CS2040 Midterm 2

Section A

1. There cannot be a tree of size > 1, such that it can both represent a min heap (i.e satisfy the min heap property) and also represent a BST (i.e satisfy the BST property).

i. True

ii. *False

False. You can have a tree of size 2 one is root the other is left child where both items are the same value, so it will satisfy both min heap and BST property.

2. We can use a 0-based array instead of a 1-based array to store a binary heap by modifying the navigation operations as follows:

parent(i) = floor((i+1)/2)-1, except for i = 0

left(i) = i*2+1, no left child when left(i) > heapsize-1

right(i) = i*2+2, no right child when right(i) > heapsize-1

However, this 0-based array will most likely not be used since the computation of the navigation operations have a larger constant compared to those for a 1-based array

i. *True because the modification is correct and the reasoning for not using it is also correct

ii. False because the modification is correct but the reasoning for not using it is wrong

iii. False because the modification is wrong even though the reasoning for not using it is correct

iv. False because the modification and the reasoning for not using it are both wrong

The modification will still work in terms of navigating the heap but it has more arithmetic operations compared to the original way of navigating the heap thus resulting in a larger constant for the navigation operations, and will most likely not be used.

3. If a given BST is a complete binary tree, then it is also a height balanced tree

i.* True

ii. False

True. since only the last level is not filled, so absolute difference in height between left and right subtree of any node is at most 1.

4. Given 2 AVL trees of size N and M respectively, in order to merge them into 1 AVL tree, the best algorithm will require the following time complexity in the worst case

i. O(1) time

ii. O((N+M)*log(N+M)) time

iii. O(NlogN+MlogM) time

iv. *O(N+M) time

v. O(min(N,M)*log(max(M,N))) time

Perform in-order traversal of the two tree to get the values in each tree in sorted order and store into 2 arrays. O(N+M) time.

Merge the 2 arrays in O(M+N) time.

Now that we have a sorted array, we can recursively create the AVL tree by starting from middle of the sorted array, and inserting that element as the root, then recursively insert middle element of the left partition as the left child and the middle element of right partition as right child. Taking O(M+N) time, in total taking O(N+M) time.

This is the solution to Q6ii) of 2017_18_S1_WQ1.

5. There is no input of size N that can allow an AVL tree to be built in O(N) time

i. True

ii. *False

Again based on Q4, if the input already sorted we can build the AVL tree in O(N) time.

6. For an undirected graph G with V vertices and E edges stored in an adjacency matrix, checking if a vertex **v** is reachable from another vertex **u** in G using DFS will in the worst case take

i. O(V+E) time

ii. O(1) time

iii. *O(V^2) time

iv. O(VlogV) time

Note this is using adjacency matrix so even if we convert from adjacency matrix to adjacency list and run DFS in O(V+E) time, the conversion will already take O(V^2) time, thus total time is O(V^2).

7. Running Bellman Ford on a graph stored in an edge list and running it on a graph stored in an adjacency matrix will both still result in O(VE) time complexity

i. True

ii. *False

Again even if we convert from adjacency matrix to edge list/adjaceny list, it will already take O(V^2) time then running Bellman For will take O(VE) so total is O(V^2+VE) and not O(VE).

8. If we run Bellman Ford, original Dijkstra and modified Dijkstra on graphs (where number of vertices V >= 2) with possibly negative edge weights but no negative cycles then

1.) Original Dijkstra will terminate but give the wrong result on some graphs

2.) Bellman Ford will terminate and give the correct result

3.) Original Dijkstra will terminate and give the right result

4.) Modified Dijkstra will not terminate on some graphs

5.) Modified Dijkstra will terminate and give the right result

6.) Bellman Ford will terminate but give the wrong result on some graphs

i. 1,2,4 is true

ii. *1,2,5 is true

iii. 3,4,6 is true

iv. 2,3,4 is true

Refer to lecture notes for answer.

9. For modified Dijkstra's algorithm, if we replace the Priority Queue with a normal queue and then perform modified Dijkstra's as per normal as shown below,

```
initSSSP(s)
Q.enqueue((0, s))  // Q is a normal queue
while Q is not empty
  (d, u) = Q.dequeue()
  if d == D[u]
    for each vertex v adjacent to u
      if D[v] > D[u] + w(u, v)
        D[v] = D[u] + w(u, v)
        p[v] = u
        Q.enqueue((D[v], v))
```

the algorithm will still work correctly for positive weighted graphs

i. *True for all cases

ii. False for all cases

iii. True only for some cases

True because we allow dequeued vertex to be re-enqueued, it does not matter then that when a vertex is de-queued it's SP is not yet found. Ultimately, once all SP is found, no relaxation can be done and thus no vertex will be re-enqueued and Q will become empty and thus the algorithm will stop. It can be very inefficient though.

10. Given any weighted graph with > 1 vertex, if the shortest simple path between every pair of vertex is also the longest simple path between those pair of vertices, then the graph must be a tree.

i. True

ii. *False

False. Consider a circular linked-list like graph.

11. Given a connected weighted graph that has a unique MST consisting of some set of edges. If every edge in the original graph has its weight increased by a constant K, running Prim's/Kruskal's algorithm on the updated graph results in the same set of edges being selected in the new MST.

i. *True

ii. False

True, because all possible spanning tree of the graph will have their cost increased by (V-1)*K. Thus the original MST will still be MST among the spanning trees of this new graph.

12. Assuming a graph is stored in all 3 graph data structures, Topological Sort using Kahn's algorithm or modified DFS algorithm on the graph cannot be optimized to go faster than O(V+E) for any valid DAG

i. *True

ii. False

True. Even in the case where E = 0, we can detect that this is a totally disconnected graph in O(1) time from the edge list size being 0, but we still need to list down a topological ordering which is O(V) which is still O(V+E) since E is 0.

13. For traversal of an undirected graph, if we do not keep track of visited vertices or the predecessor of each visited vertex for DFS and BFS, both algorithms will never terminate (assuming infinite memory)

i. *True

ii. False

True. This means if there are cycles, we can keep cycling forever. Even if no cycle, we can moving between the predecessor of a vertex and the vertex itself indefinitely.

