1.

1. (a)

8. $C \leftarrow C + j - 1$

12. —

14 while $i \leq l$ do

$A[k] \leftarrow L[i]$;

$i \leftarrow i + 1$;

$k \leftarrow k + 1$;

$C \leftarrow C + r$;

while $j \leq r$ do

$A[k] \leftarrow R[j]$;

$j \leftarrow j + 1$;

$k \leftarrow k + 1$;

Termination At each step we increment i or j. So it is clear at some point $i \geq l$ or $j \geq l$. In the loops that follow (14), the other pointer will be incremented until it is greater than l or r, respectively.

When the loop exits we have either $i > l$ or $j > r$, so only one while loop will execute. All elements of either L or R have been added to A, then, if L has been exhausted, the second while loop is the same as case $L[i] > R[j]$ in the main loop, else if R has been exhausted, the first while loop is the same as case $L[i] \leq R[j]$. Thus at the completion of the code in 14, $k = l + r + 1$, $i = l + 1$, $j = r + 1$, and by the invariants we have that A is the concatenation of $L + R$ sorted in non-descending order and C is the # of inversions in LR.

Partial Correctness

(b) We use the following invariants: (1) C equals the number of inversions in $L[1, i-1]R$

(2) $A[1, k-1]$ is the first $k-1$ elements of the concatenation of L and R sorted in non-decreasing order

(3) $i, j, k \in \mathbb{Z}^+$, $i \leq l+1$, $j \leq r+1$, $k = l + r + 1$

Base Case

Initially $i = 1$, so $L[1, i-1]R = L[1, 0]R = R$, and R is sorted, so it has 0 inversions. Since $C = 0$ initially, (1) follows. Now, since $k = 1$ initially, $A[1, k-1] = A[1, 0]$ is an empty array, so it is the first 0 elements of the concatenation of L and R sorted in non-decreasing order, thus (2) follows. (3) follows based on initialization of i, j, k.

Inductive Step

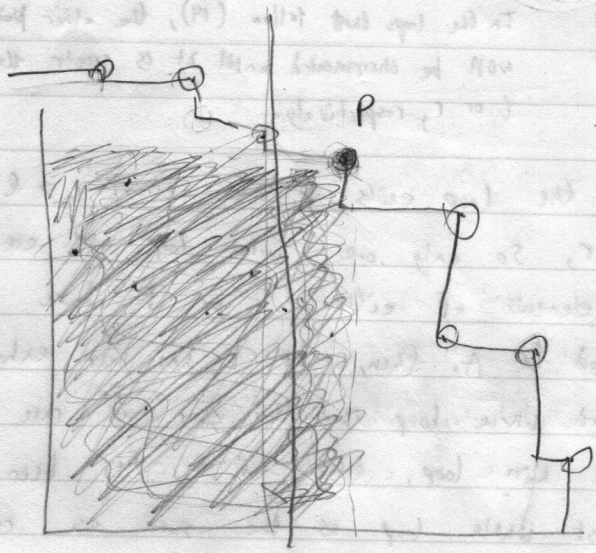
Now, suppose $L[i] \leq R[j]$. In the previous iteration, we either added $L[i-1]$ to A or $R[j-1]$. If it is the former, we have $L[i] \geq L[i-1]$ since L is sorted, and so adding $L[i]$ to A makes (2) hold. If it is the latter, then in the previous iteration we had $R[j-1] < L[i]$, so again adding $L[i]$ to A makes (2) hold. Thus (2) holds when $L[i] \leq R[j]$. Then all elements of A are less than $L[i]$. Now to handle invariant (1), we must count the number of inversions of $L[i]$ with R. This is the number of elements of R less than $L[i]$, which is $j-1$, so invariant (1) holds.

Note that by exchanging L with R and i with j, we have the result for when $L[i] > R[j]$. Finally, (1) holds by the conditions of the while loop and first we only ever increment by 1.

Approach 1: Divide & Conquer

2.

