CS2040S 2021/2022 Semester 1 Final

MCQ

This section has 10 questions and is worth 30 marks. 3 marks per question.

Do all questions in this section.

1. Which of the following will allow us to determine if an undirected graph is disconnected?

a) The number of vertices only
b) The number of edges only
c) The sum of the degrees of all vertices only
d) None of the other options.

It is possible to deduce if an undirected simple graph is disconnected if $E \leq V - 1$. However, this requires both the number of vertices and number of edges. In the general case, it is required to perform a BFS / DFS traversal. The sum of degrees of vertices is just twice the number of edges by handshaking theorem, which is not useful.

2. You are given a collection of *n* integers to be stored in a data structure *X*. The data structure *X* needs to support one operation **removeSmallTwo()**, which removes the second smallest element from *X*. The operation **removeSmallTwo()** is the most efficient (in terms of worst case time complexity) in general if *X* is a?

a) Array
b) Minimum binary heap
c) Binary search tree
d) None of the options can implement this operation.

Using an array we can find the second smallest in O(n) time(it is not said to be sorted) and removal will take another O(*n*) time. For a binary search tree that is not necessarily balanced, the search will take O(*n*) time to find the minimum, followed by another successor operation. For a binary heap, we can ensure that the second smallest element is a child of the root, and we can do a normal delete procedure by swapping with the last element and do a shift-down in O(*log n*) time.

3. The following strings are inserted successively into an initially empty Trie:

celz
algo
clez
alto
aelz

How many nodes (including the root) will be present in the Trie after all the strings are inserted?

a) 15
b) 17
c) 18
d) 20

After adding “celz”, there are 1+4 = 5 nodes (**celz**)

After adding “algo”, there are 5+4 = 9 nodes (**algo**)

After adding “clez”, there are 9+3 = 12 nodes (c**lez**)

After adding “alto”, there are 12+2 = 14 nodes (al**to**)

After adding “aelz”, there are 14+3 = 17 nodes (a**elz**)

4. Consider a UFDS with the parent array below:

Index	0	1	2	3	4	5	6	7	8
p	1	4	3	4	4	4	7	4	7

The size of the set with the largest number of elements is

a) 3
b) 4
c) 6
d) 9

Reproduce the sets by drawing the diagram. All the elements belong to the same set with parent 4.

5. Select the option that fills in the blanks most appropriately.

To store a directed, weighted graph represented by a ____, we require ____ space. We can then check if any two vertices are neighbours in ____ time.

a) Adjacency matrix, O(V), O(1)
b) Adjacency list, O(V), O(E)
c) Adjacency list, O($V + E$), O(V)
d) Edge list, O(E), O(1)

(c) is the only valid option. Adjacency matrix requires $O(V^2)$ space. Edge list can only query in O(1) if you pre-process it until it's like adjacency mat (defeat the purpose of edge list).

6. Consider a weighted graph with V vertices and E edges where all edges are negative, but without any negative cycles. We can find the shortest paths from one vertex to all other vertices in not more than

a) O($V + E$) time
b) O(VE) time
c) O($E \log V$) time
d) Shortest paths cannot be found.

Since the graph has no negative cycles and all edges are negative, this means the graph has no cycles (must be a tree) so can just run DFS/BFS on it to get the answer in O(V+E) time.

7. Which of the following data structures can be used to implement a Queue ADT which requires the operations ***enqueue***, ***dequeue***, and ***peek***?

i. Array
ii. Linked list
iii. Binary heap
iv. Binary search tree

a) i and ii only
b) i, ii, and iii only
c) i, ii, and iv only
d) i, ii, iii, and iv

Array and Linked list are standard. For binary heap and binary search tree, all we need to do is insert them into the data structure with an insertion "time", and use the "time" as the basis for comparison in the data structure. This allows us to maintain the FIFO property.

8. Consider an **undirected** connected weighted graph with *V* vertices and *E* edges with non-negative edge weights, and a vertex in the graph labelled X. We want to find the shortest paths from all other vertices to the vertex X. The most efficient algorithm to do this will run in worst case

a) O(VE) time
b) O((V+E)*logV) time
c) $O(V^3)$ time
d) None of the other options

We can run Dijkstra's algorithm from X to get the shortest paths from X to all other vertices. Since the graph is undirected, reversing these paths will give us the shortest paths from all other vertices to X.

9. Let *E* be an array of edges representing an undirected graph. Consider the following algorithm called *unknown*.

```
function unknown(E)
    sort E in decreasing order
    i = 0
    while i < size(E)
        edge = E[i]
        delete E[i] // size of E decreases by 1
        if graph is not connected then
            E[i] = edge
            i = i + 1
    return E
```

