CS2040 2022/2023 Sem 4 Final

MCQ: 40 Marks, 10 Questions, each question worth 4 marks

1. Suppose you are given a BST (not necessarily balanced) of **n** integers. You are asked to create a min heap containing all the values in the BST. The best algorithm to do so runs in worst case time of:

a. O(1)
b. O(logn)
c. O(n)
d. O(nlogn)

Ans:

This can be done by performing inorder traversal on the BST and outputting the result into an array (starting from index 1 instead of index 0, if needed). This takes O(n) time.

2. You are given a UFDS (using path compression and union-by-rank) consisting of **n** disjoint items. You are asked to form a single set which is of rank 3 (i.e the representative item of the set has rank 3) and height 1 (i.e the representative item of the set has height 1, where height is number of edges from the representative item to some deepest leaf).

   You are only allowed to use the **unionSet(i, j)** operation on items belonging to 2 different sets. The direct use of **findSet(i)** and **isSameSet(i, j)** is prohibited. Determine the smallest possible value of **n** required to fulfill your task.

   (Implementation note: for **unionSet(i, j)** the set containing i is moved under the set containing j if ranks are equal)

a. 8
b. 9
c. 10
d. 11

Ans:

This can be done by starting with 10 disjoint items, and running unionSet on the following ordered pairs (this can be verified on VisuAlgo):

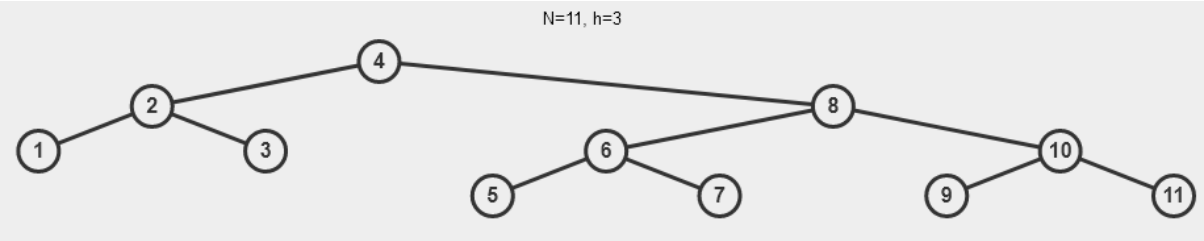
(0, 1), (2, 3), (4, 5), (6, 7), (1, 3), (5, 7), (3, 4), (0, 8), (2, 9)

3. You are given an AVL tree where the only information given is that the tree consists of 11 distinct integers. How many of these integers could be the value stored at the root of the tree?

a. 3
b. 4
c. 5
d. 6

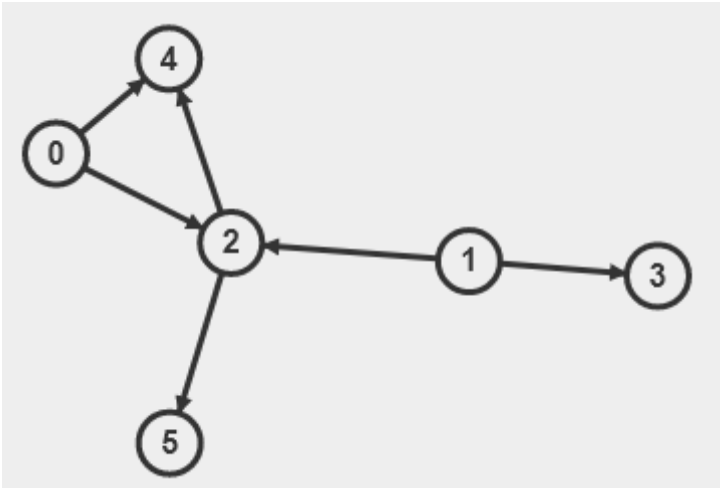
Ans:

To solve this, we need to minimise the number of nodes on one subtree of the root. In this case, the minimum number of nodes on one side is 3, as seen below:



Other valid AVL trees can be formed by moving the root down to the side with less nodes, and shifting a node from the larger side up to the root. This argument can be applied to the mirrored version of this tree as well. Therefore, the integers that could be the root is everything except the smallest 3 integers, and the largest 3 integers, for a total of 5 possible integers.

4. You are given the following graph G', which is a subgraph formed from another graph G by removing some edges from G. No vertices were removed.



