and m edges. Suppose also that m = n -1. Then G is connected, and hence a tree
Lemma 7: Let G be a connected graph with n vertices and m edges. Suppose also that m = n. Then G contains exactly one cycle. (Such a graph is called unicyclic.

UNION Using Up-Trees:
To merge two subsets, make the root of one Up-Tree point to the root of the other. UNION(a, f)
FIND Using Up-Trees: To determine the representative element for x, start at the node for x and follow the links to the root of x's Up-Tree.

Up-Trees An Up-Tree is a tree in which each node (except the root) has a single "pointer" that points to its Parent. A set of Up-Trees is called a Forest of Up-Trees. A Forest of Up-Trees can represent the subsets for the UNION-FIND ADT

Enhanced FIND
FIND(x): While performing this operation, compress the paths of nodes encountered along the way, so that they are closer to the root.
a. Full Path Compression. After finding the root r, make another pass from x to r making each node along the way point to r.
b. Path Halving: Make every node along the path from x to r point to its grandparent.
c. Path Splitting: Make every other node along the path from x to r point to its grandparent.
Path Halving and Path Splitting are more efficient than Full Path Compression, but they have same Worst-Case complexity. Unlike Full path compression, they don't require a second pass.
Enhanced UNION
UNION(x,y): Try to make the smaller tree point to the root of the larger.
Rank method: Try to use the higher root as the new root.
• Difficult to update height easily when path compression is done with FIND; we can use "rank" instead, which is an estimate of height.
• When x and y have different ranks, make the tree with the smaller rank point to the root of the tree with the larger rank, and keep ranks as they were.
• When x and y have the same rank, make one of them point to the other, and add 1 to the rank of the one that's now the overall root.
Size method: Try to use the root of the larger Up-Tree as the new root.
• Maintain a node count at the root of each Up-Tree.
• When a new node gets inserted, it will have to FIND the root, so it's easy to maintain this count.
• Count at nodes that are not the root doesn't matter, so it doesn't have to be maintained.

Ackermann Function A(i,j):
j + 1 if i = 0
A(i - 1,1) if i > 0 and j = 0
A(i - 1, A i, j - 1) if i > 0 and j > 0
A(i,j) grows very rapidly. For example, A(4,3) = 2^65336 - 3 So the function f(k) = A(k,k) also grows very rapidly. Its inverse function, written α(n) is called the Inverse Ackermann Function, and it grows very slowly
α(n) = min { k | A(k,k) n }
α(n) < lgk(n) = min { k | lg (lg (lg ... (n))) 1 }
[where lg is applied k times]

The Substitution Method (Handout)
We begin with the following example.
() = { 2 1 < 3
3(/3) + 3
Suppose that we are able to guess somehow that () = (log()). In order to prove this guess, we must find positive numbers c and 0 such that () log() for all 0. If we knew appropriate values for these constants, we could prove this inequality by induction. Our goal then, is to determine c and 0 such that an induction proof can be made to work.
Observe that the recurrence itself contains two base cases: = 1 and = 2. This indicates that the induction proof may also require multiple base cases. However, the inequality to be proved () log(), is actually false in the case = 1, since (1) = 2 and log(1) = 0. Since log(2) 0, the same problem does not occur at = 2. Indeed, for = 2 we seek to show (2) 2 log(2), which can be made true by a proper choice of . Therefore we take the lowest base case to be 0 = 2. It remains to determine the highest base case, which we will denote by 1. We begin by mimicking the induction step IId, with lowest base case = 2 and highest base case = 1. In what follows, it will be algebraically convenient to take log() to mean log3(). Let > 1, and assume () log( ) for all k in the range 2 < . In particular, when = /3 we have (/3) /3 log(/3). We must show that () log( ). We have
() = 3(/3) + (by the recurrence for T)
3/3 log(/3) + (by the induction hypothesis)
3(/3) log(/3) + (since for all x)
= (log( ) - 1) +
= log( ) - +
To obtain () log( ), we need to have - + 0, which will be true if 1. Thus as long as c is chosen to satisfy the constraint 1, the induction step will go through. It remains to determine the constant 1, which represents the highest base case. Since we only use the induction hypothesis in the case = /3, we require that 2 /3 < whenever > 1 . This is equivalent to > 1 6 , which indicates we should choose 1 = 5. The base cases are then = 2, 3, 4, 5, and it is required that
(2) 2 log(2),
(3) 3 log(3),
(4) 4 log(4),
(5) 5 log(5)
One checks that (2) = 2, (3) = 9, (4) = 10 and (5) = 11. To satisfy all constraints then, we take to be the maximum of the numbers { 1, 2/2 log(2) , 9/3 log(3) , 10/4 log(4) , 11/5 log(5) }. A few comparisons reveal this maximum to be = 9/3log3(3) = 3. It is important to realize that we have not proved anything yet. Everything we have done up to this point has been scratch work with the goal of finding appropriate values for and 0. It remains to present a complete induction proof of the assertion: () 3log() for all 2.