14. The largest integer in a non-empty min heap containing non-unique integers can possibly be the root of the min heap.

i. *True

ii. False

True. A min-heap with 1 item.

15. There is no way to merge a max heap and a min heap into a max heap in < O(NlogN) time.

i. True

ii. *False

False. We can simply copy both into a new array and perform fast heap create in O(N) time.

16. Given a UFDS initialized with n disjoint sets, in order to end up with one set having the maximum height by calling a combination of UnionsSet(i,j) and findSet(i) operations, we can only call UnionSet(i,j) and findSet(i) where i and j are the representative items of their respective set and not on any other items.

i. True

ii. *False

False. you can call UnionSet and FindSet on i and j where i and j are children of the representative item (root), since they are already attached to the representative item, it does not matter.

17. Without using any other additional data structure except for another min or max heap, the best algorithm for finding the K^{th} smallest item in a min heap with N items will take

i. O(KlogN) time

ii. O((K+N)*logN) time

iii. *O(KlogK) time

iv. O(NlogN) time

v. O(KlogK+N) time

The easiest is to simply keep call extractMin *k* times. At the k^{th} time, we will have the k^{th} smallest element. Time is $O(k \lg n)$.

Algorithm idea for $O(k \lg k)$ solution:

The current smallest element is at the root of the heap. The 2^{nd} smallest element will be in the set {root.left, root.right}. Assume it's root.left, then to get the 3^{rd} smallest element we need to compare among items in the set {root.left.left, root.left.right, root.right} as these 3 are the only possible candidate for the 3rd smallest element. A similar argument exist if root.right is the 2nd smallest element. As you can see after getting the 3^{rd} smallest element, we remove it from the set and simply add its left and right child to the set and determine the 4^{th} smallest from there. This pattern continues until we can find the k^{th} smallest element.

In order to efficiently find the i^{th} smallest element, we need to insert the items that are in the set of possibilities into another heap (a secondary heap), so that extractMin is efficient. Let the original heap be oh and secondary heap be sh. The algorithm is then as follows

```
Repeat k times
  If (1st time)
    Insert root of oh into sh  // insert an object that stores both node index and node key, but
                               //   will use only node key for shiftUp and shiftDown purposes
    i = sh.extractMin()
  else
    Insert node at oh.left(i.index) into sh if oh.left(i.index) < heapSize
    Insert node at oh.right(i.index) into sh if oh.right(i.index) < heapSize
    i = sh.extractMin()
 return i.key
```

Now the secondary heap can only contain at most *k* vertices (add 2 vertices via left and right child and remove 1 vertex via extractMin each time we get the next smallest element, so in all add 1

vertex into the secondary heap each time) when we find the k^{th} smallest element.

Now since we don't do extractMin on the original heap, but simply traverse it expanding vertices to be added to the secondary heap as directed by the current minimum vertex from the secondary heap, we only need to visit at most 2*k vertices in the original heap. Since the secondary heap has at most k vertices at any time, extractMin and insert only cost $O(\lg k)$ and we perform 1 extractMin and 2 insert per iteration. So total time complexity is $O(2 * k) + O(3*k * \lg k)$, which is $O(k \lg k)$.

18. Finding rank of a value in a BST will take at most O(logN).

i. True

ii. *False

Note it says BST not bBST or AVL. In the worst case the BST can be entirely skewed to one side so total time to find rank if O(N) not O(logN).

19. To find the largest weighted edge along a minimax path (the path that minimizes the largest weighted edge) from a vertex **s** to a vertex **d** in an undirected connected weighted graph with more than 1 vertex, all we need to do is run Prim's algorithm using **s** as the source vertex and keep track of the largest edge added to the MST as the algorithm is run. Once vertex **d** has been added to the MST, terminate Prim's then return the weight of the largest edge found so far.

i. *This algorithm is always correct

ii. This algorithm is always wrong

iii. This algorithm is correct only for some input graphs

This is correct. By the time d is reached there must be a path from s to d. Now the largest edge weight W found so far must be along the path from s to d. Now suppose W belong to an incoming edge to a node N that is not along the path from s to d. That means that according to the algorithm all edges along the path from s to d do not have weight > W, so the edges and the vertex they linke to in the path must have been enqueued and dequeued before W. This is a contradiction that W is already dequeued (otherwise we wouldn't have tracked W).

20. A directed graph with V vertices and >= V out-going edges cannot be a DAG

i. True for all cases

ii. *False for all cases

False. 3 vertices 1,2,3 connected as such (1,2), (1,3), (2,3) forms a DAG with 3 edges.

21. To compute the topological ordering of a DAG with >= 1 vertex using the modified DFS algorithm, we can start from any vertex **v**, not necessarily those with no in-coming edges

i. *True for all cases

ii. False for all cases

iii. True only for some cases

True. This is because

1.) we call DFS on all vertices.

2.) Even if we start from a vertex V with in-coming edges, we will still ultimately call DFS on vertices which will reach V. Now in the toposort array those vertices will be placed after V, since we reverse the array in the end, those vertices will be before V which is the correct topological ordering.

22. The only way for a connected weighted graph to have a unique MST is for all its edge weights to be unique.

i. True

ii. *False

False. A graph which is already a tree will be its own MST regardless of what are the edge weights.

23. When running Prim's algorithm on a connected weighted graph G, no edge in G will be enqueued in the priority queue more than once

i. *True

ii. False

True, since each vertex is only processed once (enqueue all its outgoing edges), then it will be set to be taken, so that it will never be processed again, that means all edges are only enqueued once.

24. Based on the cut property, we can generate the MST for any given connected weighted graph G with more than 1 vertex by the following algorithm.

