CS2040 2023/2024 Sem2 Final Assessment – Sample Solution

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

1. You are given the following sequences of operations for an AVL tree:
   1. Insert 1, insert 2, insert 3, insert 4, insert 5, insert 6
   2. Insert 5, insert 4, insert 3, delete 5, insert 1, insert 2
   3. Insert 3, insert 1, insert 2, insert 4, delete 1, insert 5
   4. Insert 2, insert 3, insert 4, insert 1, insert 5, delete 3

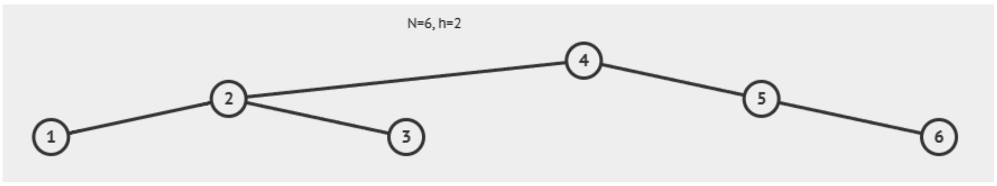
   Which sequence will result in a complete binary tree? Recall that complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes of the last level are as far left as possible.

a. 1
b. 2
c. 3
d. 4
e. None of the above

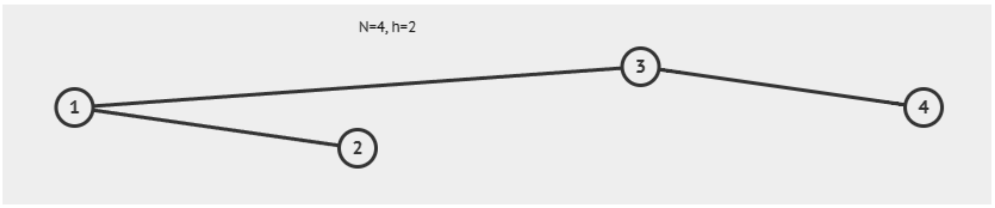
Ans: d

Simply simulate the insertion/deletion.

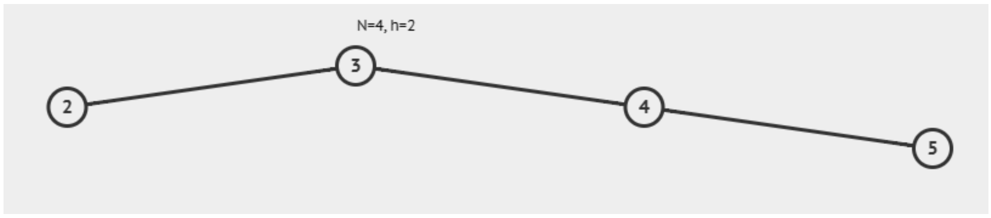
1. Insert 1, insert 2, insert 3, insert 4, insert 5, insert 6



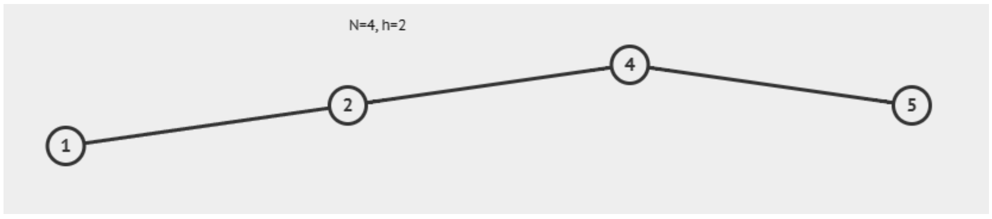
2. Insert 5, insert 4, insert 3, delete 5, insert 1, insert 2



3. Insert 3, insert 1, insert 2, insert 4, delete 1, insert 5



4. Insert 2, insert 3, insert 4, insert 1, insert 5, delete 3



2. Alice, Bob, and Eve are tasked to construct a valid binary search tree (which may or may not be balanced) using all integers from 1 to 1000. After they were done, they were asked to search for an integer X. These are the sequences of nodes visited by each of them:

- Alice	: 2, 55, 100, 289, 900, 809, 700, X
- Bob	: 987, 678, 123, 654, 234, 345, 543, X
- Eve	: 1, 50, 137, 217, 398, 976, 497, X

Which of the following is NOT a possible value of X?

a. 653
b. 499
c. 500
d. 397
e. None of the above

Ans: d

Property of a binary search tree: for every vertex x and y, y.key < x.key if y is in the left subtree of x and y.key is > x.key if y is in the right subtree.
If we encounter a value X and we go to the right, then we are not considering all values less than X. In other words, all subsequent values must be greater than X. Conversely, if we encounter a value X and we go to the left, then all subsequent values must be less than X.
After going through the sequence, we will get the following inequalities:

- Alice: 289 < X < 809
- Bob: 345 < X < 654
- Eve 398 < X < 976

Taking the intersection, X must be between 399 and 653 (inclusive).