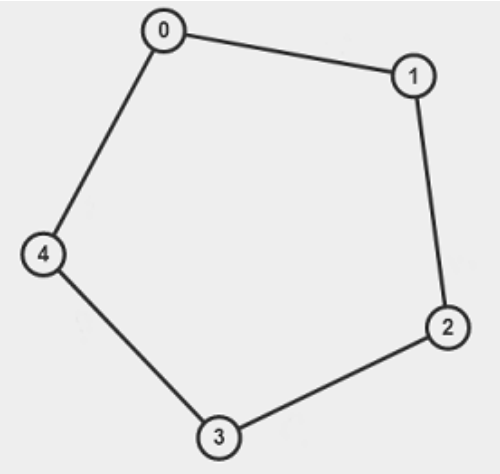
It is known that (0, 1, 2, 3, 4, 5) is a valid toposort of G. Determine the maximum number of edges that were removed from G to form G'.

a. 8
b. 9
c. 10
d. 11

Ans:

The maximum number of edges that can appear in G is 15 (each vertex with a smaller vertex number has an edge pointing to an edge with a larger vertex number). Therefore, with 6 edges present in G', 9 edges could be removed from G to obtain G'.

5. You are given the following graph, consisting of edges of unknown weights:



It is known that if all the edges were of the same weight, there would be a total of 5 valid MSTs for the graph. However, we are more interested in the maximum number of MSTs when the graph consists of 2 distinct edge weights. Suppose that you had to assign each edge a weight of either 1 or 2. At least one edge must have a weight of 1, and at least one edge must have a weight of 2. Under the above constraints, determine the number of edges that should have a weight of 2 in order to achieve the maximum number of valid MSTs for the graph.

a. 1
b. 2
c. 3
d. **4**

Ans:

An edge with the largest weight is always removed from this cycle to form an MST. Therefore, the most possible different MSTs occur if as many edges in the graph are of the larger weight, since this gives more options as to which edges can be removed to produce an MST. Under the constraints, the largest possible number of edges with the largest weight is 4.

6. You are given an unsorted array **A** consisting of 15 distinct integers. You are asked to rearrange **A** such that it forms a valid min heap (with the root of the heap at **A**[0]). To do this, you are given a magic function **magicPartition(A, l, r, x)** where:

   A is the array
   l and **r** are the left and right indices of a subarray of **A**, so **l** <= **r**
   x is an integer within the range [**l**..**r**]

   The method will take the values currently stored in **A**[**l**] to **A**[**r**] (both ends inclusive) and rearrange them such that from **A**[**l**] to **A**[**r**], all elements to the left of **A**[**x**] are smaller than it, and all elements to the right of **A**[**x**] are larger than it.

   You are only allowed to modify **A** via the use of the **magicPartition(A, l, r, x)** method. Determine the minimum number of calls that needs to be made to **magicPartition(A, l, r, x)** such that **A** is guaranteed to represent a valid min heap.

a. 3
b. 4
c. 7
d. 8

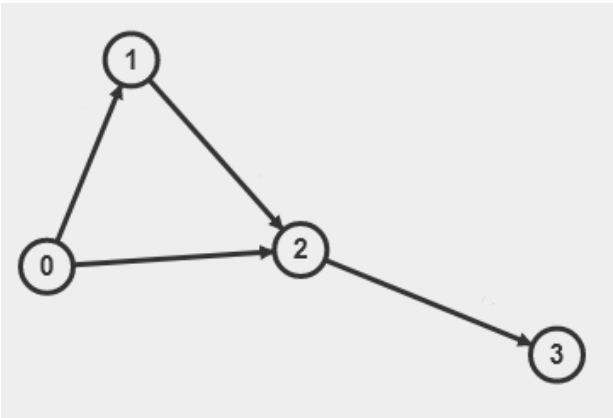
Ans:

The calls that need to be made are:

magicPartition(A, 0, 14, 0): fixes smallest element to A[0]
magicPartition(A, 1, 14, 2): fixes next 2 smallest elements to A[1] – A[2]
magicPartition(A, 3, 14, 6): fixes next 4 smallest elements to A[3] – A[6]

The remaining 8 elements must then already be in A[7] to A[14], or the last level of the heap, thereby ensuring that the parent of each node is always smaller than it.

