**Insertion sort:** Like one sorts cards. Consider looking through the cards one by one from left to right, and then any time you see something that should be lower you move all the cards up and put that card in the correct spot.

**Asymptotic bounds** refer to O, Omega, etc.

**Red black trees:** binary trees that are ensured balance. Take O(log n) in worst case. || You can build a **min/max heap** in O(n) time. || **Quick sort is worst** when array is already sorted or reverse sorted. **|| Quick sort is not very efficient** on small lists and is slow if there are many identical keys. || **Quick sort** is not stable. || **Stable sorting algorithm**: a sort in which equal items maintain the same relative order. || We can make **quicksort** run in Theta(n log n) using median finding. Can get ith largest element in theta(n) worst case using median. || **Randomized quick sort** is no different best/worse case than quicksort. ||

**Binary and Splay Tree:** search log(n), insert log(n), deletion log(n).

**Merge sort:** Recursive call to merge sort on two halves of the call, and at the end a call to merge which will actually do the work for us of combining sorts.



9. When you break a substring at the ith position, you add the length of the substring and the cost of breaking the two subproblems.

Then you have two subproblems: breaking the string from 0 to i and breaking the string from i to n. We consider as possible breakpoints all the values in the L set that have not been used so far.

When we are breaking the substring Si, Si+1, …, Sj we know the precomputed values of breaking such string at the breakpoints not used left (these are the subproblems, so if we apply bottom-up we know the smaller subproblem values). If we take the cut that has the optimal solution (optimal solution of the subproblem), then by adding the length of the string we will get the best optimal solution for the problem. If there is another subproblem with the best optimal solution, then by cut-and-paste we can get rid of our solution and get the solution using the best optimal solution of the subproblem, proving that we will get the best optimal solution.

(Kind of like the way he explains the best optimal substructure in the slides for the optima BST).

Need to show this is O(n)

T(n) = T(n - 1) + O(1)

Plan to show by def

T(n) <= cn for all n >= n0

assume

T(n - 1) <= c(n - 1)

then

T(n) = T(n - 1) + O(1)

T(n) <= c(n - 1) + d

T(n) <= cn + d - c

T(n) <= cn for c >= d

To Solve: *T*(*n*) *=* 3*T*(⎣*n*/3⎦) *+ n*

* Guess: *T*(*n*) *= O*(*n* lg *n*)
* Want to prove: *T*(*n*)≤ *cn* lg *n*, for some *c* > 0.
* Hypothesis: *T*(*k*)≤ *ck* lg *k*, for all *k* < *n*.
* Calculate:  
    *T*(*n*) *=* 3*T*(⎣*n*/3⎦) *+ n* (by definition)

≤ 3*c* ⎣*n*/3⎦lg⎣*n*/3⎦ *+ n*  (by I.H.)

≤ *c n* lg(*n*/3) *+ n* (prop. floor)

= *c n* lg *n* –  *c n* lg3  *+ n* (prop. lg)

= *c n* lg *n* – *n* (*c* lg3–1)

≤ *c n* lg *n*  as long as *c* ≥1/lg 3

* + Guess: *T*(*n*) = *n* lg*n* + *n*.
  + Induction: we establish the guess by induction on *n*
    - **Basis:** *n =* 1 ⇒ *n* lg*n* + *n* = 1 = *T*(*n*).
    - **Hypothesis:** *T*(*k*) = *k* lg *k* + *k* for all *k* < *n*.
    - **Inductive Step:** *T*(*n*) = 2 *T*(*n*/2) + *n*

**=** 2 ((*n*/2)lg(*n*/2) + (*n*/2)) + *n*

*= n* (lg(*n*/2)) + 2*n*

= *n* lg*n* – *n* + 2*n*

= *n* lg*n* + *n*

**Quick sort:** (again recursive) You can have non-random and random. In non-random, you choose the last element to be the pivot, and then you organize everything into what’s less than and what’s greater than the pivot and then once you’ve compared everything you put the pivot right between those two groups. Then, you call again on the two groups, those greater and those less than the pivot. The loop invariants are that everything below is in one area, above is in another, and pivot is certain index.