3. A 1-based compact array is used to implement a maximum heap. If the heap contains 35 distinct elements, what is the smallest possible index of the second smallest element?

a. 9
b. 32
c. 18
d. 17
e. 16

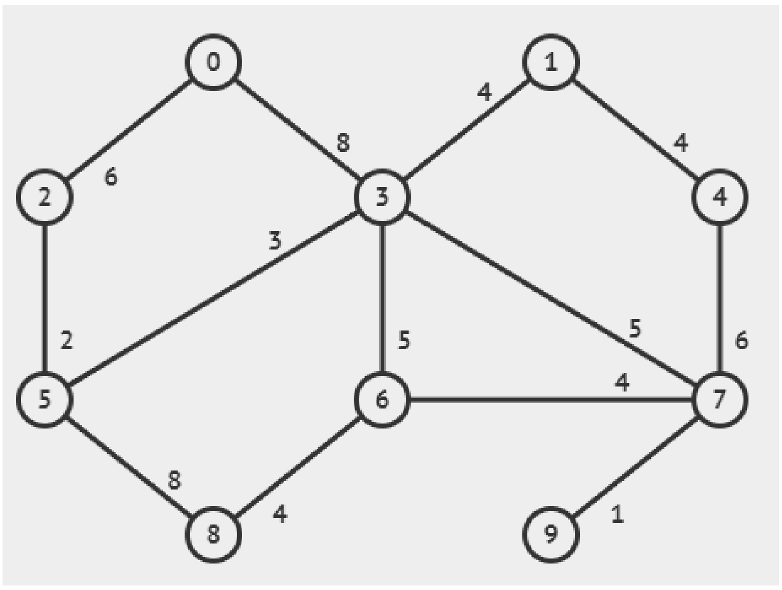
Ans: c

Property of a maximum heap: for all i except the root, A[parent(i)] >= A[i].
Since heap is a complete binary tree, we only have two cases:

- If N is odd -> all nodes have either 0 or 2 children.
    - 2nd smallest element must be a leaf node. If not, then it will have two children -> there are two other elements with value less than it. Contradiction.
    - Therefore, smallest index = leftmost leaf node, which is floor(N / 2) + 1
- If N is even -> all nodes have either 0 or 2 children, except for one node (with one child).
    - 2nd smallest element is either a leaf node or the node with one child

For the node with one child, its child must be the Nth node since it's a complete binary tree. The index of that node = N / 2 (smallest index)

4. For this question and the following 2 questions, refer to the graph below:



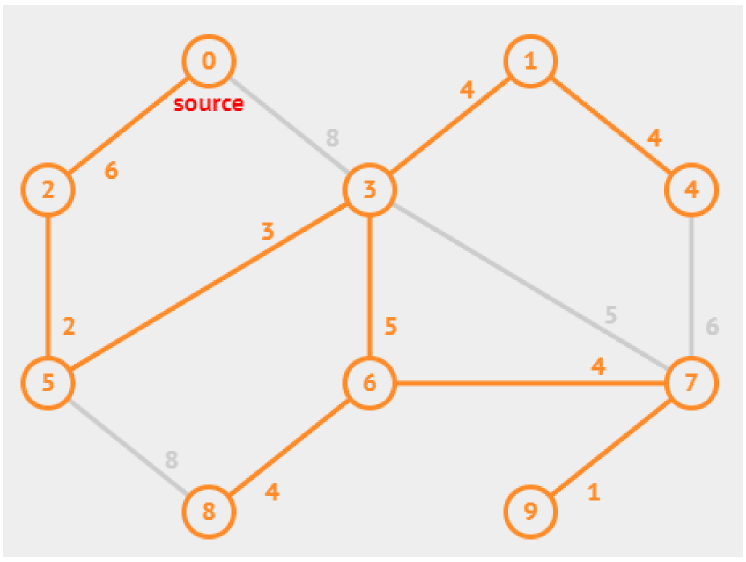
What is the shortest distance between vertex 0 and vertex 9 while ensuring the maximum edge weight along the path from 0 to 9 is minimized?
Note: It is more important to first ensure the maximum edge weight along the path from 0 to 9 is minimized.