7. You are given the following graph, with edges of unknown weights:



It is known that:

1. Each edge has a distinct integer edge weight from [1..4].

2. D(0,1) < D(0, 2) < D(0, 3), where D(a, b) is the minimum distance from vertex a to vertex b.

Determine the possible edge weight of edge (1, 2). Select all options that apply.

a. **1**
b. **2**
c. **3**
d. **4**

Ans:

All of them will work.

This question does not demand trying all 4! = 24 permutations of weights. Instead, consider the following:

1. The condition D(0, 2) < D(0, 3) is irrelevant (the only paths from 0 to 3 pass through edge 0 -> 3, which is guaranteed to have weight > 0). Therefore, we only need to consider D(0, 1) < D(0, 2)
2. Using an edge weight of 4 for 0 -> 2 guarantees that D(0, 1) < D(0, 2), since the edge 0 -> 1 must then have a weight of 3 or less. It thus becomes possible to have edge 1 -> 2 be of weights 1, 2 or 3 in this manner.
3. Using an edge weight of 1 for 0 -> 1 guarantees that D(0, 1) < D(0, 2), since:
    a. The path 0 -> 1 -> 2 must have total cost greater than that of 0 -> 1, as all edges have positive weight.
    b. The edge 0 -> 2 must have a weight greater than 0 -> 1, since 1 is the lowest possible weight.
4. From (3), it is possible to use edge weights 2, 3, or 4 for the edge 1 -> 2.

From (2) and (4), all possible weights of 1, 2, 3, 4 are covered.

8. Given a directed unweighted graph in adjacency matrix form (as given in the lecture notes), the sum of all entries in the adjacency matrix represents:

a. Number of directed edges in the graph
b. Number of vertices in the graph
c. Total distance from vertex 0 to all other vertices combined
d. Nothing in particular

Ans:

As every edge appears as a 1 for an unweighted graph, summing up all entries simply gives the number of edges in the graph.

9. You are given the following pseudocode to run on an undirected graph **G**, given in adjacency list form, which is used for Q9 and Q10:

```
int DFS(int vertex, int pre):

    ans = 1
    neighbours = AL.get(vertex)
    for i in neighbours:
        if (i != pre):
            ans += DFS(i, vertex)
    return ans
```

The initial call to this method is **DFS(0, -1)**.

Determine the time complexity of this pseudocode when **G** is a tree with **V** > 5 vertices. You may assume you have infinite memory to run the method.

a. O(V)
b. $O(V^2)$
c. $O(V^V)$
d. The call to DFS(0, -1) will run infinitely

Ans:

The pseudocode runs in much the same manner as standard DFS (O(V) time on a tree), as it is not possible to go back to a previously visited vertex, since there are no cycles in a tree.

10. Determine the time complexity of the pseudocode (shown in Q9, the previous question) when **G** is a complete graph with **V** > 5 vertices and **DFS(0, -1)** is called.

a. O(V)
b. $O(V^2)$
c. $O(V^V)$
d. The call to DFS(0, -1) will run infinitely

Ans:

In a complete graph, it becomes possible to revisit a previously visited vertex due to the presence of cycles. The pseudocode may thus be trapped in a cycle (eg. constantly visiting 0 -> 1 -> 2 -> 0 -> 1 -> 2 -> 0, assuming neighbours are sorted in ascending order in the AL).

Analysis: 16 marks, 4 questions (4 marks each)

11. Connor wants to implement an ADT with the following operations and time complexity requirements for each operation:

    1. insert(x) - inserts an integer value x into the ADT in amortized or worst case O(1) time.
    2. query(x) - return true if the integer value x is on the ADT, false otherwise. Must be done in worst case O(logn) time or better. (Note that average case O(1) is not better than worst case O(logn)).

    There are 2 more pieces of knowledge that Connor has about the integers to be inserted:
    1. The integers will be inserted from smallest to largest
    2. The largest integer is known

    Connor believes that none of the DSes - {**array, BST (not necessarily balanced), hashtable**} used on their own and not in combination can implement this ADT.

    (Select the most appropriate option)

    a. Connor is correct.

    b. Connor is incorrect. All the DSes listed can be used to implement his ADT.

    c. Connor is incorrect. The array can be used to implement this ADT, since the integers can be inserted to the back of the array (amortized O(1) time) and query can be done by performing binary search (O(logn) time). The other DSes listed cannot implement his ADT in the required time complexity.

    d. Connor is incorrect. The array and hashtable can be used to implement this ADT.

    e. Connor is incorrect. The BST can be used to implment this ADT since a reference can be maintained to point to the current largest in the BST and when inserting the new node can be inserted as the right child of the current largest abd the reference updated to point to the new node (O(1) time).