Recurrence: *T*(*n*) = 1 if *n* = 1

*T*(*n*) = 2*T*(*n*/2) + *n*  if *n* > 1

**Heap sort:** Put all the elements into a binary tree, and then pull them out one by one?

Breath first:

Run-time: O(V+E).

Essentially, you start at some position and go uniformly out, in a layered fashion.

Invariant: Queue contains all (discovered) adjacent vertices in discovered/increasing distance order from source

Depth first:

Run-time: O(V+E).

Invariant: Vertex’s discovered time is > that of all previously discovered vertices and finishing time> all finishing times of vertices adjacent.

1. State what you want to prove.

2. For which variable.

3. Base cases.

4. “I want to prove for greater than base”

5. Induction Hypothesis (for greater than base < k < n).

6. Want to prove for n.

7. Work —> break n into smaller cases.

1 **procedure** DFS(*G*,*v*):

2 label *v* as discovered

3 **for all** edges from *v* to *w* **in** *G*.adjacentEdges(*v*) **do**

4 **if** vertex *w* is not labeled as discovered **then**

5 recursively call DFS(*G*,*w*)

void quickSort( int a[], int l, int r)

{ int j;

if( l < r )

{// divide and conquer

j = partition( a, l, r);

quickSort( a, l, j-1);

quickSort( a, j+1, r);

}}

int partition( int a[], int l, int r) {

int pivot, i, j, t;

pivot = a[l];

i = l; j = r+1;

while( 1)

{

do ++i; while( a[i] <= pivot && i <= r );

do --j; while( a[j] > pivot );

if( i >= j ) break;

t = a[i]; a[i] = a[j]; a[j] = t;

}

t = a[l]; a[l] = a[j]; a[j] = t;

return j;

}

If vertex is connected, there is a path between every pair of vertices. |E| \geq |V| -1.

If |E| = |V| -1, then it’s a tree.

A forrest is a collection of trees.

Adjacency lists just stores a linked list of all vertices connected to a given vertex. So max size would be V^2.

Adjacency list requires O(V+E) storage for directed and undirected.

Adjacency is best for sparse, matrix is best for dense.

Minimum Spanning Trees:

Kruskals: look at every edge, from smallest weight to greatest, and add them to the minimal spanning tree and if you hit a cycle, don’t add that edge, and continue until you have all of the vertices. Runtime O(E log V).