a. 14
b. 16
c. 17
d. 21
e. 33

Ans: c
The final selected path should be a minimax path based on the 1st criteria and among all minimax paths it needs to be the shortest.
There are many possible minimax paths from 0 to 9, one of them is 0-2-5-3-6-7-9 which is the path from 0 to 9 when you construct the MST of the graph (here the maximum edge is 0-2 with weight 6) as shown in the diagram below. The total cost of this path is 6+2+3+5+4+1 = 21

However, this is not the shortest possible minimax path. The shortest possible is the one where you go directly to 3-7 instead of 3-6-7, i.e it is the path 0-2-5-3-7-9 with total cost = 6+2+3+5+1 = 17.

In this case replacing the edge 3-6 with 3-7 results in an alternative MST, however in some cases the shortest possible minimax path might not even be part of any MST. For example, if the edges along the path 3-1-4-7 and 3-6-7 are all weight 4, no MST will have the edge 3-7 which is of weight 5, but the shortest minimax path from 0 to 9 will use this edge.



5. If you run Original Dijkstra's algorithm from source vertex 0, what is the maximum number of entries in the priority queue at any one point in time?

a. 1
b. 2
c. 3
d. 4
e. 5
f. 6
g. 7
h. 8
i. 9
j. 10

Ans: j

Initialize PQ with 10 entries including source vertex, and it will decrease overtime, since you would update the remaining entries during relaxations and not add any new entries.

6. If you run Modified Dijkstra's algorithm from source vertex 0, what is the maximum number of entries in the priority queue at any one point in time?

a. 1
b. 2
c. 3
d. 4
e. 5
f. 6
g. 7
h. 8
i. 9
j. 10

Ans: d

Sequence of priority queue state with entries (w, v)
Initialize PQ: (0, 0)
After relaxing outgoing edges of vertex 0: (6, 2), (8, 3)
After relaxing outgoing edges of vertex 2: (8, 3), (8, 5)
After relaxing outgoing edges of vertex 3: (8, 5), (12, 1), (13, 6), (13, 7)
After relaxing outgoing edges of vertex 5: (12, 1), (13, 6), (13, 7), (16, 8)
After relaxing outgoing edges of vertex 1: (13, 6), (13, 7), (16, 4), (16, 8)
After relaxing outgoing edges of vertex 6: (13, 7), (16, 4), (16, 8)
After relaxing outgoing edges of vertex 7: (14, 9), (16, 4), (16, 8)
After relaxing outgoing edges of vertex 9: (16, 4), (16, 8)
After relaxing outgoing edges of vertex 4: (16, 8)
After relaxing outgoing edges of vertex 8: Empty PQ

7. Given 100 disjoint sets each of rank 2, what is the maximum possible height that can be obtained by calling $unionSet(i, j)$ operations assuming that only the union-by-rank heuristic is used?

a. 5
b. 6
c. 7
d. 8
e. None of the above

Ans: d

This question is similar to the Q1 of Tutorial 6. The difference is we are given sets each of rank 2. The rank increases only if two sets of equal rank are combined the resulting set will be of one rank higher than the original set. When 50 ($\left\lfloor \frac{n}{2} \right\rfloor, where\ n = 100$) pairs of sets each of rank 2 are combined, we will get 50 disjoint sets of rank 3. Similarly, when 25 pairs of disjoint sets of rank 3 are combined, we will get 25 disjoint sets of rank 4. Continuing this way, we would end up with one set of rank 8.

8. Let G be an undirected graph which is stored in an adjacency matrix. The diagonal elements of its adjacency matrix are zero and the non-diagonal elements are k, where $k \geq 1$ and k is a constant. Assuming that the number of vertices in G are n, identify the statement which is correct about G:

a. Graph G has only one MST with a total sum of weight of edges = $(n - 1)$
b. Graph G has only one MST with a total sum of weight of edges = $k(n - 1)$
c. Graph G has multiple MSTs each with a total sum of weight of edges = $k(n - 1)$
d. Graph G has multiple MSTs each with a total sum of weight of edges = $(n - 1)$
e. Graph G has multiple MSTs each with a unique total sum of weight of edges

Ans: c

Diagonal elements of adjacency matrix are zero, i.e., no self-loops. Off-diagonal elements are k, i.e., it's a fully connected graph with equal weight k across all edges. The graph would have multiple MSTs as the edge weights are not unique. Total sum of weights of edges in the MST would be $k(n - 1)$ as there would be $n - 1$ edges in the MST each of weight k.