    Query is simply a search in the BST (O(logn) time).

Ans:

d. is correct.

Array can be used, reason as given in option c. Hashtable can also be used since largest is known, a DAT containing boolean values of size = largest value can be created.

For insertion simply go to index = x - 1 and set the slot to true (O(1) time). For query simply go to index = x-1 and return the boolean value (O(1) time).

BST cannot be used, because even though insertion is O(1) this leads to an unbalanced tree (totally skewed to the right), thus query will be worst case O(n) time.

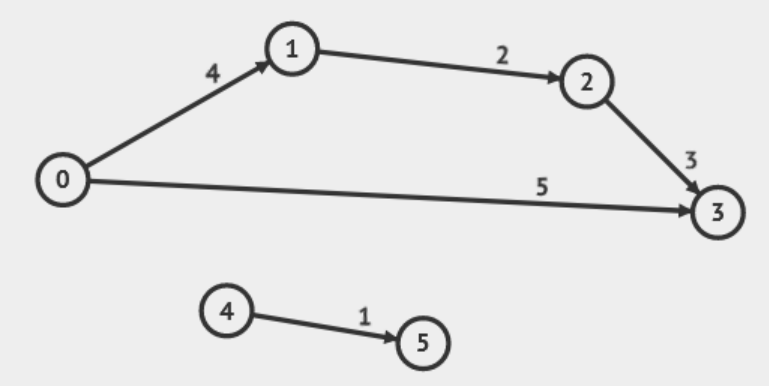
12. Given a positively weighted, directed graph **G** and a source vertex **s** of **G**, a shortest path spanning tree (SPST) from **s** must include the smallest weighted edge in **G** if the weights of all the edges in G are unique.

    (select all option(s) that is/are part of a correct rationale for whether you think the statement is true or false).

a. True, since the smallest edge must be reachable from **s** and thus must be used in the SPST.
b. True, since the edge weights are unique thus there is no alternative to the smallest edge and so it must be in the SPST.
c. True, because there must be a least costly path from **s** to some other vertex involving the smallest edge (call it (a,b)).
   If we do not use the smallest edge in that path it must be replace with a path from a to b which must be more costly than (a,b) thus there is no better shortest path.
d. False, since the smallest edge might not even be reachable from **s**
e. False, since the smallest edge (call it (a,b)) might not be involved in all possible shortest paths from s to b, thus all possible SPs using s to b as a subpath will not use the edge (a,b) at all, so (a,b) will not be in the SPST at all.
f. False, uniqueness of edge weights is not required to include the smallest edge in the SPST from **s**. Even if there are multiple smallest edges, all of them should be included in the SPST.

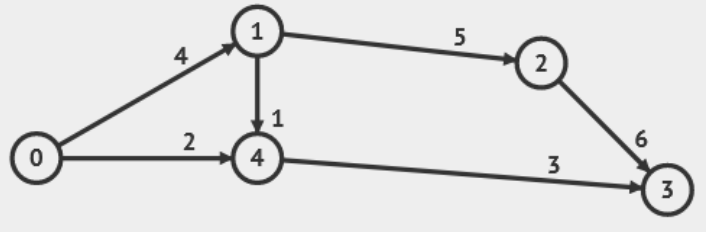
Ans:

Only option d and e is correct. All options that support true is not correct. A counter example is given below:



Here all the edge weights are unique. If we use 0 as source, the SPST from 0 will not involve vertices 4 and 5 since they are not reachable from 0, thus the edge (4,5) which is the smallest edge in the graph is not included in the SPST. This also shows that option d is correct.

Option e is also correct and example is given below



In the example, (1,4) is the smallest edge but if we use 0 as the source, it is not involved in the only SP from 0 to 4 since that SP is 0->4 instead of 0->1->4. Thus it is also not involved in the SP from 0 to 3 which uses the SP from 0 to 4 as a subpath (the path 0->4->3). So the edge (1,4) will not be in the SPST from 0 at all.

Option f is not correct, since it is trying to say that all the smallest edges (whether they are unique or not) should be included in the SPST.