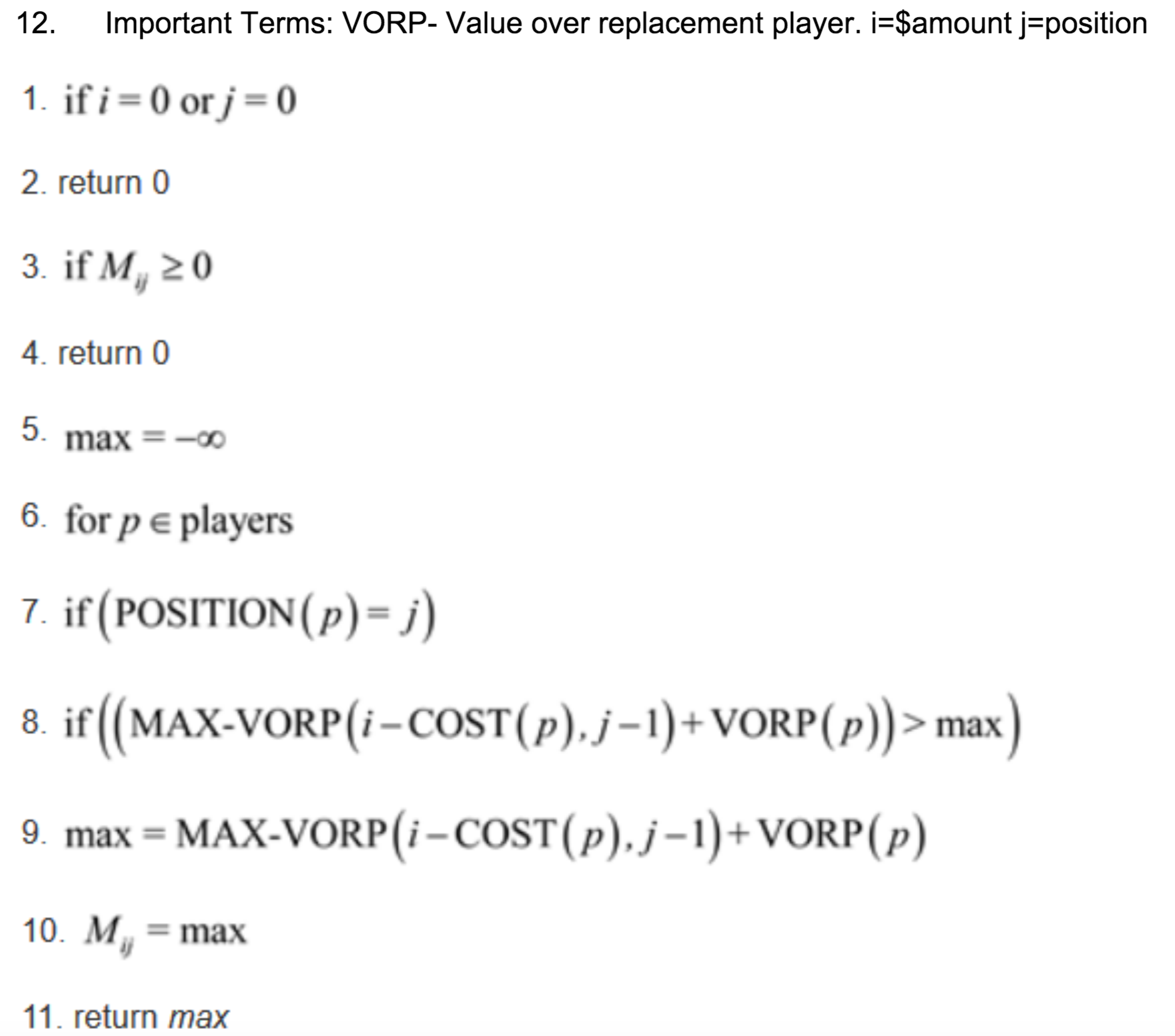
Primm’s: start with a node, look at all connected nodes and chooses the one that has the smallest edge. Runtime O(|E| log |V|).

Finding shortest past between two nodes:

Dijikstra: O(V^2). It picks the unvisited vertex with the lowest-distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Bellman Ford: time complexity O(V\*E). Space: O(V). The Bellman-Ford algorithm is one of the classic solutions to this problem. It calculates the shortest path to all nodes in the graph from a single source.

* Base: for all *n* ≤ 24, *T*(*n*) ≤ 24*n*
* For *n* > 24, *T*(*n*) ≤ *an+ T*(*n*/5) + *T*(7*n*/10+1.2)
* We want to find *c*>0 so for all *n*>0 *T*(*n*) ≤ *cn.*
  + Base implies *c* ≥ 24
  + *T*(*n*) ≤ *an+ T*(*n*/5) + *T*(7*n*/10+1.2)  
     ?≤ *an+ c n*/5 +  *c* 7*n*/10 + 1.2*c  
     = cn* – (*c n*/10 – *an –* 1.2*c*) *= cn* – ((*c*/20 – *a)n + (n*/20 *–* 1.2)*c*)  
     ≤ *cn,* as long as c ≥ 20*a*.
  + So, *c =* max(24, 20*a*) works



(a) Costs and prices at all docks are the same: fi = F, and pi = P, gi = G.