Solution (set of points S):



1. Find median x coordinate of all points and split points into sets L (left of median) and R (right of or equal to median) (largest x)
2. Find largest y-valued point in R, it must be undominated. Call it P.
3. Remove points in L and R dominated by P.
4. Recursively call this procedure on L and R (if they are nonempty) and return Solution(L) ∪ Solution(R) ∪ P

if |S| = 1 the single point is undominated. Assuming Solution is correct for input of size ≤ n, the argument given then shows Solution is correct for n+1.

Proof of Correctness

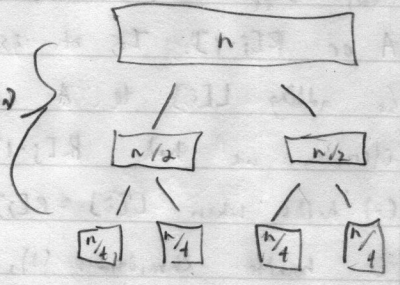
We first show correctness assuming Solution(L) and solution(R) return correctly. This is later proven using induction.

We see that in step 3, P is an undominated point (max x if tie) because it has maximum y value in R, and everything in L has smaller x value. Removing dominated points of P means that every point in Solution(L) is not dominated by P, thus is an undominated point of the original set S. The same is true of Solution(R) for the same reasoning.

Observe that every undominated point in S is either to the right or above P, which is still present in L and R. Thus Solution(L) ∪ Solution(R) ∪ P covers exactly all undominated points in S.

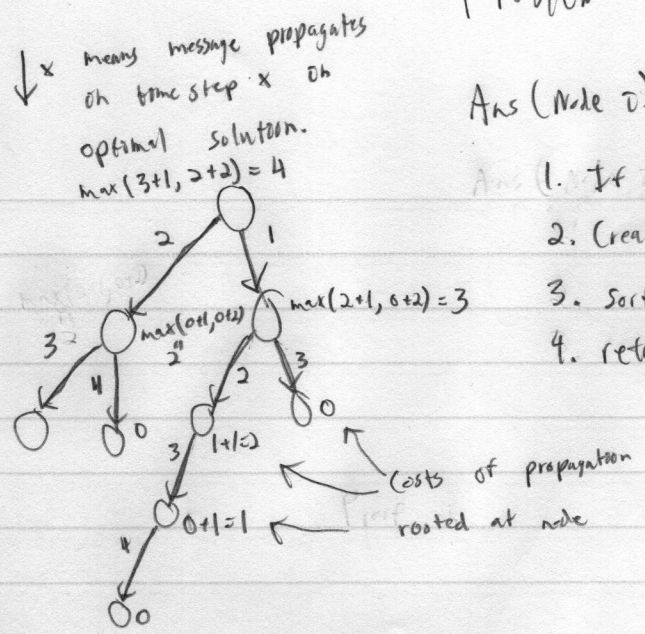
Correctness then follows by an inductive argument on |S|. Clearly

Proof of Runtime: the sizes of L and R are at most |S|/2, so we have recursion tree:



Note that each level is O(n) since medians can be found in O(n) time, and all other steps require only a linear scan. Thus the complexity is O(n log n).

Problem 3



Ans (Node i)

1. If i has no children, return 0;
2. Create array $A = \{Ans(child_1(i)), Ans(child_2(i)), \ldots\}$
3. Sort A in decreasing order
4. return $\max(A_1+1, A_2+2, \ldots)$

Proof of Correctness:

We argue correctness via induction on the size of the subtree.

Base Case: Tree is a single node. No communication is necessary, so returning 0 on Step 1 is correct.