CoinChange(,)
```
[1 ; 0 ]
 = length[]
for = 1 to
  [, 0] = 0
for = 1 to
  for = 1 to
    if = 1 and < [1]
      [1, ] =
    else if = 1
      [1, ] = 1 + [1, - [1]]
    else if < []
      [, ] = [ - 1, ]
    else
      [, ] = min([ - 1, ], 1 + [, - []])
return [, ]
```

Bellman Ford: O(nm)
```
for n vertices and m edges
For v in V:
  d[v] =
d[s] = 0
For i = 1,..,n-1:
  For each edge e = (u,v) in E:
    d[v] min( d[v] , d[u] + weight(u,v) )
For each edge e = (u,v) in E:
  if d[v] > d[u] + weight(u,v)):
    return Negative Cycle Found
```

Dijkstra's Algorithm
```
Mark all nodes as unsure.
Set d[s] = 0, and set d[v] = for all other nodes.
  Loop
    Pick the not-sure node u with the smallest estimate d[u].
      Update all u's not-sure neighbors v:
        d[v] min( d[v] , d[u] + weight(u,v))
      Mark u as sure.
  Repeat loop until all nodes are marked as sure.
```

Unbounded knapsack
```
UnboundedKnapsack(W, n, weights, values):
  K[0] = 0
  for x = 1, ..., W:
    K[x] = 0
    for i = 1, ..., n:
      if wi x:
        K[x] = max { K[x], K[x – wi] + vi }
  return K[W]
```

Kruskal(G = (V,E)):
```
  Sort E by weight in non-decreasing order
  MST = {} // initialize an empty tree
  for v in V:
    makeSet(v) // put each vertex in its own tree in the forest
  for (u,v) in E: // go through the edges in sorted order
    if find(u) != find(v): // if u and v are not in the same tree. Handles matter!
      add (u,v) to MST
      union(u,v) // merge u's tree with v's tree
return MST
```

Unbounded knapsack that also finds the items
```
UnboundedKnapsack(W, n, weights, values):
  K[0] = 0
  ITEMS[0] =
  for x = 1, ..., W:
    K[x] = 0
    for i = 1, ..., n:
      if wi x:
        K[x] = max { K[x], K[x – wi] + vi }
        If K[x] was updated:
          ITEMS[x] = ITEMS[x – wi] { item i }
  return K[W]
```

Red-Black Trees:
Only info required for final is that all operations including Search, Insert, Delete, Find-Min, Update are O(log n)

Insertion-Sort: Correctness Proof Using Loop Invariant with (Weak) Induction
Loop Invariant: At the start of the iteration of the outer loop for value j, the first j-1 elements of the array are sorted, and the original values are still in the array.
Initialization: At the start of the first iteration, variable j is 2. The first element of the array, A[1], is sorted (trivially), and the original values are still in the array
Maintenance: When we begin iteration j, the previous iteration j-1 put A[j-1] where it belongs among the earlier(already sorted) elements of A. So the first j-1 elements of A are sorted, and the original values are still in the array.
Termination: When the variable j equals A.length+1, the loop terminates. The first n = A.length items of A are sorted, and the original values are still in the array, so Insertion-Sort is correct.