9. Let the simple and undirected graph G contains edges with distinct weights. Consider the following statements related to this graph.

   Statement 1: If the graph G contains at least one cycle and if the lowest weighted edge in the cycle is E_{min}, then every MST of this graph must include the edge E_{min}.

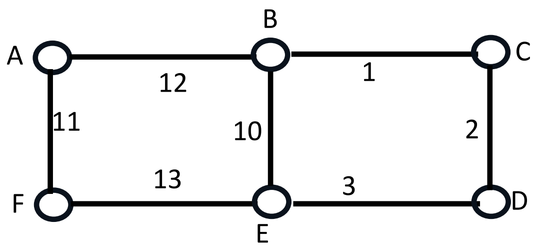
   Statement 2: If the graph G contains at least one cycle and if the highest weighted edge in the cycle is E_{max}, then every MST of this graph must exclude the edge E_{max}.

   Select the appropriate answer.

a. Both Statements are wrong
b. **Statement 1** is wrong while the **Statement 2** is correct
c. **Statement 1** is correct while the **Statement 2** is wrong
d. Both Statements are correct

Ans: b

If a graph contains a cycle, we must always delete the largest weighted edge by cycle property. On the other hand, there is no guarantee that we can retain lowest weighted edge in the cycle. An example is below:



Focus on the two cycles: A-B-E-F-A and B-C-D-E-B
Edge (B,E) is the largest weighted edge in the cycle B-C-D-E-B and it’s also the least weighted edge in the cycle A-B-E-F-A.
Edge (F,E) is the largest weighted edge in the cycle A-B-E-F-A.

We need to remove edges (B,E) and (F,E). Hence, **Statement 1** is wrong while the **Statement 2** is correct.

10. A directed weighted simple (no self-loops, no multiple edges) graph **G** contains 4 vertices and 6 edges. The edges have weights 1, 2, 3, 4, 5, 6, but you don't know which edge each weight belongs to. You are also given the **distance matrix after Floyd-Warshall APSP algorithm has been run** on **G**. The matrix finally looks like:

```
0  1  3  5
∞  0  4  ?
∞  ∞  0  ?
∞  ∞  ∞  0
```

where ∞ is a value representing "infinity", and '?' represents some positive value which is unknown because you forgot to write it down.
How many different possible graphs of **G** could there be?

a. 0
b. 1
c. 2
d. 3
e. 4
f. a number between 5 to 6 inclusive
g. a number between 7 to 12 inclusive
h. a number between 13 to 24 inclusive
i. a number between 25 to 720 inclusive
j. a number above 720

Ans: c

The output shows that the 6 edges are:
0->1, 0->2, 0->3,
1->2, 1->3, and
2->3
There is only one possible path from:
0->1 hence its weight must be 1 from D[0][1],
1->2 hence its weight must be 4 D[1][2],
2->3 but its weight is unknown from D[2][3]
There are 2 paths from 0 to 2:
0->1->2 which has distance 5 == 1+4
0->2 must then have distance 3 as the shortest distance D[0][2] == 3
At this point 3 out of the 6 edges have known weights, with unassigned weights {2, 5, 6}
As D[0][3] == 5:
1->3 cannot have weight 2 otherwise 0->1->3 has distance 1+2 (3 < 5, contradiction)
0->3 cannot have weight 2 (2 < 5, contradiction)
therefore 2->3 must have weight 2 so that 0->2->3 has distance 3+2

Since we have already found a shortest path from 0 to 3, now 0->3 and 0->1->3 can have distances ≥ 5. Therefore edges in {0->3, 1->3} can have any of weights in {5, 6}.
Drawing the graph out first, and then filling in weights you know for sure and ticking off shortest distances that you have fulfilled, should help you greatly.

Analysis: 18 marks, 4 questions (4 to 5 marks each)

11. **[4 marks] Claim:** Given any directed weighted graph **G**, running original Dijkstra's on **G** from any given source vertex **s** will allow us to detect if there is a negative weighted cycle in **G**. This is because if there exists at least 1 negative cycle, we will discover at some point in the running of the algorithm that there is a vertex **v** which has already been removed from the PQ but original Dijkstra's is still able to relax it. So we can modify original Dijkstra's to check that when it needs to update an entry in the PQ, and the entry cannot be found, it means there is a negative cycle.

    Note: For the original Dijkstra's algorithm, whenever an entry (**d**,**v**) is dequeued from the PQ, **v** will only attempt to relax its outgoing edges if **d** != infinity.