Induction Step: Assuming Ans() returns correctly for trees of size $< n$, we show it returns correctly for trees of size n. Take a rooted subtree at node i of size n. Then all its children have size less than n, so we can take array A as correct via our inductive hypothesis. Now, whichever child node i chooses to send the message to first can propagate the message as node i sends the message to its other children. This holds with the second and third, etc. child we send the message to, but at each step we take an extra unit of time to physically send the messages. Propagation from node i is complete when all i's children have completed propagation, or equivalently when the last propagating path has completed. Thus if C_1 is the time the first child of i to receive the message takes to complete, and likewise for C_2, etc. Then the answer at node i is

$\max(C_1+1, C_2+2, \ldots)$

Now we must show sending to the child that takes longest to propagate first, second longest second, etc., is optimal. This follows from an exchange argument as on HW 4. Take 1, 2, 3, ... to be ordering routes of the divisors and $C_1, C_2, C_3, \ldots$ to be ordered routes. The max of $C_i + i$ is minimized when C is in descending order. Thus steps 3 and 4 yield the minimum propagation time from node i and the induction is complete.

Proof of Runtime:

We can argue runtime via induction on the size of the tree. Our claim is that for a tree of size n, the runtime is at most $O(n \log n)$.

Base Case: Tree is of size 1, returning at Step 1. in $O(1)$ time.

Inductive Step: Suppose node i has $m < n$ children with subtrees of size s_j, with $1 \leq j \leq m$. We must have $s_j < n$, so by our inductive hypothesis, each subtree takes $O(s_j \log s_j)$ time. Then Step 2. takes time:

$$O(s_1 \log s_1 + s_2 \log s_2 + \ldots + s_m \log s_m)$$

which is less than;

$$O(s_1 \log n + s_2 \log n + \ldots + s_m \log n)$$

which equals;

$$O((s_1 + s_2 + \ldots + s_m) \log s_m)$$

Which, since $\sum_{i=1}^{m} s_i = n-1$, is time:

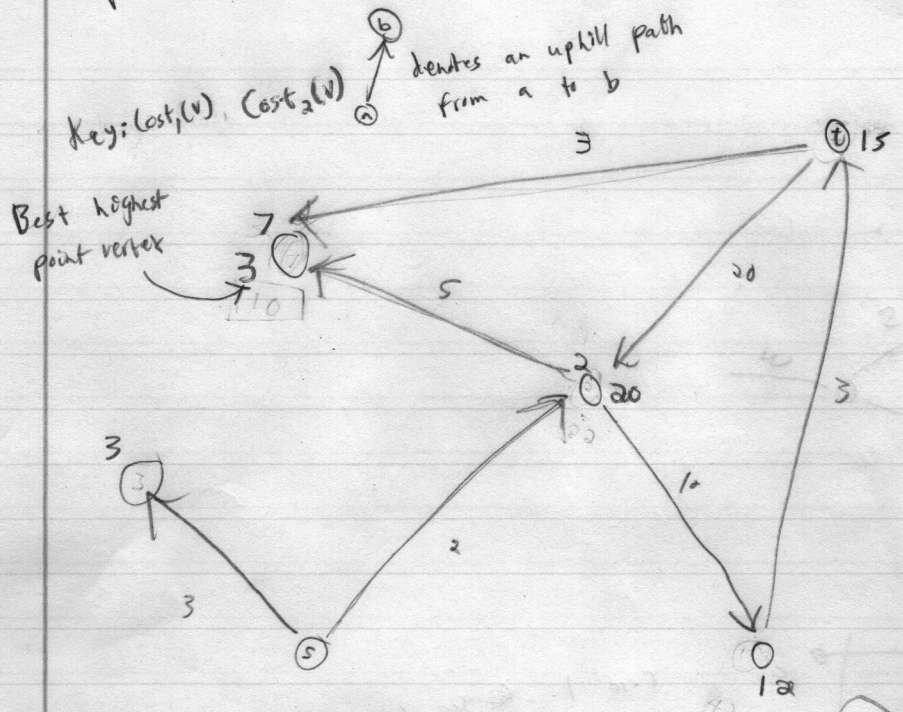
$$O(n \log n).$$

Therefore Step 2 takes at most $O(n \log n)$ time.
Now Step 3 clearly takes $O(m \log m)$ time, which, since $m < n$, is less than $O(n \log n)$ time. Finally, Step 4. takes $O(m)$ time, which is again less than $O(m)$ time.