The algorithm *unknown* returns a list of edges that represents

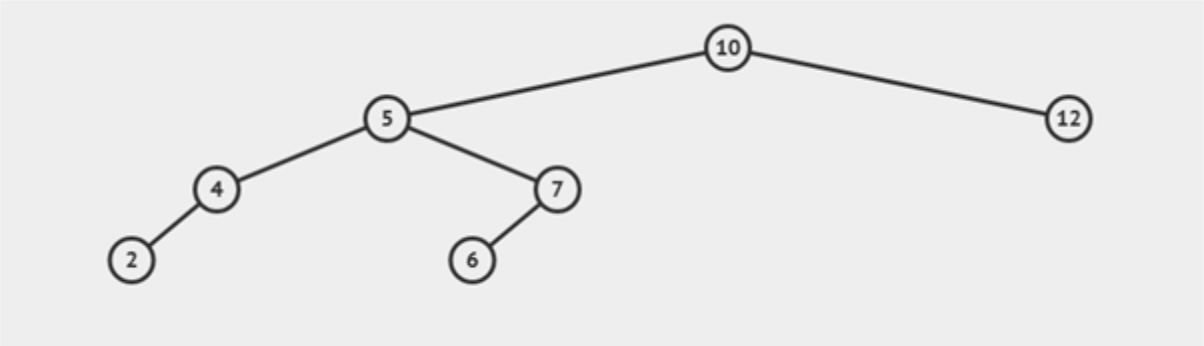
a) Edges not in the minimum spanning tree
b) Minimum spanning tree
c) Edges not in the maximum spanning tree
d) Maximum spanning tree

The algorithm is in fact the reverse-delete algorithm, which is the reverse of the Kruskal's algorithm. We start with the original graph and try to delete edges from the graph in descending order. If the deleted edge results in the graph being disconnected, we add the edge back into the graph and check the subsequent edges. Otherwise, we remove it permanently.

The intuition behind the algorithm is similar to Kruskal's. By the cycle property, the largest edge in every cycle will not be part of any MST. By looking at edges in descending order, we can remove those largest edges that are part of cycles. Since those edges are part of cycles, they will also not disconnect the graph.

The full proof can be completed by proving the inductive proposition: There exists a subset of edges in *E* that is a minimum spanning tree of the graph.

10. A subtree *S* of a tree *T* is a collection of nodes where the root node r of *S* is a node *g* in *T*, and all the descendants of *r* are the same as that of *g*, in the same structure.



For example, the tree in the second diagram is a subtree of the tree in the first diagram.

Consider two balanced BSTs *A* and *B* with *n* and *m* nodes respectively, where n >= m, and there are no repeated nodes (i.e no repeated keys) in either tree. Checking if *B* is a subtree of *A* requires worst case time complexity not more than

a) O(*n* + *m*)
b) O(*log n* + *m*)
c) O(*n* + *log m*)
d) O(1)

First step is to use O(*log n*) time to search for the root of *B* in *A* to find the candidate subtree. Then, run a post-order traversal (pre-order traversal also can be used) on *B* and on the candidate subtree in O(*m*) time. If the sequence of nodes visited for B and the candidate subtree are the same then *B* is a subtree of *A*.

Analysis

This section has 4 questions and is worth 16 marks. 4 marks per question.

Please select True or False and then type in your reasons for your answer.

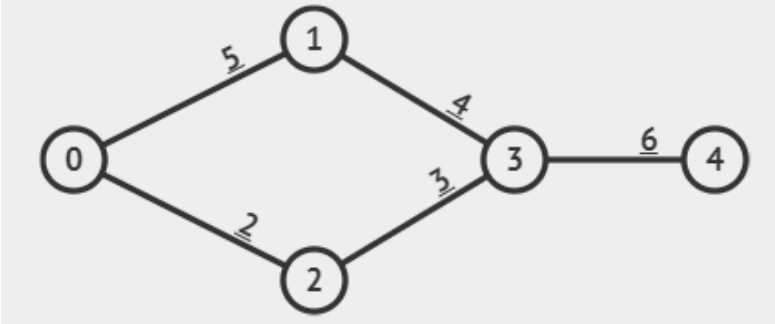
Correct answer (true/false) is worth 2 marks.

Correct explanation is worth 2 marks. Partially correct explanation worth 1 marks.

Do all questions in this section.

11. For a given undirected weighted graph G, any minimax path from some vertex v to some vertex u in G must be a path from v to u in some valid MST of G.

False. In the diagram below, 0,1,3,4 is a minimax path but it will not exist in any MST of the graph since it contains edge 0,1 which cannot exist in any MST of the graph.



Marking scheme:

2 marks for correct answer
1 mark for explaining that the minimax path need not be unique
1 mark for explaining that the minimax path need not be part of MST
(If counterexample is given, both marks for explanation will be awarded)

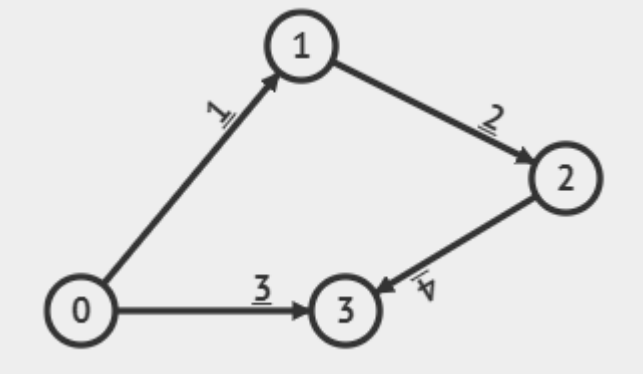
Common mistakes:

1. Giving the answer is True, assuming that the minimax path is unique or will always be part of the MST.
2. Explaining that the minimax path need not be part of the MST, without explaining why the minimax path is not necessarily unique (because we know that the MST must contain one minimax path, without non-uniqueness you have a contradiction).
3. Giving invalid counterexamples that do not illustrate why the statement is false.
4. Using non-distinct edge weights as explanation by saying that there can be multiple MSTs and thus multiple minimax which does not disprove the statement.
5. Confusion between MST, shortest path and minimax path definitions.