Iterate through each vertex **v** (in no particular order) in G and create a cut with **v** in one partition and the rest of the vertices in the other partition. Now just pick the smallest edge (**v**,**u**) crossing the cut (if there are multiple smallest edges just pick any one of them) and include it in the MST. If vertex **v** and vertex **u** are both already included in the MST ignore (**v**,**u**) (since it will cause a cycle) and move on to the next vertex. At the end of this algorithm, the edges chosen will form the MST

i. It is true for all cases and the time complexity is < O(ElogV) in the worst case

ii. It is true for all cases and the time complexity is still O(ElogV) in the worst case

iii. *It is not true for all cases and the time complexity is < O(ElogV) in the worst case

iv. It is not true for all cases and the time complexity is still O(ElogV) in the worst case

This is not a correct algorithm. For example if we run this algorithm on the example graph CP4.10 in the MST module in Visualgo, and we process the vertices in the order from 0,1,2,3,4. When we process 0 we will choose (0,1) of weight 4, then we process 1 choosing (1,2) of weight 2, then we process 2 now only choosing (2,3) of

weight 8 since the other edges lead to vertices already added to the MST. Next we process 3 choosing (3,4) of weight 9 again because the other edges lead to vertices already added to the MST. The total cost is 23 but the actual MST only cost 18.

This is because we do not put all vertices already added to the MST into a partition, rather we simply place each vertex in a partition and all the other vertices regardless of whether they are already added to the MST or not into the other partition resulting in a wrong algorithm.

25. Implementing a new type of Priority Queue which allows insert, extractMax, extractMin, getMax (just return the maximum item without removing it) and getMin (just return the minimum item without removing it) can be done using an AVL tree and a few extra variables such that

i. All the listed operations run in O(N) time in the worst case

ii. All the listed operations run in O(logN) time in the worst case

iii. {insert,extractMax,extractMin} = O(N) time in worst case, {getMin,getMax} = O(logN) time in worst case

iv. *{insert,extractMax,extractMin} = O(logN) time in worst case, {getMin,getMax} = O(1) time in worst case

v. all the listed operations run in O(1) time in the worst case

We can perform insertion in the PQ as insertion in the AVL taking O(logN) time. Now we can keep 2 variable which keep track of the current largest and smallest value in the AVL. For every insertion and deletion from the AVL we can update the 2 variable in O(logN) time. So it doesn't change the time complexity of insertion and deletion in the AVL.

getMin and getMax can then be O(1) time as we simply need to return the value of the 2 variables.

extractMin and extractMax will still take O(logN) time to perform by calling delete(getMax()) and delete(getMin) and then to get the new min and new max we simply call findMin and findMax again, still taking time O(logN), thus extractMin and extractMax is still O(logN).

26. Given a possibly cyclic graph and a randomly shuffled edge list, it is possible for bellman ford to produce the correct answer for the shortest path distances from the source vertex to all other vertices in just one pass.

i. *True

ii. False

True if by chance we get the correct edge ordering by shuffling. Although in general this is hard to determine.

27. For a directed weighted graph G with V vertices and E edges where edge weights are only from the set {K,2K,3K,4K,5K} for some positive integer value K, the best algorithm for finding the shortest path from a source vertex **s** and a destination vertex **d** in G will take

i. O(1) time in the worst case

ii. *O(V+E) time in the worst case

iii. O((V+E)logV) time in the worst case

iv. O(VlogV) time in the worst case

v. O(VE) time in the worst case

Since the edge weights are multiples of K, any path cost must be divisible by K. Which means if we divide the path cost of all possible paths from s to d the SP cost will still be the smallest. So we can simply transform the graph into one where the edge weights are divided by K. Now in this graph all edge weights are just 1 to 5. We can now further transform this graph by replacing each edge of cost K' (which is between 1 to 5) by a series of K' edges and K'-1 vertices linked in linked list like fashion, so that each edge is of cost 1. Now in this graph the number of vertices will be O(5V) = O(V) and the number of edges will be O(5E) = O(E) so a small constant blowup in the size from the original graph. Since the graph is now all edge weight 1, we can run BFS to find the SP from s to d in O(V+E) time.

28. You are given a special kind of connected graph with 3N+1 vertices for N >= 1 where the vertices are connected as follows

for i = 0 to N-1
  vertex 3i has an undirected edge to 3i+1
  vertex 3i has an undirected edge to 3i+2
  vertex 3i+1 has an undirected edge to 3i+3
  vertex 3i+2 has an undirected edge to 3i+3

In this graph each vertex is given a positive weight.

Your goal is to find the smallest vertex by weight which when removed will disconnect the graph (this kind of vertex is also known as an articulation point or cut vertex).

The best algorithm to solve this will take

i. O(1) time in the worst case

ii. $O(N^2)$ time in the worst case

iii. *O(N) time in the worst case

iv. O(NlogN) time in the worst case

In this special graph only vertices that are multiple of 3 are possibly articulation points (except for 0 and 3N). So just do a linear scan of those vertices and return the smallest weighted vertex among them. This takes O(N) time.

29. You have a near complete graph with N vertices (N >= 7) and with up to 10 missing edges in the graph. In order to find if any vertex **v** is reachable from any other vertex **u** in the graph, the best algorithm will need

i. $O(N^2)$ space to represent the graph in order to answer the query in O(N) time in the worst case

ii. O(N) space to represent the graph in order to answer the query in O(N) time in the worst case

iii. O(1) space to represent the graph in order to answer the query in O(N) time in the worst case

iv. *O(1) space to represent the graph in order to answer the query in O(1) time in the worst case

This is essentially question C.1 from 2015-16-S1-WQ2. Instead of storing the near complete graph itself, we store the complement of the graph, that is, only the edges which are missing. Since only up to 10 edges removed regardless of the size of the graph, the space required is at most O(1).

Now if there are >= 10 vertices in the graph, removing only 10 edges cannot disconnect the graph so for such graphs always return true.

For graphs with <= 10 vertices, we run DFS/BFS reachability test where we start from u. For each vertex we are currently at, its neighbours are those vertices not listed in the complement edge list. We can find such

vertices in O(K) time where K is the size of the graph. So total time taken for the reachability test is $O(K^2)$. Since K <= 10 this is bounded by O(100) regardless of the size of the graph, thus this is still O(1) time.