Question 13 to 14 refers to the following problem

13. **[2 marks]** Given a sequence of non-repeating integer values to be inserted one by one from the first integer to the last integer into a max heap, the 1-based compact array used to implement the max heap will always contain the same integers at the same position if you reverse the order of insertion (i.e insert the integers from last integer to the first integer in the sequence)
    Note: The insertion algorithm is the one in the lecture notes for binary heaps.

    True/**False**

14. **[2 marks]** Give your rationale for your answer to the previous question.

Ans:

False

An easy counter example is as follows:

Inserting 3,2,1 one by one into a binary max heap will result in the following 1-based compact array:

3,2,1 (ignoring index 0)

Inserting 1,2,3 one by one into a binary max heap will result in the following 1-based compact array:

3,1,2 (ignoring index 0)

Thus you get 2 different max heaps.

Grading Scheme:

Two marks is awarded if students are able to come up with a valid counter-example.

One mark is awarded if students came up with a valid counter-example but traced it wrongly (mostly due to coming up with a counter-example of a huge size with >5 elements).

Some notable (incorrect) answers which are not awarded any marks:

1. Some argued in this manner: "Because we can only guarantee the root of the max heap to be the maximum, we cannot guarantee the position of the integers is the same if inserted in the reverse order." The nature of the binary heap root do not explain why the positions of the remaining non-maximum integers could be different if inserted in reverse ordering.

2. Extending from point 1: "Because of max heap property which states that parent values must be bigger than the children values, we cannot guarantee the position of the integers is the same if inserted in the reverse order." The max heap property do not explain why the positions of the remaining integers could be different if inserted in reverse ordering.

3. Some also argued in this manner: "If we insert values in descending order, the array will not violate max-heap property. If we insert values in reverse, the position of the integers **will** change since it requires shiftUp operations." This is not necessary true, considering inserting {1,2} and {2,1}.

4. Some argued that entries will be sorted in ascending order in the array when inserting all values into a binary heap, and hence will always result in the same integers in the same position. **This is a critical misconception.**

5. Some argued that the last few entries will be added to the "right side" of the heap. This is not true when the number of entries does not amount to a complete binary tree (or a "near-complete" binary tree). For example, when there are 4 elements, the last element will be added to the "left side" of the heap.

6. Continuing from the above point, there are situations where integers inserted in the left subtree of the root will end up in the right subtree (and vice-versa) e.g. inserting {1, 2, 3} in sequence will result in 2 being inserted in the left subtree of the root, and then landing in the right subtree of the root at the end.

7. Some came up with counter-examples that does not prove the point, most commonly the insertion of {1,2,3,4} in sequence and in reverse, which will yield the same binary heap.

Question 15 to 16 refers to the following problem

15. **[2 marks]** Given an unweighted directed graph stored in an Adjacency Matrix A, if we want to reverse the direction of the edges, we can do this without changing A. Simply reverse the meaning of 0 and 1 in the matrix. 0 in an entry A[x][y] will now mean there is an edge x->y and 1 will mean there is no edge from x to y.

    True/**False**

16. **[2 marks]** Give your rationale for your answer to the previous question.

Ans:

False.

For vertices x and y where there is an edge x->y, yes this will reverse the edge, however, for vertices where there are no edges to begin with, then they will be a bidirectional edge!

Grading Scheme:

Two marks is awarded if students can identify either (A) vertices that do not have a edge between each other now has two edges AND/OR (B) vertices now have an edge to itself.

One mark is awarded if students have small misconceptions about graphs that does not affect their rationale (e.g. directed graphs are only uni-directional graphs).

For some reason a majority of the students misunderstood the question: we are only changing what it means to have values 0 and 1 in the adjacency matrix, not changing all

values in the adjacency matrix from 0 to 1 and vice-versa. No marks are deducted if the rationale fits the latter scenario.

Minimal students got zero marks for this, but if they do, it's most likely due to other misunderstanding of the question (e.g. Adjacency List instead of Adjacency Matrix, repeating what the question is trying to do, wrong assumptions such as applying the changing of meaning on an undirected graph).

Structured Questions: 4 questions

This section is worth 44 marks. Answer all questions.

Write in **pseudo-code**.

Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes.

Partial marks will be awarded for correct answers not meeting the time complexity required.

17. **[10 marks]** District X is the central business district of city Y and people from other districts in Y will go to X for work in the morning and go back home in the evening. These people are so familiar with how to get from their home to X that they will always use the fastest route to get there. You may assume that is only 1 fastest route from each district to district X. In fact all fastest route to get from any district to X is used by someone.