If any di>D, then we cannot reach dock i from the previous, even on a full tank, so

assume all di<=D. Who we buy gas from does not matter.

If P-F is negative, or if we need to make more than K stops, then the best solution

is to minimize the number of stops. The problem of minimizing stops has both

optimal substructure and the greedy choice property: If someone gives us an

optimal solution that starts by going from dock 0 to dock j, then

optimal subst: we have a route with the minimum number of stops from j to n.

and greedy choice: we can adjust the first dock as late as possible and still be optimal. Thus, the greedy algorithm of always going to the last reachable dock gives an optimal solution. If we have fewer than K stops and P-F>0, then we want to add stops to deliver product. Total: O(n) time.

Alt) define D\_k to be the entire distance up until dock k.

b) Docking fees and prices may differ by dock, but gas costs are the same at all docks and K=n, so you have plenty of product. You can do this in n time. Input: for docks [i..j], costs c\_i=0,c\_i+1,…c\_j-1, c\_j=0. Distance limit D and distances d\_i+1,…d\_j. Output: for docks k \in [i..j], min cost C\_k and distance D\_k from i. Main observation: You’ll never stop at a more expensive dock if you have enough gas to reach a cheaper one. Here’s the algorithm: initialize C\_i = 0, Q <- i. Invariant: Q stores candidate docks within D at dock k in increasing order of cost and distance. Code:

for k=i+1 to j:

while D\_k-D\_Q.front > D: pop(Q\_front);

C\_k=C\_Q.front +c\_k;

While C\_k <= C\_Q.back: pop(Q.back);

Push(k,Q.back);

Time is O(n) – charge the cost of popping in ‘while’ loops to the pushes.

B alt) To maximize your profit, you want to stop at each profitable dock (p\_i>=f\_i), plus docks 0 and n. Each time you stop, top off your gas. The only work that remains is if there is a gap >D between two profitable docks, i+j, find the cheapest path between them.

c) As in b, docking fees and prices may differ, gas costs are the same, but now you only have K unites of product. So, if K>=n, use (b). Otherwise, modify the b) code to be limited by k (apparently by keeping k copies of the queue Q from above, getting run time of O(nK)=O(n^2) in worst case. Alt (c) using b’s graph) we can make a graph G= ([i..j],{(k,l)|d\_k+1+…d\_l < D}) with weight w(k,j)=0 and w(k,l)=f\_l-p\_l forall i<l<j. Use Dikstra to find shortest path I to j. (Takes O(n^2) worst case). Now c’s extension: Create graph G=((i,l), E) with edges: don’tsell: ((i,l),(j,l)) at cost f\_j forall docks i+j within D. Sell if l>0: ((i,l),(j,l-1)) at cost f\_j -p\_j forall docks i,j within D. The minimum-cost path in this graph from (oK) to (n,L) for any l \in [0,K] is minloss (profit; f<0).

d) Docking fees and prices are zero, and you want to minimize your gas cost. Input: gas prices g\_i, with g\_0=0, and distances d\_i, D\_i. Output: amount of gas to buy at each dock, a\_i, minimizing Sig(a\_i g\_i) subject to Forall k>0 (D\_k/D <= Sig\_i<k (a\_i\*g\_i)<=(D\_k-1/D)+1 and Sigma\_i a\_i\*g\_i = Dn+1. Those terms are (have gas to reach dock k), (don't overfill at k-1) and (fill at n) respectively. Key: an optimal solution burns cheapest gas available, so never pases dock i carrying gas that gosts >g\_i. Easy greedy algorithm: if you are stopped at dock i, look ahead to all docks within distance D: if the cheapest gas, at j, has g\_j>g\_i, then completely fill your tank now, stop at j. else let j be the nearest dock with g\_j <= g\_i top off, if necessary, to D\_j-D\_i/D, which just reaches j. Time: need to find min in sliding window of width D. O(nlogn) with a heap/priority queue or by sorting gas and adding intervals in decreasing order.