30. Given a weighted connected graph with N nodes and M edges and all edge weights are unique, find the path from node 1 to node N that minimizes the **k**th largest edge along the path.

If there are no paths with >= **k** edges from 1 to N, then any path from 1 to N will do.

For example, if we have edges of weight 10,2,3,5,8 in a path and **k** = 4, then the **k**th largest edge will have weight 3 in the path. The most efficient algorithm to find such a path from node 1 to node N is:

i. Use Dijkstra's Algorithm in some way, resulting in $O(M\log^2 N)$ time in the worst case

ii. Use Prim's Algorithm in some way, resulting in O(MlogN) time in the worst case

iii. Use BFS in some way, resulting in O(N + M) time in the worst case

iv. *Use BFS in some way, resulting in O(MlogN) time in the worst case

v. Use DFS in some way, resulting in O(N + M) time in the worst case

This is a trap question. Also a hard question. Most will think this is a minimax problem so it can be solved using Prim's in some way. However this is not the case, as Prim's/Kruskal's only minimizes the maximum edge in O(MlogN) and not the Kth largest edge along the path from 1 to node N.

To solve this, we need to do a form of binary search for SP from node 1 to N. Go through the edge list and record down all the edge weights in an array A. Sort A in O(MlogM) time which is O(MlogN) since M is between O(N) and $O(N^2)$.

1. Let L = 0 and R = N-1, and P = A[(L+R)/2] (middle value) make a copy of the original graph and go through all edges in the graph setting all edges with weight >= M to 1 and all edges < P to 0. Let curBestPath = ""

2. Now run BFS SSSP from node 1. Use a double-ended queue instead of a normal queue. In this queue all the 0 weight edge comes before the 1 weight edge. For each node we visit, we enqueue all 0 weight neighbor (neighbour linked by a 0 weight edge) to the front of the queue and all 1 weight neighbor (neighbour linked by 1 weight edge) to the back of the queue. This doesn't change the time complexity of BFS.

3.

i. Now if SP cost from node 1 to node N is < K, that means there are less then K edges along any path from 1 to N with edge weight >= M. This means P is too large a value for the kth largest edge from 1 to N.

If (L == R) and (curBestPath == "")

There are no paths from node 1 to node N with >= K edges, so simply perform DFS from 1 to N and output the path used.

Else if (L == R) and (curBestPath != "")

return curBestPath

Else if (L < R)

set R = (L+R)/2, P = A[(L+R)/2], make a new copy of the graph and set all edges < P to 0 and those >= p to 1 and repeat step 2.

ii. Now if SP cost from node 1 to node N > K, that means that the Kth largest edge along any path from 1 to N must be > P (unique edge weights), which means minimized kth largest edge must also be > P.

If (L == R) and (curBestPath == "")

There are no paths from node 1 to node N with >= K edges, so simply perform DFS from 1 to N and output the path used.

Else if (L == R) and (curBestPath != "")

return curBestPath

Else if (L < R)

set L = (L+R)/2, P = A[(L+R)/2], make a new copy of the graph and set all edges < P to 0 and those >= p to 1 and repeat step 2.

iii.Now if SP cost from node 1 to node N == K, find path from node 1 to N using the predecessor array and save to curBestPath. --> O(N) time.

If (L == R) return curBestPath

else

Set R = (L+R)/2, P = A[(L+R)/2] and make a new copy of the graph and set all edges < P to 0 and those >= p to 1 and repeat step 2. // try with smaller value

Now step 2 and 3 can only be run at most O(logM) time since we keep halving A. step 2 can be done in O(M+N) time, step 3 can be done in O(N+M) time too so in total together with the initial sorting of the edges, this whole algorithm is O(2*(M+N)logM+NlogM) = O((M+N)logM) = O(MlogM+NlogM) = O(MlogN+NlogN) = O(MlogN) since M ranges from O(N) to $O(N^2)$ in a connected graph.

Section B

More Binary Heap Operations

For the following 4 questions, all the heaps store integer values and are of size N unless otherwise stated.

31. Given a max heap A of size N and a max heap B of size M. If we want to form a new max heap C out of all items in A and B > K, the best algorithm to do this has worst case time complexity

i. O(N'logN+M'logM) where N' are the number of items in A > K and M' are the number of items in B > K

ii. O(N+M)

iii. O((N+M)*log(N+M))

iv. *O(N'+M') where N' are the number of items in A > K and M' are the number of items in B > K

We can get all items > K in A in O(N') time where N' is the number of items in A > K (Tutorial 5 Q2). Similarly, we can get all items > K in B in O(M') time where M' is the number of items in B > K. Then to build the new heap, use fast heap create which takes O(N'+M') time. So total time taken is O(M'+N')

32. Given a 1-based array A representing a min heap, the fastest way to update the last K (1 <= K <= N) items in A by a positive value L can be done in

i. O(logN) time in the worst case

ii. O(KlogK) time in the worst case

iii. O(KlogN) time in the worst case

iv. O(N) time in the worst case

v. *O(K) time in the worst case

For each item in the last K items of A, if it has a parent which is among the last K items, updating both by positive value L does not change their ordering. If it has a parent which is outside the last K items, updating the item by L will still not change their ordering since its parent is already <= to it. Thus updating the last K items is simply going from index N to index N-K+1 and updating the values there without need to re-heapify the heap. This takes O(K) time.

33. Given 2 non-root values and their position (array index) in a max heap, the best algorithm to find the first value in the max heap just bigger than both of them requires in the worst case

i. O(1) time

ii. O(logN) time

iii. *O(N) time

iv. O(NlogN) time

Unlike BST which gives total ordering of the elements, a heap does not do that. Thus to find the value just larger than both of them (if it exists) you have no choice but to scan through the entire array. This takes O(N) time.