Proof by Induction:Huffman Encoding
Induction Hypothesis:After the it'th step: There is an optimal tree containing the current subtrees as "leaves"
Base Case: After the 0'th step:There is an optimal tree containing all the characters.
Induction Step: Suppose that the Induction Hypothesis holds for t-1 ,After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."
Want to show: After t steps, there is an optimal tree containing all the current sub-trees as leaves.
Lemma 1: If x and y are the two least-frequent letters, there is an optimal subtree where x and y are siblings.
Lemma 2: Suppose that there is an optimal tree containing as a subtree. Then we might as well replace that subtree with a new letter that has frequency
Conclusion :After the last step:There is an optimal tree containing this whole tree as a subtree.That is:After the last step, the tree that we've constructed is optimal.
Greedy Scheduling Solution
scheduleJobs(JOBS):Sort jobs by the ratio: ri=ci/ti = cost of delaying job i/ time job i takes to complete
For each i, let sorted_JOBS[i] be the job with the i'th biggest ri
Return sorted_JOBS.The running time is O(nlog(n))
Greedy Scheduling Induction
Inductive hypothesis:There is an optimal ordering so that the first t jobs are sorted_JOBS[1..t].
Base case:When t=0, this says that: "There is an optimal ordering so that the first 0 jobs are []." That's true.
Inductive Step: This says that: There is an optimal ordering on sorted_JOBS[t+1..n] in which sorted_JOBS[t+1] is first. That follows from our Lemma.(Given jobs where Job i takes time ti with cost ci ,there is an optimal schedule in which the first job is the one that maximizes ratio ci/ti)
Conclusion: When t=n, this says that: "There is an optimal ordering in which the first n jobs are sorted_JOBS." So what we returned is an optimal ordering of all n jobs

Some Lemmas for Disjoint-Set Operations with Union by Rank and Path Compression
Amortized cost is taken across m MAKE-SET, UNION and FIND operations, which are converted into O(m) MAKE-SET, LINK and FIND operations. UNION uses 2 FIND operations and 1 LINK operation.
Lemma: The cost of each MAKE-SET operation is O(1).
Lemma: The amortized cost of each LINK operation is O(α(n)).
Lemma: The amortized cost of each FIND operation is O(α(n)).
Theorem: Any sequence of m MAKE-SET, UNION and FIND operations, n of which are MAKE-SET operations, can be performed on a Disjoint Set Forest in worst-case time O(m α(n)).
• The UNION operations are done by Rank.
• FIND operations use Path Compression (any version works).
In theory, this is not linear, but in practice it is, ... since log*n   4 for values of n that come up in practice.
Induction example
Prove that for n 0, the nth Fibonacci number F(n) is less than 2n.
Proposition/Statement:
For n 0, the nth Fibonacci number F(n) is less than 2n, where:
F(0) = 0 F(1) = 1
For n 2, F(n) = F(n-1) + F(n-2)
Strong Induction.
Base Cases are for n=0 and n=1.
F(0) = 0 < 20 =1 F(1) = 1 < 21 =2 Check!
Induction Hypothesis: For all 0 k < n, the kth Fibonacci number F(k) < 2k.
[Strong Induction]
Induction Step: We need to prove that for all n 2, if F(k) < 2k for all k such that 0 k < n, then F(n) < 2n.
Okay, assume that we have a value of n 2, and that for n 2, F(k) < 2k for all k such that 0 k < n. Let's see if we can prove that F(n) < 2n.
Since n 2, F(n) = F(n-1) + F(n-2). Also since n 2, both n-1 and n-2 are 0, so we can use the Induction Hypothesis, which says that F(n-1) < 2n-1 and F(n-2) < 2n-2. Hence F(n) = F(n-1) + F(n-2) < 2n-1 + 2n-2 < 2n-1 + 2n-1 = 2n.