e) Not going to type the entire thing out. Input: gas g\_i, fee f\_i, price p\_i, distance D\_i, D, and K. Output f\_i = { 1 if dock at i, 0 if don't}. sigma\_i = {1 sell at i, 0 don't}. a\_i =amount of gas to buy. Note: if we knew where we where stopping, we'd pay the fee at each, sell product at the best K and use d to decide on gas. (d) suggests that we track gas by where we will run out: either at a dock j with cheaper gas than last bought or at D\_i+D if we filled the tank at dock i. Thus merging {D\_0...D\_n} with {D\_0+D...D\_n+D} gives the list of 2n numbers that represent possible tank status: if we were at D\_i with tank X>D\_i, then D+D\_i -X / D fills the tank and D\_j-X / D gets to D\_k+D>=D\_j>=X. Let's figure out, for each dock, the minimum cost C\_i,k,g, to be at i with k product and g gas (after fees, sales and gas purchase.) i\in [0...n] dock number. k\in [0..min(n,k)] whenever need more than n product. g \in {D\_i, D\_i+D} is where the tank empties. This is an n x K x 2n table at worst. Then you compute. (code ignored) Solution has cost Min C\_n,k,D\_n+D and is recovered from the F pointers. Time is O(n^3) table entries each look at O(n^2) dock and gas values: O(n^5) worst case.

Take unsorted list of n numbers A[0…n-1] and sort the i smallest elements. Run time?

Build a min heap in O(n) time then pull out the first i elements in O(log(n) a piece. Totally O(n+i\*log(n)) time.

What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

Assume that A[*i*]<A[*i*+1}, so *(i,i*+1) is an inversion. What does a swap A[*i*]↔ A[*i*+1] do? It removes inversion (*i,i+*1), and changes other inversions involving i to i+1 and vice versa. Specifically, for all *k≠i*, we change *(i,k) →(i+1,k); (k,i) →(k,i+1); (i+1,k) →(i,k); and (k,i+1) →(k,i)*. The total number of inversions decreases by one, thus the number of swaps performed is exactly the initial number of inversions, IA. The total time is O(n+IA)

Another induction. For some constants c,d,e and for all n>0, T(n) <= cn log(n) +dn +e.

Base case: n=1. T(1)=1 <= c\*1\*log(1) + d\*1 +e iff d+e >= 1.

Induction step: we want to prove this for n>1.

Induction hypothesis: assume that for all k<n T(k) <= cklogk + dk + e.

To show: T(n) <= c\*n\*log(n) + dn + e

T(n) = 3T(n/3) + n.

<= 3(c\*n/3 \* log(n/3) + dn/3 +e) + n

= cn(log(n)-log(3))+dn+n+3e.

<= cn\*log(n)+dn+e-(clog(3-1)n+2e.

<= cn\*log(n)+dn+e.

Choose e=0, d=1, c>1/log(3)

All sorts and searches in Big-Oh (average, worst):

Depth first search: none, |E| + |V|.

Breadth first search: none, |E|+|V|.

Binary search: log(n), log(n).

Shortest path by Dijkstra using min-heap as pri-Q: (|V| + |E|) log |V|, same.

Shortest path by Dikstra, using unsort array as pri-Q: |V|^2, same.

Shortest path by Bellman-Ford: |V||E|, same.

Quicksort: nlog(n), n^2.

Mergesort: nlog(n), nlog(n).

Heapsort: nlog(n), nlog(n).

Insertionsort: n^2, n^2.

Bubblesort: n^2, n^2.

Select sort: n^2, n^2.

Bucket Sort: n+k, n^2.

Radix Sort: nk, nk.

All the sorts of trees have log(n) for search, insertion, deletion.

