CS2040 Midterm 2

Time Allowed: 2 hours

For each question, select the correct option.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

i. True

ii. False

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

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

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

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

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

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

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

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

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

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

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

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 $\mathbf{k_{th}}$ 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 $\mathbf{k_{th}}$ 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(Mlog^2N) 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

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

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

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

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]

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

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

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

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

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

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.

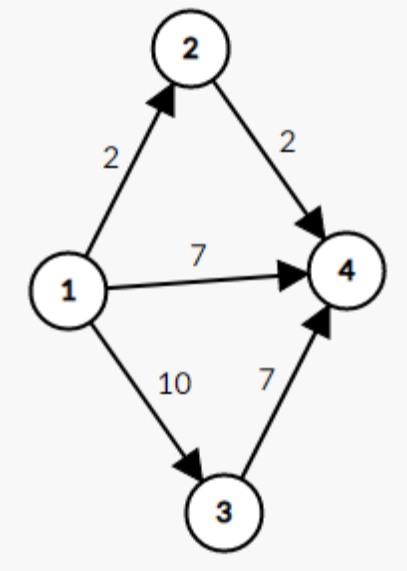
In the new version of the game, Alice still wants to get from node 1 to node N. However, the new version of the game consists of two phases, where Bob performs the first phase once, and Alice performs the second phase once.

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.

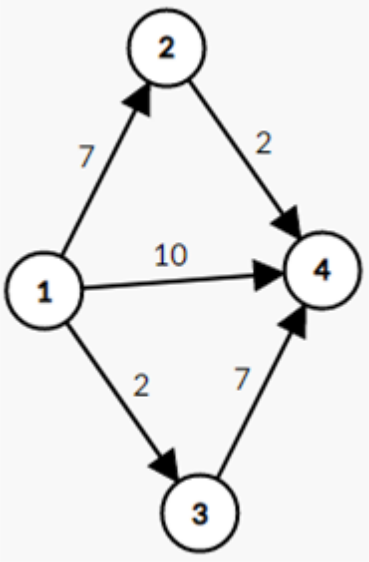
In the **second phase**, Alice begins from node 1 and moves to node *n* via a path of edges. Alice pays Bob an amount of money corresponding to the length of the path.

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

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.

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

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

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

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 each query 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

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