So in total, the algorithm takes $O(n \log n + n \log n + n) = O(n \log n)$ time. Thus we have shown the inductive step.

Then we have that Ans() takes at most $O(n \log n)$ time.

Problem 4 Approach 1



Algorithm:

1. Find shortest uphill paths sourced at s, denote cost cost$_1$(v)
2. Find shortest uphill paths sourced at t, denote cost cost$_2$(v)
3. Find best highest intersection v, the one with minimum cost$_1$(v) + cost$_2$(v), and return cost$_1$(v) + cost$_2$(v).

Proof of Correctness:

Observe that in any shortest path from s to t composed of an uphill segment and downhill segment, there must be a highest point v. Further observe that if this is a shortest path from s to t then the path from s to v and from v to t also must be shortest. Thus, we can first find all shortest paths going uphill starting at s (and going to every reachable vertex), and all shortest paths going downhill from every vertex to t to find best highest pt v. Naively implemented, this second step would take O(V) many shortest path calculations. However, for any shortest downward path from a vertex u to t, it is also a shortest upward path from t to u. Thus in one shortest path computation going uphill from t we have computed the shortest downhill path from every vertex to t.

Once we have this information, from steps 1. and 2. of the algorithm, we must find the best uphill intersection. This can be done by finding the vertex v with minimum cost$_1$(v) + cost$_2$(v). Thus step 3 computes this information and the algorithm is correct.

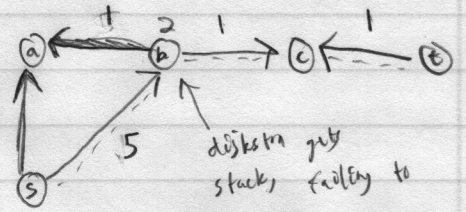
Proof of Runtime:

The algorithm runs in O(V+E) time, where V = # vertices and E = # edges.

Since shortest uphill paths cannot be cyclic, we are looking for shortest paths in a DAG, solvable in O(V+E) time using DP. Then step 1. and 2. take O(V+E) time, and 3. takes O(V) time, so the whole algorithm takes O(V+E) time.

Problem 4 Approach 2

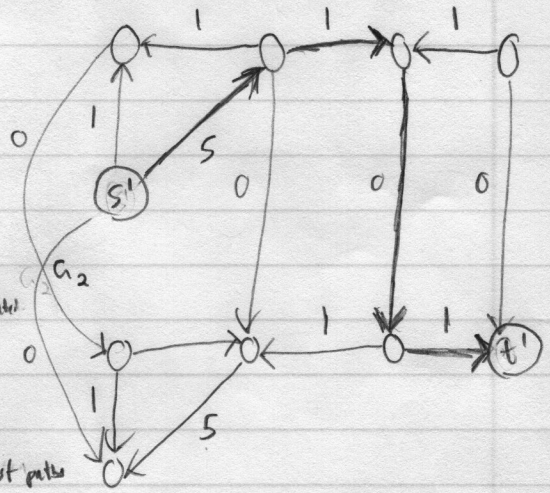
Counterexample to dijkstra w/ keeping track of edge direction



Many of you may think of modifying Dijkstra to keep track of if a downhill edge has been used. Naively implemented, this algorithm is not correct, similar to how Dijkstra fails on graphs with edge weights that increase with the # of edges traversed. (HW 4 review problem 1) A counterexample is shown to the right.

In general it is not a good idea to try to modify algorithms discussed in class in this way. Rather, it is much easier to show correctness (and to be correct) by using in class algorithms as a black box. (similar to unweighted median on HW3).

The fix for dijkstra on this graph is simply to transform the graph, and then run black-box shortest paths on the new graph. Specifically, we need the ability to visit each node twice.