12. The algorithm below can be used to detect cycles in a **directed** graph G of V vertices (labelled 0 to V-1) stored in an edgelist L as follows:

```
1. Sort L according to increasing edge weight
2. create a UFDS U containing V items/disjoint sets initially
3. Go through each edge (u,v) in the sorted L
    if U.isSameSet(u,v) == true
       return true
   else
       U.unionSet(u,v)
4. return false
```

False. Due to the direction of the edges, the following graph will be falsely determined to have a cycle although it has none.



Marking scheme:

2 marks for correct answer
1 mark for explaining that UFDS unions are “bidirectional”
1 mark for explaining why the algorithm fails to identify cross edges
(If counterexample is given, both marks for explanation will be awarded)

Common mistakes:

1. Assuming the graph is undirected.
2. Invalid counterexamples.

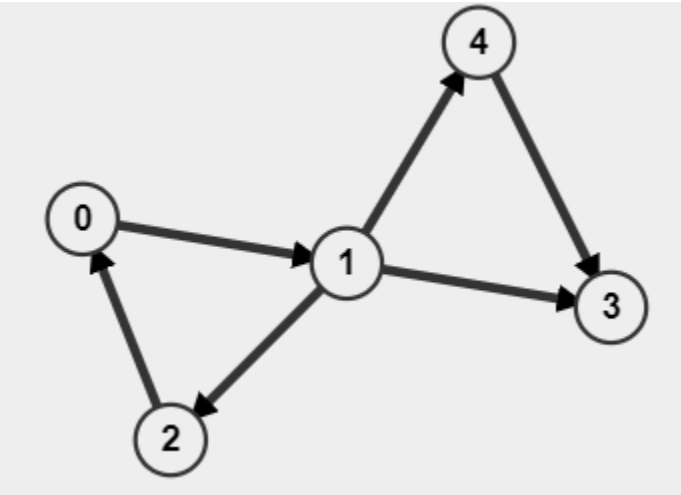
13. Kosaraju's algorithm is as follows:

```
1. Use DFS topological sort on the graph G to post-order process the vertices
   and get K, the post-order sequence of the vertices
2. Tranpose the graph G to get the transpose graph G'
3. perform counting of SCCs by going through all vertices from last to first
   as listed in K on G'
```

The following modification to Kosaraju's is still a correct algorithm to count SCCs:

```
1. Use DFS topological sort on the graph G to post-order process the vertices
   and get K, the post-order sequence of the vertices
2. Perform counting of SCCs by going through all vertices from first to last
   as listed in K on G
```

False. In the graph below, if neighbors are scanned from smallest to largest vertex number, the post order sequence is 2,3,4,1,0. Now counting SCC starting from vertex 2 on the graph will result in only 1 SCC which is wrong. There are 3 SCCs (0,1,2),(3) and (4).



Marking scheme:

2 marks for correct answer
1 mark for explaining importance of transpose graph / reverse topological order
1 mark for explaining why the algorithm will undercount the number of SCCs (you can visit vertices outside of the current SCC)
(If counterexample is given, both marks for explanation will be awarded)

Common mistakes:

1. Missing explanation on why the algorithm will undercount the number of SCCs.
2. Wrong counterexamples. Some counterexamples are also overly complicated. In fact, the simplest counterexample you can give is just a graph with 3 vertices. Eg. 1 -> 2, 2 -> 1, 1 -> 3 has 2 SCCs, but the algorithm will output 1.
3. Correct counterexamples with wrong accompanying explanation.
4. Examples that illustrate that Kosaraju's works, but not that the modified algorithm is wrong.

14. There is no complete binary tree with more than 1 node (where the keys in all nodes are unique) that satisfies both min heap property and BST property.

    True. The smallest complete binary tree considered is 2 nodes. Thus it must be the root and a left child. If it satisfies the min heap property, then the left child must be bigger than the root and so it cannot also satisfy the BST property. Now for any complete binary tree with more than 2 nodes, it must definitely have a root and a left child of the root and already these 2 do not satisfy BST property thus no complete binary tree with more than 2 nodes satisfy BST property. So the statement is true.

Marking scheme:

(note that cbt = complete binary tree)

2 marks for correct answer
1 mark for explaining why the min heap / bst property cannot coexist for two nodes
1 mark for explaining the general case
(If the general case is explained directly, both marks for explanation will be awarded)

Common mistakes:

1. Only restating definitions of the properties, but not explaining the answer based on cbt. No marks are awarded for explanation for such submissions
2. Stopping at the 2 nodes example and not extending it to the general case (that for any cbt, the left child cannot satisfy both properties at the same time)
3. Not explaining the general case correctly.
4. Confusion about the definition of cbt. If wrong definition is used or implied, no marks are awarded for explanation. Eg. proving for perfect binary trees instead cbt
5. Mixing up definition of binary tree, full binary tree, complete binary tree, perfect binary tree and binary search tree.
6. A significant number of mistakes in definitions and properties. For some of the obvious typographical errors (missing or substituted) where the explanations are correct and make sense, benefit of doubt is given.