34. Given a max heap H and a value i, the best algorithm to re-heapify H after all nodes in the subtree rooted at H[i] is updated by a positive value L and where size of the subtree is > logN will require in the worst case

i. O(logN) time

ii. O(N) time

iii. *O(KlogN) time where K is the number of nodes along the path from H[1] (i.e root of H) to H[i]

iv. O(1) time

v. O(MlogN) time where M is the number of nodes in the subtree rooted by H[i]

Here, the values in the subtree rooted at H[i] already has their values updated. Now within the subtree itself, max heap property is maintained, but with respect to the rest of the heap this may not be true.

So to re-heapify the heap, you can call shiftup for all nodes in the subtree. This will take time O((subtree size)*logN).

However note that the nodes affected by the shiftup is at most all the nodes along the path from H[i] to the root of the max heap (there are at most logN of such nodes). Since it say in the question that size of subtree > logN, the best way is to instead call shiftdown on all the nodes along the path from H[i] to the root. So option iii is the best.

The following 4 questions refer to the same story

35. Due to the problem of a certain disease, cities are experiencing increased demand for certain essential supplies. A factory in a particular city, Factoria, would like to deliver supplies to these cities via a truck-based delivery service. To reduce the odds of contaminating the supplies, it is decided that a box of supplies will be assigned to a single driver only (ie. a box of supplies will not be passed between multiple drivers). However, this may cause a problem, as a driver can only travel a certain number of hours without resting. Furthermore, to facilitate contact tracing, drivers helping to deliver these supplies are asked to only rest in Factoria (so they cannot rest at any other cities).

Given a list of V cities (including Factoria), all of which require supplies, and E roads making up the road network between these cities with the time required to travel each road in hours (which is the same in both directions), determine the minimum amount of time a driver needs to go without rest in order to be able to reach every city. It is guaranteed that all cities are connected (either directly or indirectly) to Factoria.

The best algorithm to determine the above involves

i. Using BFS or DFS graph traversal, taking O(V+E) time in the worst case

ii. Using Prim's or Kruskal's algorithm, taking O(ElogV) time in the worst case

iii. Using Bellman Ford algorithm, taking O(VE) time in the worst case

iv. *Using Dijkstra's Algorithm, taking O((V+E)logV) time in the worst case

Dijkstra from the Factoria node and find SP to all other nodes --> O((V+E)logV) time

Find the one with the largest SP cost, then multiply by 2. This the minimum time required for the driver to drive without resting so that he can reach every other city from Factoria and go back to Factoria. --> O(V) time

Total time taken is O((V+E)logV) time

36. Suppose the restrictions were now relaxed, and drivers are allowed to rest at other cities. The best algorithm to give the minimum amount of time a driver needs to go without rest in order to be able to reach each city will involve

i. Using BFS or DFS graph traversal, taking O(V+E) in the worst case

ii. *Using Prim's or Kruskal's algorithm, taking O(ElogV) in the worst case

iii. Using Bellman Ford's algorithm, taking O(VE) in the worst case

iv. Using Dijkstra's algorithm, taking in O((V+E)log V) in the worst case

Because driver can now rest at any city, the minimum time required is simply a minimax problem which can be solve by performing Prim's/Kruskal's and return the largest edge weight in the MST created. Total time taken is O(ElogV) time.

37. Seeing the difficulty of having to make deliveries to so many cities, an agreement was reached to allow access to a high speed rail network for the purposes of making deliveries.

However, given the way the rail network was designed, it allowed for travel in **one direction** along the rails only. While it was known that if the rail network allowed for travel in both directions, it would allow for Factoria

to deliver supplies to every city, it was not as clear if it still held true if the rail network only supported one direction of travel.

Suppose that there was no need to worry about whether a train would be able to return to the factory after making a delivery.

The best algorithm to give a list of cities (if any) which would not have access to supplies will involve

i. *Using BFS or DFS graph traversal, taking O(V+E) in the worst case

ii. Using Prim's or Kruskal's algorithm, taking O(ElogV) in the worst case

iii. Using Bellman Ford's algorithm, taking O(VE) in the worst case

iv. Using Dijkstra's algorithm, taking in O((V+E)log V) in the worst case

v. Using Kahn's algorithm or modified DFS to perform topological sorting, taking O(V+E) in the worst case

This is a reachability problem. Just run DFS/BFS from Factoria and return all nodes which are not visited. Total time taken is O(V+E).

38. After realizing that there still would be cities that would receive no supplies, the high speed rail network was expanded to include factories in other cities, in the hopes that this would allow for all cities to receive supplies.

Now given this new information (there are K cities now containing factories from which a train can start, including Factoria) , the best algorithm to find out which cities would not have access to supplies will involve

i. *Using BFS or DFS graph traversal, taking O(V+E) in the worst case

ii. Using BFS or DFS graph traversal, taking O(K(V+E)) in the worst case

iii. Using Bellman Ford's algorithm, taking O(VE) in the worst case

iv. Using Dijkstra's algorithm, taking in O((V+E)log V) in the worst case

Still a reachability problem. Just run DFS/BFS from all cities with rail network, but maintain 1 visited array, so already visited vertices are not visited again. Since each vertex is still processed once which also means all edges are processed once, total time taken is still O(V+E)

The following 4 questions refer to the same story

39. Alice and Bob are playing a game on a graph. The graph contains N nodes numbered 1 to N, and M directed edges, where each edge has a non-negative weight.

Initially, Alice begins from node 1, and she wishes to get to node n. It is guaranteed that there is a path connecting node 1 to node N. Alice and Bob repeat rounds of movements until Alice reaches node N. In each round:

Suppose Alice is currently at node u. First, Alice selects one of the outgoing edges from node u, and tells Bob the edge that she has selected. Assume that this edge takes Alice from node u to node v.

Next, Bob takes the set of outgoing edges S from node u, and shuffles the edge weights of S among the edges in S in any way he likes.

Finally, Alice moves from node u to node v, and pays Bob an amount of money corresponding to the weight of the edge.

Alice wonders what is the smallest amount of money she needs to pay Bob in order to reach node N from node 1 if Bob is out to maximize the amount of money she needs to pay.