Hw6 3b)
We can reach any vertex if we do DFS(A,0). But if we executed DFS(C, 0), then vertices A and B would not be visited. But we could keep track of any unvisited vertices. Then, after executing DFS(C, 0), if any vertices were unvisited, we could find an unvisited vertex (e.g., B) and execute DFS(B, x+1), where x is the value returned by DFS(C, 0). And then we could execute DFS again, as long as there are any unvisited vertices. What's the Asymptotic Running Time of an efficient implementation of this algorithm? Your answer should be expressed using n, the number of vertices and m, the number of edges. Don't assume that the number of edges is O(n2). Background on this problem: This relates to an algorithm called Topological Sort, which works on acyclic directed graphs. In Topological Sort, you're given a set of edges, and you need to find a sequencing of all the vertices which "respects" the ordering specified by the edges. That means that for each edge, the source vertex appears earlier i the sequence than the corresponding destination vertex. For example, vertices could correspond to tasks, and some tasks might have to be finished before other tasks. If you can only do one task at a time, what's a sequence in which you can accomplish all the tasks? Topological Sort finds such a Sequence
Answer 3b):
An efficient implementation (which is called Topological Sort) would take time O(n) to iterate through all the vertices (to ensure that they were all visited) + O(m) to traverse each of the edges. Why O(m)? No vertex v is visited more than once. Also, each edge (u, v) is traversed only once, when we visit vertex u (which is not visited more than once). So there are two things that are done per vertex, visiting it and iterating through all the vertices to find vertices that haven't been visited. Hence work done by all vertices is O(n). And the work done per edge is also constant, so the work for all edges is O(m). Hence this algorithm is O(n+m). That's the correct answer. If you assume that m n (reasonable assumption in most situations), then it's O(m). Explanation for answer was not requested.
(Additional info about Topological Sort follows, for those who are interested in why this algorithm matters.) Suppose that we think of each edge as saying that a task must be completed before a subsequent task can be started. So for our graph, that says that A can be started immediately, but (for example) D can't be started until both A and B have been completed, because there are edges AD and BD. We'd like to find a legal order in which we can do the tasks, where "legal" means that we honor the requirements given by the edges. Topological Sort gives us a legal order for the tasks if we read off the Finish Times backwards (which is easy to do if we push tasks with new Finish Times on a Stack—sorting is not required).For our example, based on Finish Times:
• If we went to B before D, we see that A B D C E would be a legal order for the tasks.
• If we went to D before B, we also see that A B D C E would be a legal order for the tasks. (Same order; that's not always the case.)
That's easy to see in this example, but it may be much more difficult to determine a legal order when there are more tasks and more edges. Topological Sort determines a legal order only when the graph has no cycles (but can be written to detect cycles when cycles exist). And the Runtime for Topological Sort is O(n+m), or if we assume that m n, O(m), and that's a great Runtime

Strong induction: Merge-Sort returns a sorted array that has the original values in it.
Initial/Base Case (number of elements is 1): A 1-element array is always sorted.
Maintenance: Suppose that:
– the Left half L, which is A[p:q], is the original A[p:q] sorted, and
– the Right half R, which is A[q+1:r], is the original A[q+1:r] sorted.
Then Merge(L,R) places values in A[p:r] that are the original values in A[p:r] sorted.
Termination: The top recursive call, Merge-Sort(A,1,n) places values in A[1:n] that are the original values in A[1:n] sorted.
Hence Merge-Sort correctly sorts the array A[1:n]

Old Final 11c:
Since the Bellman-Ford algorithm solves Single Source Shortest Path for graphs in which the edge weights are arbitrary values, why would anyone ever choose to use Dijkstra's algorithm, rather than always using Bellman-Ford?
Answer 11c): Bellman-Ford algorithm runs in time O(nm) on a graph G with n vertices and m edges, which is O(n3) on a graph that has O(n2) edges. Dijkstra is a lot faster. Hope that you justified that using at least one of the following:
• If we use Red-Black trees, Dijkstra runs in time O((n + m)log(n)), which is O(m log n) if we assume that m n.
• If we use Priority Queue/Heap the running time is O(n log(n) + m log(n)), which is O(m log(n)) if we assume that m n.
• If we use Fibonacci heaps (not discussed in Lecture, so you're not expected to remember this one), Dijkstra runs in (amortized) runtime O(m + nlog(n))