Algorithm:

1. Create a graph G_1 with directed paths showing an uphill path.
2. Create graph G_2, a duplicate of G_1. Reverse all edges in G_2. Then add directed edges from a vertex in G_1 to its copy in G_2 of cost 0. Let s' be s in G_1 and t' be t in G_2.
3. Return the shortest path using directed edges from s' to t'

Proof of Correctness: While we are on G_1, we traverse only edges that are uphill. As soon as we take an edge from G_1 to G_2, we will never take an edge back to G_1, since edges from G_1 to G_2 are directed towards G_2. Once in G_2, the reversed edge directions represent downhill paths in the original graph. Thus any path consisting of uphill then downhill segments can be realized by a path starting in G_1 and ending in G_2, and vice-versa. Since we put t' in G_2, it follows that the shortest path from s' to t' in the new graph is the shortest path from s to t consisting of uphill then downhill segments.

Proof of Runtime:

In G_1, every path is uphill, so we can never cycle. In G_2, every path is downhill so we can never cycle. Moving from G_1 to G_2 can only happen once, so no cycles can arise by moving between G_1 and G_2. It follows that the new graph is acyclic.

Since the new graph is acyclic, we can compute the shortest path from s' to t' in $O(V'+E')$ time using dynamic programming. Note that the V' and E' here represent the number of vertices and edges in the new graph we constructed. Copying the graph increases the number of vertices and edges by a factor of 2. Adding an edge from every vertex on G_1 to its copy in G_2 increases the number of edges by the number of vertices in the original graph. Thus we have:

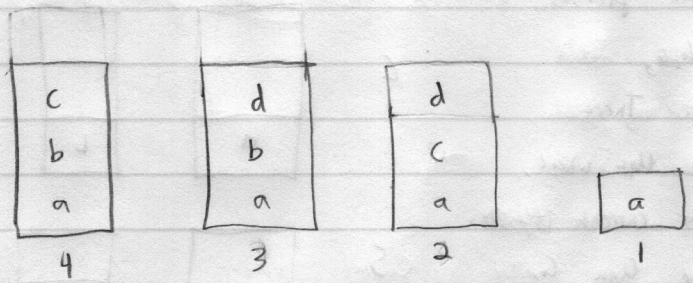
$$V' = 2V \qquad E' = 2E + V$$

Thus the time complexity in terms of the original graph is

$$O(V'+E') = O(2V + 2E + V) = O(V+E).$$

Problem 5

Tables:



Teams:

4	2	2	2
a	b	c	d

Algorithm:

1. Sort teams on decreasing order of number of athletes
2. Add tables to BST keyed on table size, ordered decreasing
3. Processing teams from largest to smallest, pair teams with tables with maximum available space, and then update the BST to account for used seats
4. If it is impossible to complete 3. because there are not enough vacant seats, then no seating arrangement is possible.

Proof of Correctness:

We can prove correctness of the algorithm with a greedy stays ahead argument on the following sense: In any strategy S that has allocated seats to the first i teams (ordered from largest to smallest) G has a larger or equal number of tables with seats still available than S.

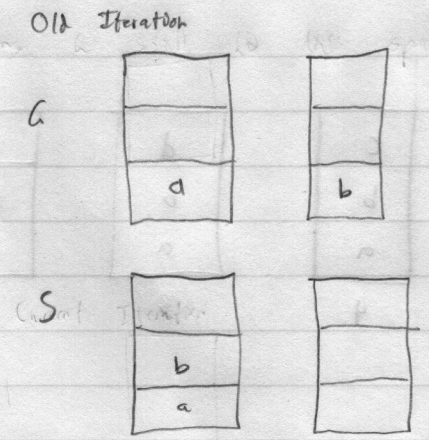
In particular, this means that if G is unable to find seats for team i at iteration i, then no strategy can.