    One day in the middle of the night, there was an earthquake and many of the roads in city Y have been damaged.
    In the morning, you hear from the news that the roads used by people to get from their district to X are the most badly affected since those roads are already under stress from over usage.
    Since you still have to get to work in district X (you are from district Z), you are worried that some roads along your usual route may not be passable. Thus you want to find an alternate route which is the following:

    The fastest route made up of only roads which is not used by anyone to get from their district to X.

    The road network of city Y is represented as a graph of V vertices and E edges where each district is a vertex and and each road (which is 2-way) linking 2 districts are undirected edges linking the respective vertices.
    The weight of each edge is a positive integer value representing the time to travel the edge/road. It is guaranteed that you can always get from any district to any other district in city Y via the road network.

Come up with the most efficient algorithm to find the best alternative route as given above. If there is no such route return "no such route". State the time complexity of your algorithm.

You may assume the graph is stored in an adjacency list **AL**, and you will be given the 2 vertices X and Z (representing district X and Z respectively).

Ans:

Idea is as follows:

1. Find all roads are that used in some shortest path from every other district in Y to district X.
2. Remove all these roads from AL then find the shortest path from Z to X.

For 1., the algorithm is as follows:

1. Run modified/original Djikstra on AL using X as source → O((V+E)logV)
2. The predecessor array p stored all edges/roads used is some SP to get to X.

Total time complexity for this part is O((V+E)logV)

For 2., the algorithm is as follows:

1. Create a new Adjacency list AL' → O(V)
2. Go through AL as follows: → O(V+E)
    a. for each index i of AL
        for each neighbor v of AL[i]
            if p[v] != i and p[i] != v then // this road not used in any SP to X
                add v to AL'[i]
3. run original/modified Dijkstra on AL' using X as source → O((V+E)logV)
4. if D[Z] is not infinity get the path from p (backtrack on p from S until hit X) and return it else return "no such route" → O(V)

Total time complexity for this part is O((V+E)logV)

Thus total time complexity is O((V+E)logV).

Grading Scheme:

There are two parts to the answer: determining the roads used by the shortest paths, and finding the shortest path along the remaining roads. Each part is worth 5 marks. Additional comments for each part:

Determining roads:

5 marks are deducted if an answer assumes that the roads to be removed from consideration are already provided to you, thereby skipping this part of the answer.

4 marks are deducted if an MST algorithm was used to determine the roads used by the shortest paths. MST solves the minimax problem, and not the shortest path problem.

2 marks are deducted for solutions proposing running Dijkstra's from all other vertices, to find dist[X] for each vertex. This runs in O(VE log V) time.

1 mark is deducted If Dijkstra's was run from X, but the subsequent determination of the edges used is inefficient. The most common such answer involves tracing the paths from each of the vertices back to X (ie. recursively checking p[i] until p[i] == X) with no optimisation, instead of only checking for a vertex's immediate predecessor. This can cause the algorithm to run in O(V^2) time.

No marks are deducted for assuming that the shortest path from district Z to X should not be removed. While this path should indeed be removed, this is relatively easy to fix in implementation, and hence the issue merely boils down to an interpretation of the question.

Some answers use a hashset (or similar DS). Operations involving hashsets are considered O(1) instead of the worst-case O(N) for this question, since the question does not specify the use of worst-case time.

Finding shortest path along remaining roads:

1 mark is deducted for missing the case where no valid path exists. While the original city layout allows for all districts to reach X, it may no longer be possible for district Z to reach district X after certain roads are removed from consideration.

Other notes:

Up to 3 marks are deducted for not specifying the source when running SSSP algorithms. As the result of SSSP greatly differs depending on the source selected, it is important that this is specified in the answer.

Other answers:

A maximum of 3 marks is given if the answer only involves the use of MST algorithm. As mentioned earlier, this does not solve the shortest path problem.

A maximum of 5 marks is given if the answer attempts to combine both parts by finding the second shortest path from Z to X. This is unlikely to provide the correct answer to the question, and additionally such implementations do not correctly determine the second shortest path to begin with.

18. **[12 marks]** Give an algorithm to build a binary max heap **H** consisting of positive integers 1 to **N** (1 and **N** inclusive) in worst case **O(N)** time without performing any **shiftDown** operation. If this is not possible, just write "not possible".

Ans:

Simply use a loop to store the integers N down to 1 starting from index 1 up to index N of the heap.