The best algorithm to answer Alice's question will take

i. O(N+M) time in the worst case

ii. O(NlogN) time in the worst case

iii. O(NM) time in the worst case

iv. *O((M+N)logN) time in the worst case

v. O(MlogN) time in the worst case

Since Bob will always use the max edge weight among the outgoing edges of any node Alice is currently at to be the edge weight of the outgoing edge she chooses, this is similar to just transforming the graph by replacing the edge weights of all edges in S_v for any node V by the maximum edge weight among the edges in S_v. This can be done in O(M) time. Then the minimum Alice has to pay is the cost of the SP from node 1 to N in this transformed graph which can be found by performing Dijkstra's for O((M+N)logN) time. So in total this is O((M+N)logN) time.

40. Alice realizes that she is paying too much money to Bob. She proposes that they change the rules of the game. In the new version of the game, the graph remains the same: there are N nodes numbered 1 to N, and M directed edges, where each edge has a non-negative weight. It is guaranteed that node 1 is connected to node N via a path.

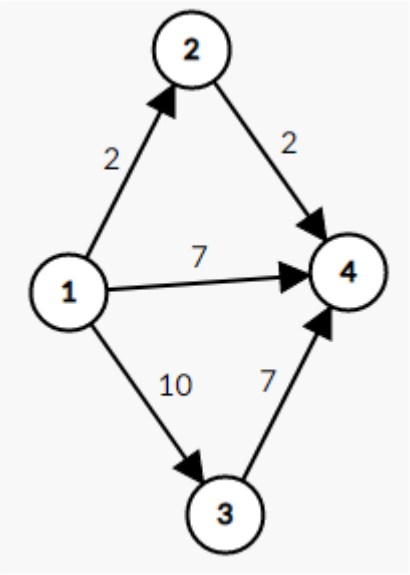
In the new version of the game, Alice still wants to get from node 1 to node N. However, the new version of the game consists of two phases, where Bob performs the first phase once, and Alice performs the second phase once.

In the **first phase**, for each node *u* in the graph, Bob takes the set of outgoing edges S from node *u*, and shuffles the edge weights of S among the edges in S in any way he likes.

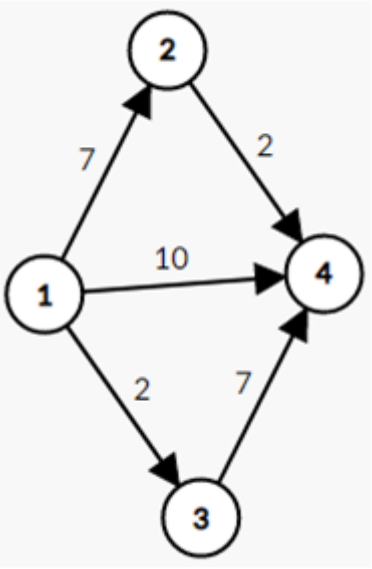
In the **second phase**, Alice begins from node 1 and moves to node *n* via a path of edges. Alice pays Bob an amount of money corresponding to the length of the path.

Alice wonders what is the smallest amount of money she needs to pay Bob in order to reach node N, assuming that Bob tries to make as much money off Alice as possible.

Example: Suppose the initial graph is:



In **phase 1**, to maximize the amount of money Bob can make from Alice, Bob may shuffle the edges such that the following graph is obtained. Notice that the edge weights of the outgoing edges of node 1 have been rearranged.



In **phase 2**, Alice may take either $1 \rightarrow 2 \rightarrow 4$ or $1 \rightarrow 3 \rightarrow 4$ to reach node 4. In either case, she needs to pay Bob $9. This is the smallest amount of money Alice must pay Bob.

Now if the graph satisfies $N > 2$, $M = 2N - 4$, the edge weights are positive integers ranging from 1 to 10000 and the following conditions

1.) There is 1 outgoing edge from node 1 to nodes 2,3, ..., N-1 respectively
2.) There is 1 outgoing edge from node 2,3, ...,N-1 respectively going to node N

The best algorithm to answer Alice's question on the above graph will take

i. O(N!) time in the worst case

ii. *O(N) time in the worst case

iii. O(NlogN) time in the worst case

iv. $O(N^2)$ time in the worst case

v. $O(2^N)$ time in the worst case

In this special graph there is exactly one edge going from node 1 to all other nodes except node N, and there is exactly 1 edge going from each of those nodes to node N.

In this scenario, Bob has to fix the edge weight first before Alice chooses the path to get from node 1 to N, so Alice knows his strategy is to "build" a graph by shuffling the edge weights so that the minimum cost from node 1 to N for Mary is maximized.

To "build" such a graph for this special kind of graph, sort all the outgoing edges from node 1 from largest to smallest (call this sorted list A), then take the outgoing edges from node 2 to nodes N-1 and sort them from smallest to largest (call this sorted list B). This can be done in O(N) time by using radix sort since there are at most O(N) edges for this graph.

Now pair the edges up by going down both lists. for the largest edge in A pair it with the smallest edge in B and so on. This will ensure that the minimum cost from node 1 to Node N will be maximized. This again takes O(N) time. SP is simply the minimum among w(node 1,node x)+w(node x,node N) for x from 2 to N-1, and can be computed while performing the pairing.

Total time for this algorithm is O(N).

41. To answer the Alice's question on a general weighted directed graph where the edge weights are positive integers between 1 and 10000, the best algorithm will involve

i. Generating a graph for each possible permutation of the outgoing edge weight of each vertex and compute SSSP from node 1 to node N on each graph. Pick the minimum SSSP cost among all the graphs generated. This results in a **O(N!)** time algorithm in the worst case

ii. Generating a graph for each possible permutation of the outgoing edge weight of each vertex and compute SSSP from node 1 to node N on each graph. Pick the minimum SSSP cost among all the graphs generated. This results in a **$O(2^N)$** time algorithm in the worst case

iii. Transforming the input graph, and use BFS in some way to solve SSSP by running on node 1 as source. At the end, the shortest path cost of node 1 to node N is the minimum cost required. This results in **O(N+M)** time in the worst case.

iv. Transforming the input graph, and use Dijkstra in some way to solve SSSP by running on node 1 as source. At the end, the shortest path cost of node 1 to node N is the minimum cost required. This results in **O((M+N)logN)** time in the worst case.

v. *Transforming the input graph and using Dijkstra in some other way that does not use node 1 as the source to find the minimum cost required. This results in **O((M+N)logN)** time in the worst case.