Quicksort runs in Θ(n2) worst-case time. true

b. An example of a worst-case input for Quicksort is list that is already sorted. true

c. Quicksort runs in Θ(n) best-case time. false

d. Using median finding, we can make Quicksort run in Θ(n lg n) worst-case time. true

e. Quicksort runs in expected Θ(n lg n) time on any input that has distinct elements. false

f. Randomized Quicksort has the same worst-case and best-case times as Quicksort. true

g. Randomized Quicksort runs in expected Θ(n lg n) time on any input that has distinct elements. true

h. Randomized Quicksort runs in expected Θ(n lg n) time on a list that is already sorted. true

Indicator variable problems: suppose that n double precision numbers x\_1 < x\_2 < x\_n are shuffled into random order and inserted one-by-one into a binary search tree. Show that the expected number of comparisons for all insertions is nlog(n). Not great solution: for ith insert into a binary tree we expect an average of log(i) comparisons, we make insertions from 1<i<n so we get sigma\_0<i<n log(i) = log(n) +log(n-1) +... +log(1) = log(n!) and log(n!) \in O(nlog(n)).

Alt: Note: only time x\_i and x\_j are compared when first from [x\_i,...,x\_j] inserted into tree. So what's prob that two x\_i and x\_j are compared? Say Z\_ij = 1 if compared 0 else. when j=i+2, P(Z) = 2/3, and 1/2 for j=i+3. So P[Z\_ij] = 2/(j-i+1). Need to sum over all inserts. E[Z] = ∑1≤i<n ∑i<j≤n 2/(j - i + 1) ---> E[Z] = 2 (∑1≤i<n ∑i<j≤n 1/(j - i + 1), let h=j-i+1, so we get E[Z] = 2 ∑1≤i<n ∑2≤h≤n-i+1 1/h, which means E[Z] ≤ 2 ∑1≤i<n Hn = 2 Hn ∑1≤i<n 1 so E(Z) < 2\*n\*log(n) so E(Z) \in O(n\*log(n)).

***o*(*g*(*n*))** = {*f*(*n*): **∀** ***c* > 0**, **∃** ***n*0 > 0** such that **∀** *n* ≥*n*0*,* we have0 ≤ *f*(*n*)< *cg*(*n*)}.

Holds true before the first iteration of the loop – **Initialization.**

If it is true before an iteration of the loop, it remains true before the next iteration – **Maintenance**.

When the loop terminates, the **invariant ― along with the fact that the loop terminated** **―** gives a useful property that helps show that the loop is correct – **Termination**.

Breath first:

Run-time: O(V+E)

Essentially, you start at some position and go uniformly out, in a layered fashion.

Invariant: Queue contains all (discovered) adjacent vertices in discovered/increasing distance order from source

If vertex is connected, there is a path between every pair of vertices. |E| \geq |V| -1.

If |E| = |V| -1, then it’s a tree.

A forest is a collection of trees.

Adjacency lists just stores a linked list of all vertices connected to a given vertex. So max size would be V^2.

Adjacency list requires O(V+E) storage for directed and undirected.

Adjacency is best for sparse, matrix is best for dense.

Depth first:

Run-time: O(2V+E)

Invariant: Vertex’s discovered time is > that of all previously discovered vertices and finishing time> all finishing times of vertices adjacent.

1. BFS code: Enqueue the root node
2. Dequeue a node and examine it
   * If the element sought is found in this node, quit the search and return a result.
   * Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

induction: P(h) = h + 2P(h-1), P(0) = 1

want to show that ∃c>0,d∀h ≥ 0P(h) ≥ c2h + dh

base case: P(0) = 1 ≥ c2h + dh. Pick easy c and d: c = 1, d = 0

P(0) = 1 ≥ 1 + 0 ✓

for h = n, assume true for n - 1.

P(n) = n + 2P(n-1) ≥ n + 2(2n-1) = 2n + n ≥ 2n for all n ≥ 0.

Q.E.D.