Comments:

A very common answer is to talk about general trees, and why the BST and min heap properties will contradict each other, **when there is a left child**. Without ensuring that there is a left child, then this statement becomes irrelevant because I can just have a right-sided tree which will satisfy both properties. This is why you need to bring in the definition of cbt since a cbt with more than 1 node ensures that there **must be a left child**.

There is a lot of confusion between the definitions of the different types of trees. Students should ensure that they are clear on the definitions, especially since most of these definitions are covered already in CS1231S. Even if there can be confusion between the different conventions (eg. complete vs almost complete) depending on the reference, these should be clarified if there are doubts. (In fact, the definition is also stated in the lecture notes, so there should not be confusion about the convention being used.)

Application Questions

This section has 5 questions and is worth 54 marks.

Write in **pseudo-code**.

Any algorithm/data structure/data structure operation not taught in CS2040S must be described, there must be no black boxes.

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

15. **[8 marks]** The UFDS data structure as taught in CS2040S is a static data structure since there are no operations to insert or remove items/disjoint sets from the UFDS.

    Consider a UFDS that has the following 2 additional operations:

    1. An **insert()** operation which will insert a new 1 item disjoint set into the UFDS where the item is labelled as **N**, where **N** is the number of items in the UFDS before the insert operation (thus the first item is labelled 0). This should run in amortized **O(1)** time.

    2. A **removeSet(i)** operation which will remove the disjoint set that item **i** (assume **i** is always a valid item) belongs to and form a new one item disjoint set for each item in the disjoint set that i belongs to (e.g removeSet(0), if 0,1,2,3 belongs to the same disjoint, then that disjoint set is removed and 4 disjoint sets will be created one each for 0,1,2,3 respectively). If i is the only item in the disjoint set it belongs to, then nothing is done. This should run in **O(N*α(N))** time where **N** is the number of items in the same disjoint set as i, and **α(N)** is the inverse ackermann function.

    Implement such a UFDS giving the algorithms for **insert()** and **removeSet(i)** in the required time complexities.

    The other operations - findSet(i), isSameSet(i,j) and UnionSet(i,j) should still run in **α(N)** time. If any of them needs to be modified to work please state the modifications

1. For the insert() operation, simply keep a count on the number of items N. If N > size of UFDS array p, then use the doubling strategy of resizing the array (ala resizing of the array in arraylist), and the new item is at index N of p and set p[N] = N. This will result in amortized O(1) for insertion.

2. For the removeSet(i) operation do the following:

i. Let rep = findSet(i)

ii. Let q be a queue

iii. for j from 0 to N-1 // N is the number of items
if findSet(j) == rep // time is O(α(N))
q.offer(j)

iv. while ! q.empty() // O(N) time
p[q.peek()] = q.poll()

total time is O(N*(α(N)))

The other operations do not need any modification and will still run in α(N) time

An alternative approach for removeSet that does not use a queue requires you to instead "preprocess" the UFDS by calling findSet on every element in the UFDS to force path compression. This allows you to have only trees of height 1, ensuring that when you do the comparison findSet(j) == rep, j will never have descendants unless j is the root itself (j == rep). In fact, you can just directly check parent[j] == rep.

Marking scheme:

2 marks for correctness of insert

- 2 marks for correct algorithm
- 1 mark for minor flaws in algorithm
- 0 marks for incorrect algorithm

2 marks for correctness of removeSet

- 2 marks for correct algorithm
- 1 marks for minor flaws in algorithm
- 0 marks for incorrect algorithm

1 mark for correctness findSet, isSameSet, UnionSet

- 1 mark if all operations are still correct
- 0 marks if any operations become incorrect

3 marks for efficiency of solution

- +1 mark for insert in amortised O(1) time

- +1 mark for removeSet in O(N * α(N)) time
- +1 mark for findSet, isSameSet, UnionSet still in O(α(N)) time
- No marks are awarded for the operation if the time complexity of each operation is not the above worst-case time complexities
- No marks are awarded for insert unless at least 1 mark is awarded for correctness
- No marks are awarded for removeSet unless at least 1 mark is awarded for correctness
- No marks are awarded for findSet, isSameSet, UnionSet unlerss 1 mark is awarded for correctness

Template to copy:
insert: 2
insert time: 1
removeSet: 2
removeSet time: 1
others: 1
others time: 1

Common Mistakes:
1. Assuming you can directly loop over the elements in the set containing i.
2. Not finding the representative set of i in removeSet. (**no set rep of i**)
3. Assuming all sets are trees of only height 1. (**only parent check**)
4. Assuming if parent[i] == i, then the set containing i is a singleton set. (**fail for descendants**)
5. Not considering that an element might still have descendants before setting p[element] = element, so descendants may not be correctly separated into singleton sets. (**fail for descendants**)
6. Using a HashMap to replace the parent array. This will achieve amortised O(1) for insert (because a HashMap can achieve O(1) amortised time for inserts only), but the other operations will not achieve the required time complexity.
7. Some students misunderstand the question and “delete” the elements in the set rather than “separate” them into singleton sets.
8. Not modifying insert with a resizable array, and just assume that it will work by incrementing a counter and / or assigning values. These solutions are not valid.
9. Using an array of a very large size, but still immutable.
10. Using a resizable array, but only increasing the size of the array by a fixed constant (usually 1). This does not allow you to achieve O(1) amortised time.