This is quite hard.
Again the goal is that Bob want to maximize the cost but Alice wants to minimize it, So the problem is maximizing the minimum that Mary has to pay.

How can Bob maximize the cost in this general graph, since Alice can simply keep choosing the smallest edge at each node she is at, and there can possibly be many ways for Alice to reach node N from node 1?

The way to do it is to start from the end, i.e node N. Look at the set of nodes R with outgoing edges to N. Now Alice must use one of those nodes to reach N, so the best Bob can do is for each node N' in R, pick the largest edge weight among all outgoing edges for N' as the edge weight of the outgoing edge to N.

What next?
Perform Dijkstra but on a transformed graph with the edges inverted from the original graph, and start from node N instead of node 1.
For each node V dequeued from the PQ, it is the node with the current smallest SP estimate, so again for each node R with outgoing edges to V maximize the cost by assigning the largest edge weight that has not yet to assigned to the outgoing edge to V. Note that this is the correct order since you maximize the cost of the nodes in increasing order of their SP estimate (which is maintained by the PQ). Keep doing this until you hit node 1.

The SP from N to 1 is now the SP from 1 to N that is maximized for Bob, and also what Mary has to at least pay to get from 1 to N, since the best she can do is to find the SP from 1 to N in this transformed graph

The detailed algo will be
1.) For each node sort the outgoing edge weights in decreasing order. This can be done in O(M) time using radix sort. Keep a pointer to each node's sorted outgoing edge list
2.) now create a graph where all the edges are inverted (can be done in O(M+N) time)
3.) start from node N, then do Dijkstra from N to 1 -> O((M+N)logN) time
4.) whenever a node V is dequeued from the PQ for each neighbor U (the outgoing edge (V,U) is the outgoing edge (U,V) in the original graph) choose the weight currently pointed to in the sorted outgoing edge list of U, then increment the pointer by one (go down the sorted outgoing edge list). So getting the edge weight for each neighbor is O(1) time and does not change Dijkstra's algo.
5.) after running Dijkstra the SP cost from N to 1 which is the SP cost from 1 to N in the original graph will be maximized which is what Bob wants to achieve and which is what Mary has to pay in order to get from 1 to N.

Total time taken is O((M+N)logN).

42. Now if the graph given is a DAG and the edge weights are positive integers between 1 and 10000, the best algorithm to answer Alice's question will take

i. O(MlogN) time in the worst case

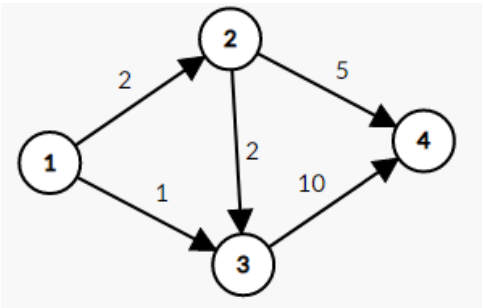
ii. *O((N+M)logN) time in the worst case

iii. O(N!) time in the worst case

iv. $O(2^N)$ time in the worst case

v. O(N+M) time in the worst case

Trap question. Students will think simply topo sort and perform 1-pass bellman. However, topological ordering of the vertices might not be the correct ordering as the objective is to order the vertices by increasing order of SP estimate and topological ordering might not satisfy this objective. An example in given below



In this example, if you invert the edges then topo sort and process the vertices in topological ordering, the SP cost from 1 to 4 will be 6, but the actual maximized SP cost from 1 to 4 is 7.

So you still have to do Dijkstra and the solution does not change from Q41. so total time is still O((M+N)logN).

The following 4 questions refer to the same story

43. John has a bag of N magic balls which are numbered with integer values from 1 to M respectively where N >= 3 and 1 < M <= N^2. These magic balls are all blue. Now John will randomly take out 3 magic balls from his bag and query which is the median magic ball (based on their number) among the 3. Then he will continue randomly picking out 1 magic ball at a time from the bag and query among the balls taken out which is the median magic ball. He will do this until the bag is empty, so there will be N-2 such queries. With the appropriate data structure(s), the best algorithm to answer each query will take

i. O(1) time in the worst case

ii. O(N) time in the worst case

iii. *O(logN) time in the worst case

iv. O(NlogN) time in the worst case

Build an AVL tree using the number of the 1st 3 balls, then for each subsequent ball taken from the bag insert its number into the AVL tree in at most O(logN) time.

After that simply call Select operation to select the ball with rank = N'/2 (median ball) if N' is odd, or select the 2 median balls with rank N'/2 and (N'+1)/2 if N' is even and return their average, where N' is the current number of items in the AVL tree. This again can be done in O(logN) time

Thus each query can be done in at most O(logN) time.

44. Two blue magic balls from John's bag can be combined to form a green magic ball (the original 2 magic balls will disappear after the combination). Two magic balls of different colors will combine to form a green magic ball. Green magic balls can also combine with green magic balls to form another green magic ball.

When 2 magic balls X and Y are combined the number of the resultant magic ball will be the sum of the number of X and Y. For example if balls numbered 10 and 12 are combined, a ball numbered 22 will be formed.

Now John will take any two magic ball from his bag and combine them to create a new magic ball which he will put back into the bag. He can keep doing this combination as long as there are magic balls in his bag. Due to this process there can be resulting magic balls in his bag which have numbers > N^2. After a combination step, John can choose to query whether a blue ball numbered H is involved in the creation of the new magic ball. If it is, the query will be true otherwise it will be false.