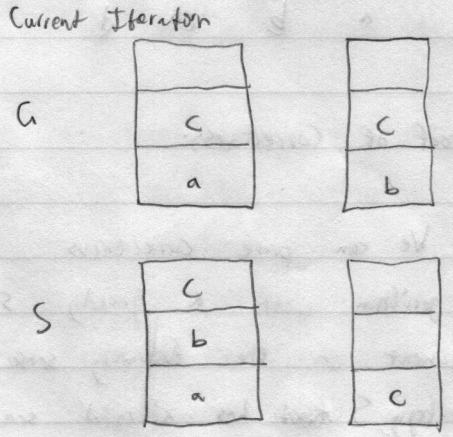
We first notice that in any matching of teams to tables (present in any complete solution), we may decompose the matchings team by team, starting with the largest team, so the ordering imposed on the greedy stays ahead argument can be applied to all strategies S without loss of generality.

Base Case: If $i = 0$, no teams have been paired by G or any S, so the claim follows trivially.

Inductive Step: By the induction hypothesis, G has at least as many tables available at time $i-1$ than any strategy S. Now consider the effect of allocating the s_i people from team i to the s_i tables with the most available space of G. If a strategy S can have more tables with available space after allocating team i, this would imply S's tables with the most available space had more available space than G's. Then by choosing to place teams at tables with less space in a previous iteration, S somehow got ahead by the current iteration. But

wherever S placed the terms in the previous iteration, G still has this space available. Furthermore, S's placement has an imbalance. There are fewer seats available where S placed than where G placed. This implies that at the current iteration, S_{i+1} could not have more available space than G. Therefore S can have no more available tables than G at iteration i and the claim follows.





Proof of Running Time:

In each iteration i, G must make s_i updates on the BST. Each of these updates takes $O(\log m)$ time. Thus over all iterations, G takes time:

$$O(s_1 \log m + s_2 \log m + \ldots + s_n \log m)$$

$$= O\left(\sum_{i=1}^{n} s_i \log m\right)$$

Which is less than or equal to

$$O\left(\sum_{i=1}^{n} s_i \log \sum_{j=1}^{m} t_j\right)$$

Since each table has capacity at least 1. Then this is less than

$$O\left(\left(\sum_{i=1}^{n} s_i + \sum_{j=1}^{m} t_j\right) \log \left(\sum_{i=1}^{n} s_i + \sum_{j=1}^{m} t_j\right)\right)$$

$$= O(N \log N) \quad \text{where} \quad N = \sum_{i=1}^{n} s_i + \sum_{j=1}^{m} t_j.$$

Actions

My proof for problem 6 in Mid Practice

Suppose we have two MST, T_a and T_b. We sort the edge in T_a and T_b by their weight as following:
$a_1, a_2, \ldots, a_n$
$b_1, b_2, \ldots, b_n$
Suppose a_i is the first edge that not equal to b_i. Assume that $w(a_i) \geq w(b_i)$.
There are two possible cases:

If T_a contains b_i, then this means $\exists a_j = b_i$, since a_i is the first edge that not equal to b_i, we know that $j > i$.
Since $a_1, \ldots, a_n$ are sorted by their weight, we have
$w(a_j) \geq w(a_i) \geq w(b_i) = w(a_j)$
This means $w(a_j) = w(a_i)$, so can just exchange a_i, a_j in the sequence.

If T_a does not contain b_i, then we add b_i to T_a. There will be a circle, since we add an edge to a MST. By the property of MST, each edge in this circle has weight less or equal to b_i. Also, there exist a_j in that circle which is not in T_b. So we have $w(a_j) \leq w(b_i)$. Since a_i is the first edge that not equal to b_i, we know that $j > i$.
This means $w(a_j) \leq w(b_i) \leq w(a_i) \leq w(a_j)$. Then we have $w(a_i) = w(a_j) = w(b_i)$.

We can add b_i in T_a and remove a_j from T_a, this won't change the value of edge weight and also maintain T_a to be a MST.

By doing the above cases again and again, we can get the desired property. Any MST has the same weight sequence up to reordering.