Comments:
We accept solutions that mention replacing the array by a resizable array / list / arraylist / vector, without explaining the doubling strategy.
Some submissions that wrote out the steps in java or very detailed pseudocode did not consider the initialization values of the resizable arrays (or are just wrong). No marks are deducted if the idea of using resizable arrays is clear.

16. **[12 marks]** A standard Priority Queue has the **offer(k)** and **poll()** operation where **offer(k)** will insert an item with priority **k** into the PQ, and **poll()** will remove the item with the highest priority from the PQ.

    Now implement a new PQ with an additional **pollLowest()** operation which will remove the item with the lowest priority from the PQ. In this new PQ, **offer(k)**, **poll()** and **pollLowest()** should all run in worst case **O(logN)** time where **N** is the total number of items in the PQ.

    You can only make use of/modify the standard binary heaps to implement this new PQ and no other data structures. If you are modifying the standard binary heap, state your modifications.

    Instead of 1 heap, we use 2 heaps. One is a max heap H, the other a min heap H'. An item is always added to both heaps. Each node has an additional attribute called the twin index t, which is the index of its twin node (node with the same value) in the other heap.

    Updating twin indices: when a node is shifted up or down, the twin indices of the node and the swapped node and their twin nodes need to be updated. The ShiftUp and ShiftDown method will have to be modified to take care of updating the twin indices. For example if i and par(i) are swapped (whether it is due to shiftUp or shiftDown) in the max heap H (Let A be the 1-based compact array used):

    H'.A[H.A[i].t].t = par(i)
    H'.A[H.A[par(i)].t].t = i
    temp = H.A[i].t
    H.A[i].t = H.A[par(i)].t
    H.A[par(i)].t = temp

    Perform similar update for swapping in the min heap H'.

    This does not affect the complexity of ShiftUp and ShiftDown methods, since updating the twin indices takes a constant time per ShiftUp/ShiftDown and by extension every other methods used in the heaps.

    offer(k): Insert k into both heaps. Make sure twin pointer is set. Perform ShiftUp on each heap. Time needed is still O(log N), as ShiftUp is only called twice, once on each heap.

    poll()/ pollLowest() is simply extractMax/extractMin respectively: Go to the corresponding heap (min heap if extractMin and max heap if extractMax) and perform the operation (this will cause a ShiftDown to be performed). Now we follow the twin index in the other heap to delete the twin node. Note that in the other heap, the twin node must be a leaf (otherwise the min or max heap property is violated), replace this twin node by by the last leaf if it is not already the last leaf and decrement heap size. Now we only need to perform ShiftUp, on the replaced node. Time taken is bounded by O(log N) since ShiftDown and ShiftUp are called once each.

Marking scheme:

O(logN) for all operations: 12 marks
Mistakes:
1. Use twin pointer (index or reference), but no details on how to maintain the pointer during shiftup and shiftdown. -3 marks

2. No details on how to delete the item from the other heap during extractMin/extractMax: -2 marks

O(N) for any one of the operations: 5 marks (iterate through the leaves of a max heap to find node with smallest priority and remove it).
Mistakes:
1. never replace leaf with min priority with last heap and call shiftUp: -2 marks

2. No details on how to delete the leaf from other heap: -2 marks

O(NlogN) for any one of the operations: 3 marks

Solutions with major mistakes/missing details
1. Only mention using a min heap and max heap without any details: 4 marks

2. Use min heap H and max heap H' but only H'.extractMax for poll and H.extractMin for pollLowest (i.e did not remove the twin node in the other heap whenever extractMax or extractMin is called) or no details given on how to delete from the other heap: 6 marks

Comment: Quite a lot of students who gave this answer say that items extracted from max heap and min heap will not overlap until there is only 1 item left, thus no extra work needs to be done. This is true if the PQ is only populated at the start and no more insertions are made. However this is not correct if insertions can be intermixed with poll() and pollLowest(). For example, after a series of insertions, pollLowest() is called. Let the item polled be called x (x is removed from min heap but not removed from max heap). Now another series of insertions are made where all the items have priority smaller than x. Now a series of poll() is called, until the point where x is now the largest in max heap. poll() is called again, and x will be removed from max heap and returned (since there is more than 1 item left in both heaps). This is not correct since x has already been removed from the PQ.