For example if there are initially 5 magic balls in the bag numbered **1**,**2**,**3**,**4**,**5** (bold numbers are blue balls and underlined numbers are green balls) and 3 combinations were made as follows:

1+**2** = 3

3+3 = 6

6+**4** = 10

After that John queries whether **3** is involved in the creation of 10. This should return true since 10 is created from the set of blue balls {**1**,**2**,**3**,**4**}**.**

With the appropriate data structure(s), the best algorithm will take

i. O(N) time to combine 2 magic balls and O(N) time to answer the query in the worst case

ii. *O(1) time to combine 2 magic balls and O(1) time to answer the query in the worst case

iii. O(logN) time to combine 2 magic balls and O(logN) time to answer the query in the worst case

iv. O(logN) time to combine 2 magic balls and O(N) time to answer the query in the worst case

v. O(1) time to combine 2 magic balls and O(logN) time to answer the query in the worst case

Can be done using a UFDS and a hashmap.

1.) Create a UFDS U with N item (numbered of course from 0 to N-1) where each item represents a magic ball from the bag.

2.) Create a hashset HS of all the N magic balls (store their magic ball number)

2.) Use a hashmap HM to map the magic ball number (key) to the item number (value) in the UFDS.

3.)

i. for combination of magic ball Y and X, get their item number X', Y' from HM in O(1) time, then simply perform unionSet(X',Y'), get the representative of the new set Z' by findSet(X') or findSet(Y').

ii. Now X'+Y' may already exist due to some previous combinations, so query HM to check if there is an entry for X'+Y' if there is, retrieve its key Z'' and now unionSet(Z',Z''), and get the representative item Z''' of the new disjoint set by findSet(Z')/findSet(Z'').

This means that for every possible magic ball number Z* there is only 1 entry in the hashmap and 1 disjoint set representing that magic ball number containing all the blue balls that can possibly combine to form Z*.

iii. create a new entry <Y'+X',Z'> in HM if no such entry exist otherwise step 3ii) applies so update entry with key Y'+X' with Z'''.

All these operations are O(1) time. So combining 2 balls take a total of O(1) time.

4.) To answer the query of whether a magic ball numbered H is involved in the creation of a magic ball numbered X'+Y', first check if H even exists among the original N magic balls by querying HS. -> O(1) time

Next if it exists, query HM using Y'+X' to get the representative item Z of the set representing Y'+X'. Then get the H's item number H' from HM and check if findSet(H') == Z. If this is true then H is involved in the creation of the magic ball else it is not.

Again the query can be answered in O(1) time.

45. Now John changes what he does with the magic balls in his bag. He wants to know given a number K where 1 <= K <= N^2, what is the largest numbered magic ball that can be created from all magic balls <= K. You can assume that the magic ball with number K can always be found in John's bag. There are P such queries.

After pre-processing that takes <= O(NlogN) time, with the appropriate data structure(s) the best algorithm to answer <u>each query</u> will take

i. *O(1) time in the worst case

ii. O(logN) time in the worst case

iii. O(N) time in the worst case

iv. O(N^2) time in the worst case

v. O($\log^2 N$) time in the worst case

Pre-process by creating an AVL tree from all the balls in the bag. Here modify the size attribute for a node V so that instead of storing the number of nodes in the subtree rooted by V it stores the sum of the values of the all nodes in the subtree rooted by V.

i.e V.size = V.value if V is a leaf, else V.size = V.left.size+V.right.size + V.value

this does not change the time complexity to update the size attribute.

Now creation of such an AVL tree will take at most O(NlogN) time since you need to insert all N magic balls into the AVL tree.

Now since each query involves K which must be a valid magic ball, we can do further pre-processing by doing a sort of rank query for each magic ball number, which will return the sum of all magic ball <= K from the AVL tree. The algorithm is as follows

```
prefix_sum(node, val)
    if (val == node.key)
      if (node.left != null)
        return node.size+node.left.size
      else
        return node.size
    else if (val < node.key)
      return prefix_sum(node.left, val)
    else if  (val > node.key)
      return node.left.size + node.key + prefix_sum(node.right, val)
```

For each K simply call prefix_sum(root,K). This will give the largest numbered ball that can be created from all balls <= K. Now use hashmap to store <k,prefix_sum(k)>.

Each call of prefix_sum is O(logN) time. We call it N times so it is still O(NlogN). Total preprocessing is O(NlogN).

To answer each query is O(1) time by using the hashmap.

Another way is simply sort the magic balls using Radix sort in O(N) time. Then go through the sorted list adding the number as we go along. For each magic ball number K we are at, store <K,K+current sum> in the hashmap. This only takes O(N) time for pre-processing and each query is still O(1) time.

46. John has upgraded his magic balls so that the magic balls used in a combination process will not disappear. For example if he uses 11 and 10 in a combination process 21 will be produced but 11 and 10 will still remain. Now given a number K where 1 <= K <= N^2, and a magic ball numbered i from the bag, John wants to know if there is a sequence of combination steps starting with the magic ball numbered i that will give a magic ball numbered K. If there is, return true else return false. There is only 1 such query.

Using the appropriate data structure(s) the best algorithm to answer John's query will take

i. O(N) time in the worst case including any pre-processing

ii. O(NlogN) time in the worst case including any pre-processing

iii. O(N^2) time in the worst case including any pre-processing

iv. O(2^N) time in the worst case including any pre-processing

v. O(N^2logN) time in the worst case including any pre-processing

vi. *O(N^3) time in the worst case including any pre-processing

This can be modelled as a graph reachability problem. Create 1 to N^2 as vertices of a graph. Now for each vertex V, there are at most N outgoing edges where each edge has edge weight equal to a magic ball from the bag. So for each edge it will link vertex V to vertex V+edge weight as long as V+edge weight does not exceed N^2. In total this graph will have $O(N^2)$ vertices and $O(N^3)$ edges.

For the query just start from vertex representing i and perform DFS/BFS then see if K is visited at the end. Total time taken is $O(N^2+N^3) = O(N^3)$.