The descending order creates a valid max heap, and there is no need to call shiftDown (i.e no need for fast heap create method).

Grading Scheme:

1 mark is deducted for answers that correctly explicitly sort the numbers (in O(N log N) time) before adding them to the heap. Explicitly sorting the numbers is not necessary, but the idea of adding them in some sorted order is there. While some students may feel that sorting is necessary due to interpreting the question as containing only a subset of numbers from [1..N], note that it is still possible to sort them in O(N) time via some other sorting algorithm eg. counting sort, which can be thought of as a specific variation of standard radix sort. Another way is to simply create a DAT of size N and mark each slot if the number represented by the index of the slot is to be inserted, then simply go through the DAT from back to front and add the numbers to **H** (in descending order which will result in a valid max heap).

Answers that otherwise involve time worse than O(N) will receive a maximum of 9 marks. The most common answer is to run standard insertion on the heap in no particular order, which while utilising only the shiftUp method, cannot be guaranteed to run in O(N) time total.

Answers that simply state “not possible” receive 2 marks.

19. **[10 marks]** You are given an undirected weighted connected graph **G** with **V** vertices and **V**+**c** number of edges, where **c** is some constant > 1, and where **V** number of the edges are of weight 1 and **c** number of them have weights > 1.
G is stored in an adjacency list **AL**. Give an algorithm to find the cost of the MST of **G** in worst case **O(V)** time.

Ans:

1. Run counting components algo on G, ignoring edges of weight > 1. Here we will label each vertex by the component number of the component they belong to. → O(V) time

2. If component count == 1, return V-1 as the cost, else go to step 3.

3. Go through the adjacency list of G, for each edge (u,v) where u and v do not have the same component number, insert <(uc,vc),weight(u,v)> into a hashtable H where uc = component number of u and vc = component number of v, (uc,vc) is the key and weight(u,v) is the value. If an entry for (uc,vc) is already present and the value is > weight(u,v) update value to be the new weight. → there are at most c such edges thus time taken is O(1) time

4. Go through H and create an edge for each entry that is stored in an Edge list EL. Thus EL represents graph G' where each vertex is a component of the original graph. → O(1) time since EL is at most size c.

5. run Kruskals on EL to select the edges to form an MST of G' (summing up the cost of the edges selected). Let MST_cost be the sum and let num_edges be the number of edges selected. → since EL is at most size c, time taken is O(1)

6. Return MST_cost+((V-1)-num_edges)

Total time taken is dominated by step 1 which is O(V).

Grading Scheme:

Full marks for a correct solution which runs in O(V) time.

Partial marks:

- 8 marks – V UFDS ops take amortized O(invAckermann(V)) time so you need O(V invAckermann(V)) time
- 6 marks – radix sort, kruskal's algo, O(V log(largest weight)) time
- 5 marks – run kruskal's algo as usual in O(V log V) time

Wrong algorithm:

- remodelling graph to split edges of weight w into w edges, then BFS/DFS
    - This algo may partially choose edges which are not supposed to be in the MST
    - Edge weights are also not bounded (inefficient) and may not even be an integer (fail to re-model)

20. **[12 marks]** The DDEA (Department to discover extraterrestrial activity) has modified their tracking of signals from space.

(Recall that DDEA will record a sequence of positive integers representing signals from space at regular intervals).

The DDEA will first "clean up" each of the recorded signals so that they will be unique from each other.
Also now the DDEA requires you to implement an ADT that consists of the following operations:

1. **insert(int x)**: insert a signal x into the ADT as the latest signal being recorded.

2. **insertAndTrack(int x)**: insert a signal x into the ADT as the latest signal being recorded and also track the largest signal from this point onwards until the next call to **insertAndTrack**. This is called a tracking.

For example if the following signals are inserted with the bolded one being inserted and tracked

1, 2431, 14, 41, **33***, 8178, 221, 13, 11

then at the point when 11 is inserted, the largest signal being tracked will be 8178

Another example, if the following signals are inserted with the bolded one being inserted and tracked

231, **1411***, 13, 131, 111

at the point when 111 is inserted, the largest signal being tracked will be 1411

Note that multiple trackings can be done. An example is as follows

321, 32131, **132***, 3113, 773111, **13119***, 3178111, 1313

when 1313 is inserted, the tracking that starts at 132 will have 773111 as the largest signal being tracked, while the tracking that starts at 13119 will have 3178111 as the largest signal being tracked. This is because the 1st tracking starting at 132 ended at 773111 and thus will not include 3178111.