3. delete and return the last leaf (node at index heapsize) of a max binary heap for pollLowest(): 2 marks
This is not correct since the node with the minimum priority can be any of the leaves in the max heap and not necessarily the last leaf: 2 marks

4. Using a max/min heap where pollLowest only correct for the 1st call (keep track of index of node with smallest priority during insert) but after pollLowest, the new node with the smallest priority is not updated/updated wrongly so future calls to pollLowest is not correct: 3 marks

Other wrong solutions: 1 to 2 marks

17. **[12 marks]** Given an AVL tree **A** where each node stores a non-repeated integer key **k** (T is sorted by the key **k**) and a possibly repeated positive integer value **v**, give an algorithm for the following operation:

LargestValueInRange(a,b): Return the largest value found in the nodes in T that have keys within the range a to b (both a and b are included) where a ≤ b. Assume that there are always nodes with keys within the range a to b.

Your algorithm should run in worst case **O(logN)** time where **N** is number of nodes in the AVL tree. The insertion and deletion operation should still run in worst case **O(logN)** time.
If you need to modify the AVL tree, and the insertion and deletion operation in order to make **LargestValueInRange(a,b)** run in the required time complexity, state your modifications.

1. Modify A so that each vertex maintains a new attribute maxV which is the largest value found in all nodes in the subtree tree rooted by the vertex.

2. Modify AVL insertion of a new key value pair <k',v'> with the additional update as follows:
   i. Once the insertion point is found create a new node with attributes k = k', v = v' and maxV = v.

   ii. For each node T from parent of inserted node back to the root, compare with T' the node before T along the path back to the root: if (T.maxV < T'.maxV) T.maxV = T'.maxV

   iii. If rebalancing happens at T. For left rotation about node T. Let Q be the right child of T. Update T.maxV and Q.maxV by checking maxV of their children after the rotation. Do similar update for right rotation.

3. For deletion given a key k, perform AVL deletion with the additional update as follows:
   i. For each node T from parent of deleted node back to the root
      if T has no more children then T.maxV = T.v
      else compare T.v with maxV of its children and update accordingly.

   ii. If rebalancing happens, then update in the same way as described for insertion.

4. LargestValueInRange(a,b):
   i. if a == b, search for node T where T.k == a and return T.k <- takes O(logN) time

   ii. if a != b perform a search until a node T where a <= T.k < b or a < T.k <= b. <- takes O(logN) time
      Let answer = T.v
      a. If a < T.k, start from T.left, search for node with key == a. For each node T' encountered along the way, if T'.k >= a, Let answer = max(answer,T'.v,T'.right.maxV) if right child of T' exist else answer = max(answer,T'.v). <- takes O(logN) time
      b. If b > T.k, start from T.right, search for node with key == a. For each node T' encountered along the way, if T'.k <= a, Let answer = max(answer,T'.v,T'.left.maxV) if left child of T' exist else answer = max(answer,T'.v). <- takes O(logN) time

   iii. return answer

Insertion and deletion (Step 2 & 3) is still O(logN), and LargestValueInRange(a,b) (step 4) is O(logN).

Marking scheme:

O(logN) -> 12 marks
O(k) for k nodes in range -> 8 marks
O(N) regardless of number of nodes in range -> 6 marks
O(k log k) for k nodes in range (worst case O(N log N) if all nodes in range) -> 4 marks
Time complexity dependent on keys O(b-a) or O((b-a) log N) -> 2 marks

Solutions with errors
- Having a properly augmented tree, found LCA and updates max properly but includes excess nodes/misses nodes in O(log N) time --> 9
- O(N) traversal but fails to consider nodes that require moving out of range then back in -> 3 marks
- Confusing k with v for each node -> 2 marks
- No explanation of how to "in order traversal from a to b" assumes first an in-order traversal over whole tree and then only processing nodes within [a, b] -> 6 marks

-1 mark for each minor mistake

18. **[10 marks]** John is currently in the hospital after breaking his leg in a car accident and is wheelchair bound. Being bored of the hospital food, he wants to get to the canteen area to buy food.

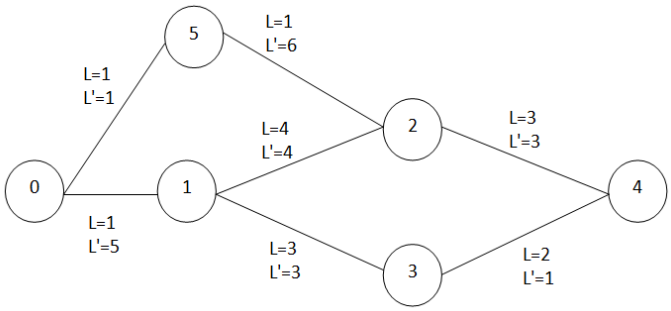
    Being in a wheelchair, getting from his ward to the canteen will pose a certain degree of difficulty. Modelling each junction/room/area of the hospital as a vertex and corridors/paths/stairs connecting them as edges, the hospital can be modelled as a graph. Now edges in the graph have 2 attributes, **L** (length of the corridor/path/stair in integer meters) and **L'** (navigation difficulty as an integer value ≥ 0).

    If the edge is a stair then L = length of the stairs and L' > 0 and represent the difficulty in navigating the stairs in a wheelchair.
    If the edge is a corridor or path then L = length of the corridor/path and L' = 0 since it is on level ground and does not pose a problem for a wheelchair.