Induction proof ex: For all integers n 1, the sum of the first n integers is n * (n+1) / 2.
Proposition P(n): For all integers n 1, the sum of the first n integers is n * (n+1) / 2
Base Case P(1): The sum of the first 1 integer is 1, which equals 1 * (1+1)/2. Check!
Induction Hypothesis: For integers n 1, the sum of the first n integers is n * (n+1) / 2.
Induction Step: We need to prove P(n+1),
For integers n 1, the sum of the first n+1 integers is (n+1) * (n+2) / 2
i(n+1) = i +f(n+1) = n * (n+1) / 2 + (n+1)

Homework 2 1b) (asymptopic equivalence)
Prove that if f(n) ~ g(n) then f(n) = g(n) + o(g(n))
Assume that f(n) ~ g(n). By definition of ~, that means that   ()/ () = 1. We need to prove that there is a function h(n) = o(g(n)) such that f(n) = g(n) + h(n). Let h(n) = f(n) - g(n). Then g(n) + h(n) = g(n) + (f(n) - g(n)) = f(n), so yes, f(n) = g(n) + h(n). But does h(n) = o(g(n))? To prove that, we'll show that   *h*()/ () = 0.   *h*()/ (). = ()-()) /() = [() /() - () /()] = ()/ () - 1 = 1 – 1 = 0 So for h(n) = f(n) - g(n) we've shown that: f(n) = g(n) + h(n) and h(n) = o(g(n)) are both true,
Homework 4 4b) Old Final 5 5b) jug matching
There are n red and n blue jugs. One of each red and blue jug match in size. Find the matching sets using only >, =, < comparisons in Ω(log).
Answer 4b): Let the red jugs be R1, R2 ,..., Rn and the blue jugs be B1, B2 ,..., Bn. Begin by comparing R1 to each of the n blue jugs until a match is found. The blue jugs could be matched to the red jugs in any order. Thus, there's a total of ! ways to match the n red jugs to the blue jugs, so this problem has ! possible answers, and hence its Decision Tree has at least n! leaves.
We proved a Theorem that if a k-ary Decision Tree has n leaves, then the height of the tree must be at leastlogkn. For this problem, k=3, since two jugs could hold equal, amounts or the blue jug could hold more, or the red jug could hold more. Any algorithm can therefore be represented by a decision tree whose height satisfies: h log3(!) = Ω(log(!)) = Ω(log) using Stirling's Approximation.
[Remember that the base of the log only changes value be a constant factor.] Therefore, any algorithm solving this problem must perform at least Ω(log) comparisons
Hw6 2c:BFS solves SSSP for both directed on undirected graphs. DFS wouldn't solve the SSSP problem for either directed or undirected graphs. Prove that DFS doesn't solve SSSP for undirected graphs. Do this by providing a counterexample graph, showing that DFS solves SSSP incorrectly for that counterexample graph.
Answer 2c): Using DFS, you report the distance to a vertex as soon as you encounter it, based on the depth-first path that you're on. That doesn't have to be the shortest path to the vertex (and often, it isn't). Simple counterexample graph: Suppose that your undirected graph is the Complete Graph on n vertices, with an edge between every pair of vertices, with each edge weight equal to 1. Then the distance that you would report using DFS would be 0 for the source vertex, 1 for the first vertex you reach, 2 for the second vertex, ..., and k for the k'th vertex you reach. But the distance for every vertex should be 1.