3. **NumSmalllerThanLatestTracking()**: return the number of trackings with the largest signal less than the largest signal of the latest tracking. If there is no tracking return -1

For examples:

1, 2431, 14, 41, **33***, 8178, 221, 13, 11

There is only 1 tracking with largest signal tracked being 8178, so there is no other tracking thus 0 is returned.

321, 32131, **132***, 3113, 773111, **13119***, 3178111, 1313

There are 2 trackings one with largest signal of the 1st tracking being 773111 and the other (the latest tracking) with largest signal being 3178111. Since the largest signal of the latest tracking is larger than the largest signal of 1 other tracking so 1 is returned.

321, **321312***, 132, **133***, 3113, 77311, 45, **222***, 98982

There are 3 trackings, the first one with largest signal being 321312, the second one with largest signal being 77311, and the third (latest tracking) with largest signal being 98982. Thus 1 is returned since 98982 is only bigger than 77311 and not 321312.

4. **delete()**: This will delete the latest recorded signal. If there is a tracking associated with the signal (i.e the signal was inserted using **insertAndTrack()**) remove the tracking too.

Implement each of the above operations so that they will run in worst case **O(logn)** time, where **n** is the current number of signals recorded.

Ans:

Use a Stack S and AVL A:

S stores an entry for each signal as a pair <largest, tracking>. tracking is 1 if the signal is the start of a new tracking, 0 otherwise, and largest is the value of the largest signal so far for the tracking that this is signal is associated with. If the signal is not associated with any tracking largest is -1.

A stores has a node for each tracking and the key for the node is the current largest signal associated with the tracking.

Let z be a reference to the node representing the latest tracking in the AVL, which is initialized to null

1. **insertAndTrack(x)**
    a) S.push(<x,1>)
    b) Create a node v containing x.
    c) Insert v into the A → O(logn)
    d) z = v

Total time complexity = O(log n)

2. **insert (x)**
    a) if z is null, S.push(<-1,0>)
    b) else S.push(<max(S.peek.largest,x),0>)
    c) if z is not null and z.key < x
        Delete z from A → O(log n)
        Create new node v containing x
        Insert v into A → O(log n)
        z = v

Total time complexity = O(log n)

3. **NumSmalllerThanLatestTracking()**
    a) If A is empty return -1
    b) Return (rank of z)-1 → O(log n)

Total time complexity = O(log n)

4. **delete()**
    a) If S.peek().tracking == 0
        S.pop()
        If S is not empty and -1 < S.peek().largest < z.key
            Delete z from A → O(log n)
            Create new node v containing S.peek().largest
            Insert v into A → O(log n)
            z = v
        else if S is empty
            z = null
            clear A
    b) else
        S.pop()
        Delete z from AVL → O(log n)
        If A is not empty
            Find node v where key == S.peek().largest → O(log n)
            z = v
        else
            z = null

Total time complexity = O(log n)

Grading Scheme:

O(log N) algorithm for every operation – graded out of 12 marks: 2 2 6 2 marks respectively for each operation from number 1 to number 4.

O(N) algorithm for every operation – graded out of 6 marks: 1 1 3 1 marks respectively for each operation from number 1 to number 4.

Approach almost there, but missing some data structures / data:

- -3 marks for not storing prefix max, so unable to recover max on delete (for solutions without bBST for each tracking)
    - -2 marks if stored for current tracking, but loses former prefix max elements once there is a new tracking
    - -2 marks for missing stack of all signal strengths, otherwise solution works (solution likely has list of bBSTs)

For the operation NumSmalllerThanLatestTracking():

- -6 marks for not finding given statistic, and design doesn't allow for quick fix
- -3 marks for not finding given statistic, but the tree has nodes that allow correct statistic to be found easily
- -1 mark if rank is off by 1

Common mistakes:

- Assuming that there are few (e.g. constant number of) trackings
- Removing from PQ, expecting operation to be efficient
- Storing (for those not using bBST in each tracking) signals instead of prefix maxes within a tracking
- Having a giant bBST of ALL elements which is not helpful in finding either max signal strength within tracking, or, rank of max trackings
- (did not penalize) Iterating through stack repeatedly (why use a stack then?)
- (did not penalize) Using HashMap<sequential index, element> when Element[] or ArrayList<Element> works