    Since John is in a wheelchair, he wants to make sure the navigation difficulty of any path he takes is as small as possible (*clarified during exam to mean that the maximum difficulty of the corridors/paths/stairs in the path is as small as possible). This is a valid path for John. Moreover, he wants to make sure that the valid path is also one that gives the shortest distance from his ward to the canteen.

    Given a graph **G** of the hospital as modelled above, **S** the starting vertex which is John's ward and **S'** the destination vertex which is the canteen, give an algorithm to find the required path as described above. The algorithm should run in worst case time ≤ **O(ElogE)** where **E** is the number of edges in **G**.

    In the example given below, if S = 0 and S' = 4, then the valid path with the shortest distance from S to S' is 0,1,3,4.



1. First run Kruskal's/Prim's on the G. Now run DFS from the vertex S to vertex S' and find the largest edge along the path. Let the weight of this edge be M. Any valid path from S to S' cannot have an edge larger than w. <- O(ElogE) time.

2. Now run modified Djikstra on G starting from source vertex S. The modification is that for the current vertex v being processed (most up to entry for that v being dequeued from the PQ), perform relaxation only on each neighbor u, where w(v,u) <= M. Ignore all edges to neighbors with weight larger than M. The shortest path from S to S' is the required path. <- O(ElogE) time to run modified Djisktra, and O(V) time to get the path from S to S' from the predecessor array.

Total time = O(ElogE)

Since quite a lot of students did not see the announcement on the clarification to the question and treat a valid path as the SP from S to S' based on L' as edge weight. I will allow this alternative interpretation. So now the question will be become one where you need to find among all SPs from S to S' based on L' as edge weight, a SP from S to S' based on L as edge weight. The following is a solution which also has time complexity O(ElogE).

1. have a distance array D for L and distance array D' for L'. Now using modified Dijkstra with source vertex S, modify the entry in the PQ to store a triplet (d,d',v) where d' == D'[v] and d = D[v] when triplet was enqueued

. 2. PQ is keyed on d' then ties broken using d.

3. Whenever the most up to date entry of a vertex v is dequeued from the PQ, scan through all it neighbors u and modify the relaxation as follows:

```
if (d'[u] == d'[v]+ (v,u).L')
    if (d[u] > d[v]+(v,u).L) // if an alternative valid path is found use the one which result in the
                             // smaller cost based on L
        d[u] = d[v]+(v,u).L
        p[u] = v
        PQ.offer((d'[u],d[u],u))
else if (d'[u] > d'[v] + (v,u).L')
    d'[u] = d'[v] + (v,u).L'
    d[u] = d[v] + (v,u).L // d[u] must be updated if there is a SP based on L' that is found to u.
    p[u] = v
    PQ.offer((d'[u],d[u],u))
```

This modification to Dijkstra does not change it's time complexity which is O(ElogE) (since no -ve edges).
After Dijkstra ends, get the shortest valid path from S to S' using the predecessor array p in O(V) time.

Marking Scheme:

based on 1st interpretation

O(ElogE) solution: 10 marks

Wrong solutions:
1. Get the largest edge along the SP path instead of finding the largest edge along the minimax path: 5 marks

2. Only find MST of the graph (based on L' as edge weight) and return the minimax path from S to S': 3 marks

3. Get MST based on L' then run Dijkstra on the MST to get SP. This is same as above, since it simply return the same minimax path in the MST and is not necessarily the shortest minimax path (based on L) among all minimax paths from S to S': 3 marks

Comment: Quite a lot of students think that Prim's or Kruskal's will find all the alternate minimax path from S to S'. This is not true MST is a tree thus there will only be 1 minimax path given from S to S'.

Other wrong solutions: 1 to 2 marks depending on how wrong.

-1 mark per minor error.

based on 2nd interpretation

O(ElogE) solution: 10 marks

Mistakes:

1. Invert the priority of L and L', i.e relax based on L then if find alternate cost path based on L, check L': -3 marks

2. no/vague/totally wrong details given for the modification to the relaxation of edges (especially case where d[u] have to be updated if a better SP to u has been found based on L'): -3 marks

3. PQ entry stores the wrong information (e.g store weight of edge instead of SP cost): -3 marks

4. Forget to modify the entry into the PQ into a triplet as mentioned in the solution: -2 marks

5. Forget to have another distance array based on L' or L: -1 mark

6. Forget to mention that PQ is keyed on difficulty then ties broken using distance: -1 mark

7. If details of modification to relaxation is given then each minor error in modification to relaxation of edges will be deducted -1 mark

O(VE) solution (use Bellman ford instead of Dijkstra): 5 marks
Mistakes:
1. No details given on how bellman ford will find SP based on all possible valid paths:-2 marks

O(V^3) solution (using Floyd warshall): 3 marks

Wrong solution:

1. Simply use original/modified Dijkstra without any modification to find SP either based on L or L': 2 marks

2. For other wrong solutions (especially other interpretations of the question): 1 to 2 marks depending on how wrong.

19. **[12 marks]** In the future, teleportation over short distances has been invented. Now if the distance between 2 points is short enough instantaneous travel is possible
using 0 time. For larger distances multiple teleportations are required and takes time of 1, 2 and 3 seconds (At most 3 seconds is required even for points which are furthest apart).