Hw6 #4: For both an actual MST (if justified as mst) would also be acceptable, but more work
A) Prove that there is a MST for the above graph that includes edge BC
Prim's algorithm finds the Minimum Spanning Tree (MST) no matter which vertex we use as a starting vertex. If we start at vertex C (not vertex B), the first edge that we choose will be BC, because that's the lowest cost edge that includes C. We know that this Greedy choice won't prevent us from reaching an Optimal Solution, an MST. So there must be an MST that includes edge BC.
B) Prove that there is a MST for the above graph that includes edges AD, DE, and FG
Kruskal's algorithm finds the Minimum Spanning Tree (MST), always adding the edge with minimum cost that doesn't create a cycle. The three edges DE (weight 2), AD (weight 3),and FG (weight 3) are the edges that have the lowest weights in the graph, and they don't cause a cycle. We know that these Greedy choices won't prevent us from reaching an Optimal Solution, an MST. So there must be an MST that includes edges AD, DE and FG

Old Midterm 2 Question 4: Using the Substitution Method, prove that the solution of the Recurrence:
T(n) = T(n/2) + n2 if n > 1
T(1) = 1
is O(n2 lg(n)) for n 3 by using the guess f(n) = n2 lg(n)). You must use the Substitution Method in your proof. You just have to show the Backward Induction, but you should identify all the Base Cases that will have to be addressed in the Forward Induction.
Answer 4: Proposition: T(n) c*n2 lg(n)) for n n0 for some c and n0 not determined yet.
Let's go through the Backwards Induction using the guess f(n) = n2 lg(n)).
Induction Hypothesis: T(n) c*n2 lg(n)) for n n0 for some c and n0 to be determined.
Base Cases: That Induction Hypothesis is false when n=1 because lg(1)=0. When n =2, T(2) = 5 and 22 lg(2) = 4, so we'll need c 5/4 for the Ind Hyp to be true when n=2. Or we could start with n0 = 3 with c=1, since T(3)=10 81 lg(3). We'll do that. The Q2 problem statement included the odd phrase "for n 3" as a suggestion that you use n0 = 3.
Backwards Induction Step:
Now let's assume that T(k) c*k2 lg(k) for 3 k < n. We want to be able to prove that T(n) c*n2 lg(n)).
T(n) = T(n/2) + n2
 c*(n/2)2 * lg(n/2) + n2 By Ind Hyp, since n/2 < n
 c*(n/2)2 * lg(n/2) + n2   Since x < x and lg is increasing
 c*n2/4 * lg(n) + n2   Multiplying, and since lg(n/2) < lg(n)
= n2 * [c* lg(n)/4 + 1)   Factoring out n2
 c*n lg(n) whenever c*lg(n)/4 + 1 c*lg(n)
Okay, when is c*lg(n)/4 + 1 c*lg(n)? By algebra, that's true when 1 (3c/4) * lg(n), which is true when 4 3c lg(n), which is true whenever 2^4 2^(3c lg(n)), which is true when 16 n3c, which is true (picking c=1) whenever 3 n. So we've completed the Backwards Induction, and supposedly shown that T(n) n2 lg(n)) for 3 n. [Or alternatively, we've shown that T(n) 5/4 * n2 lg(n)) for 2 n. We'll continue using n0= 3, not 2 below. The next discussion on Base Cases similar, but a little different, if we use c=5/4 and 2 n. Either alternative is okay.] Except that our Induction Step won't work unless n/2 3, i.e., unless n 6, since the Induction Hypothesis only holds when k3.. So we'll have to check that our guess works not only for 3, but also for 4 and 5. (You only had to identify the Base Cases that require evaluation; you didn't have to evaluate them. But we'll evaluate them.)
We've already checked above that T(n) n2 lg(n)) when n=3.
When n=4, T(4) = T(2) + 42 = 5 + 16 = 21 16 * lg(4) = 32.
When n=5, T(5) = T(2) + 52 = 5 + 25 = 30 25 * lg(5) since lg(5) > 2. You weren't required to do the Forward Induction for Q4

Old Midterm 2 Question 7a
Write pseudocode that finds an LCS of X and Y. It's okay to find the LCS backwards, appending to a sequence of characters.
You should assume that you already have the C[m,n] array.Your algorithm's runtime should be efficient.

```
Result = Empty String // Empty Common Subsequence
i = m // Position in X
j = n // Position in Y
WHILE i > 0 AND j > 0 // Looking for matches until a position hits 0
  IF X[i] = Y[j] // Found match; diagonal move
    Append (or Prepend) C[i,j] to Result
    i = i -1
    j = j – 1
  ELSE IF C[i,j] = C[i,j-1] // No match; move left
    j = j – 1
  ELSE IF C[i,j] = C[i-1,j] // No match; move up
    i = i -1
  ELSE SIGNAL ERROR // Impossible
```