Now you are given **N**, the number of points (they are labelled from 0 to N-1) in the galaxy and **M** descriptions, where **M** = **O(N)**, consisting each of a triplet (x,y,w) where x and y are 2 points and w is the time required to teleport between the 2 points (w can only be 0,1,2 or 3), i.e from x to y or from y to x. You are sure that there is always a way to from any point to any other point among the N points.

You are also given a starting point s and an ending point s', and you are required to find the smallest possible time to get from s to s'.

Model this as graph problem (state what your vertices, how they are linked and what are the weight on the edges), and give an algorithm to solve the problem in the best possible worst case time complexity you can think of.

State any modifications to algorithms and/or data structures taught in the course if you are modifying them to solve the problem.

1. Model the problem as an undirected graph G as follows:
   i. Each of the N point is a vertex
   ii. Each of the M descriptions is a undirected edge linking the a pair of vertices x and y

2. Now transform G into another graph G' as follows:
   i. for each edge (x',y') which has weight > 1, do the following
      a. If w(x',y') == 2, add an intermediate vertex v between x' and y' thus we have x'-v-y'.
         Let w(x',v') and w(v,y') be 1
      a. If w(x',y') == 3, add 2 intermediate vertices v and v' between x' and y' thus we have x'-v-v'-y'.
         Let the w(x',v),w(v,v')and w(v',y') be 1

G' simply replaces edge weights > 1 with multiple edges of weight 1 that add up to the same original edge weight and thus G' can replace G. The number of vertices in the G' is < N+2M and the number of edges in G' is < 3M.

3. Now G' only consists of edges of weight 0 and 1. Now run a modified BFS as follows:
   i. Change the queue in BFS to a double ended queue to allow enqueuing to both the back of the queue and the front of the queue. Assuming tailed linked list implementation of the queue, enqueuing an item to the front of the queue (call it offerFront(item)) is simply performing an insertion to the head of the tailed linked list.

      cost of offerFront() is thus O(1) time, same as offer()

   ii. Let q be the double-ended queue. Now run BFS from s, by first performing q.offer(s)

iii. Loop while q is not empty and dequeue v from front of q.
    for each neighbor u of v where D[u] = inf // each unvisited neighbor
    if w(v,u) == 0
      q.offerFront(u)
      D[u] = D[v] // no change in distance since weight of edge is 0
    else
      q.offer(u)
      D[u] = D[v]+1 // add 1 to distance since weight of edge is 1

iv. return D[s'].

total time is O((N+2M)+3M) = O(N) since M = O(N).

The idea of the algorithm in step 3 is that any 0 cost edge will not affect the SP cost at all not matter how many of such edges are used, thus they should be explored first to try to get to the destination (by putting them to the front of the queue so they are dequeued first), while the cost 1 edges are put to the back (same as for BFS on weight 1 graph).

~~~~~~

A O(N*alpha(N)) solution is to transform the graph as given in the first solution then make a further transformation by merging all vertices connected by 0 weighted edges into a super vertex, so that finally the graph only has weight 1 edges. Solution is as follows:

1. create UFDS F of size N where each vertex is an item in F.
2. Go through all edges (the M description) and for each edge (u,v) where w(u,v) == 0 perform F.unionSet(u,v). <- O(N*alpha(N) time)
3. Now go through all edges again <- O(N) time
    i. remove all 0 weight edges
    ii. for all non-0 weight edges(u,v) replace u with findSet(u) and v with findSet(v).

4. Now with the updated descriptions, construct a weight 1 graph as described in the 1st solution using an adjacency list <- O(N)

5. run SSSP BFS for unweighed graph on the constructed graph as described in solution <- O(N)

total time O(N*alpha(N)) time
~~~~~~

Marking Scheme:
O(N) solution: 12 marks
Mistakes:
1. Incorrect use of double ended queue to solve the problem everything else being correct: -5 marks
2. correct modelling of the graph and transformed graph but totally wrong algo on this graph to find SSSP: -7 marks

O(N*alpha(N)) solution: 10 marks
Mistakes:
1. algorithm stating the use of UFDS in merging but missing details or incorrect algorithm: -6 marks
2. merely state to merge vertices of 0 edges but no details on the algorithm, not even mention of using UFDS (everything else being correct): -7 marks

O(NlogN) solution: use Djikstra on the original graph: 7 marks
$O(N^2)$ solution: Djikstra using edgelist or adjacency mat: 5 marks
$O(N^3)$ solution: Floyd Warshall on original graph: 3 marks

Deduction of marks for other errors:
1. consider DFS along with BFS as possible algo to find SSSP (after correct modelling): -5 marks

Wrong solution:
1. Only modelling of original graph is correct: 2 marks

2. Try to transform the original graph but done wrongly (e.g Change weight 0 edges to weight 1 and before converting the other edges to weight 1 edges and running BFS) -> 3 marks

Comment: This is a common mistake thinking that weight 0 and weight 1 are equivalent. This is not true, since no matter how many edges a path with all edges of weight 0 have, it will still have 0 cost. On the other hand, a path with all weight 1 edges will have cost = number of edges in the path. Thus BFS cannot be used directly on a graph with edges of weight 0 and 1.