• Could do the comparisons between C[i,j] and the left/up positions in either order.
• Must have gotten value of C[i,j] from either left or up neighbor (or both) if X[i] is not equal to Y[j].
• Okay if you didn't check for the impossible error case

Old Midterm 2 Question 7
We used the following algorithm to solve the Longest Common Subsequence (LCS) Problem for strings X and Y, using Dynamic Programming. The length(X) is m, and the length(Y) is n. X[i] is the ith character of X, and Y[j] is the jth character of Y.

```
LCS(X, Y):
  C[i,0] = C[0,j] = 0 for all i = 1,...,m, j=1,...n.
  For i = 1,...,m and j = 1,...,n:
    If X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
    Else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

After running this algorithm, the LCS of X and Y has length C[m,n]. But we haven't found an actual LCS of strings X and Y.

Old Final Question 7 (Longest Weakly Decreasing Subsequence, Dynamic Programming):
This question involves an efficient to solve the Longest Weakly Decreasing Subsequence problem. We need to find the length of the longest subsequence in an array A[1:n] such that the array values for that subsequence are never getting bigger.
For example, if the values in array A[1:8] are 21, 11, 21, 5, 16, 16, 14, 2:
• Positions 1, 2, 4 and 8 of A are a Weakly Decreasing Subsequence, with values 21, 11, 5, 2. That subsequence has length 4.
• But the Longest Weakly Decreasing Subsequence of A has length 6, corresponding to positions 1, 3, 5, 6, 7, 8, with values 21, 21, 16, 16, 4, 2.
• And the value that we are seeking, the length of the Longest Weakly Decreasing Subsequence, is 6.
7a): What is the Optimal Substructure for a Dynamic Programming solution to this problem?
The Optimal Substructure is that the optimal solution to the Longest Weakly Decreasing Subsequence problem involves computing the length of the Longest Weakly Decreasing Subsequence of the subarray A[1:i] that ends with index i. To calculate L[i], we consider the Longest Weakly Decreasing Subsequence lengths L[j] of all the previous values, and select the greatest such that A[j] A[i] and j<i. If you find any value, you set L[i] to be the maximum (of the appropriate L[j] values) + 1, and if you don't find any, you set L[i] to be 1
7b)Provide the Recurrence that can used to solve this problem, and explain clearly why that Recurrence is correct.
Let L[i] be the length of the Longest Weakly Decreasing Subsequence of an array ending at index i. If there is at least one j such that A[j] A[i] and j < i, then L[i] should equal the max of their L[j] values + 1. But if there aren't any such j, then L[j] should be 1.
Thus, L[i] can be written as:
L[i] = 1 + max(L[j]) //if there is at least one j such that j < i and A[j] A[i]
L[i] = 1 //if there are no such j
Note that the second clause implies that L[1] = 1.
The recurrence is correct because you get an weakly decreasing subsequence if you put A[i] after a sequence that ends in A[j] if and only if j is before it and A[j] A[i]. And the longest of those (+1 since you're appending A[i]) gives you the longest subsequence ending in A[i]. It is, however, possible that A[i] is bigger than all the preceding values, in which case L[i] must be 1
7c) Provide pseudocode for your algorithm.

```
LongestWeaklyDecreasingSubsequence(A[1:n])
FOR i = 1 TO n DO
  L[i] = 1
  FOR j = 1 TO i-1 DO
    IF ( A[j]  A[i] ) THEN // Maybe we can improve L[i]
      L[i] = MAX(L[i], L[j] + 1)
RETURN MAX( L[j] ) //where the maximum is taken over all j from 1 to n
// We could also just keep track of the biggest value of L[i] that's ever set in the algorithm so that we don't have to iterate
through the values of L[i] at the end.
```