    Choose all options which are correct.

    a. True because every vertex involved in any of the negative cycle(s) can be relaxed indefinitely.

    b. True because vertices reachable from any of the negative cycle(s) can be relaxed indefinitely.

    c. False because the negative cycle(s) may not even be reachable from the given source vertex s

    d. False because we cannot differentiate whether this is caused by a negative cycle or just negative weighted edges with no negative cycle.

    Ans: c & d

    c is obvious since if the negative cycle is not reachable from s vertices/edges will never be relaxed, thus those vertices v will have their entries (d,v) dequeued with d = infinity, and this will not cause any relaxation as stated.

    d is correct since a graph with no negative cycle but only negative weights can also cause a vertex which has been dequeued from the PQ to be relaxed again (refer to the example in slide 26 of lecture notes 16).

12. **[4 marks]** You are given a weighted connected undirected graph **G** containing **V** vertices and |**E**| edges (|**E**| ≥ **V**), as well as a non-empty set of edges **S** ⊂ **E**. NO cycle can be formed by using only the edges in **S**. You are to form a "Minimum" Spanning Tree on **G** that **MUST contain ALL** edges in **S** first.

    This means that you are to find a Spanning Tree **T1** of **G** that contains all edges in **S**, such that there will **NOT exist** any other Spanning Tree **T2** of **G** for which **T2** contains all edges in **S**, and yet **T2** has a lower sum of edge weights than that of **T1**.

    You are given 2 algorithms **P** and **K** which are adapted from Prim's and Kruskal's MST algorithm respectively. Changes to the standard algorithm are <u>underlined</u>. Assume each edge has attributes u, v, w representing the 2 vertices at the ends of the edge and the weight of the edge respectively.

    Algorithm **P**: Run **Prim's** algorithm with a slight modification in O(|**E**| log **V**) time to produce the result **T1**

```
Build adjacency list AL of G using V and E
Initialize the resultant "M"ST T1 with all edges in S
Initialize an array M of length V to mark already visited vertices, all initially false
Initialize an initially empty PQ of edges ordered by (edge.w, edge.v) in ascending order

For each edge e in S
  mark both e.u and e.v as true (visited) in M
  for each other edge o (e.u, o.v, o.w) also incident on e.u, add (o.w, o.v) into the PQ
  for each other edge o (e.v, o.v, o.w) also incident on e.v, add (o.w, o.v) into the PQ

While the PQ is not empty
  e = PQ.removeMin()
  If M[e.v] then continue // skip e as it is NOT in the cut set any more
  M[e.v] = true // subsequently will not be in cut set
  T1.add(e) // confirm e in the "M"ST
  For each neighbour edge e' in AL[e.v] // neighboring edges incident on e.v
    PQ.enqueue((e'.w, e'.v)) //all neighbors of v can now be in cut set
```

    Algorithm **K**: Run **Kruskal's** algorithm with a slight modification in O(|**E**| log **V**) time to produce the result **T1**

```
Sort the edge lists E and S ordered by (edge.w, edge.u, edge.v) in ascending order
Create a UFDS U of V elements, and for each edge e in S perform U.unionset(e.u, e.v)
Initialize the resultant "M"ST T1 with all edges in S

For each edge e in E
  If U.isSameSet(e.u, e.v) then continue // skip e as it will form a cycle
  U.unionSet(e.u, e.v)
  T1.add(e)
```

Claim: Both **P** and **K** can find the "Minimum" Spanning Tree that **MUST contain ALL** edges in **S** first, as defined above

Select the option that has the correct answer to the claim, and has the best explanation

a. True. Both **P** and **K** work as they are both "M"ST algorithms
b. False. Only **P** works, as **K** may not properly keep track of visited vertices
c. False. Only **K** works, as **P** may not properly keep track of edges along the cut
d. False. Both **P** and **K** do not work to solve this problem

Ans: c

Multi-source Prim's algo works for building a Forest that includes all vertices (as in the powering cities tutorial question), aside from the edges in **S**. However, it does not work well for this question where a Spanning Tree is required. **V**-1 edges should be selected by the end of the algorithm.

Using the cut property with Prim's algorithm to find an MST, there is only 1 vertex on one side of the cut initially, and each time an edge is picked one more vertex moves over to the other side of the cut. Therefore **V** -1 edges will be picked when all vertices end up on one side of the cut.

In this question, after first picking |**S**| edges which may involve up to 2|**S**| vertices, there may initially be more than |**S**|+1 vertices on one side of the cut. If that happens, the standard part of the algorithm will go on to pick less than **V**-|**S**|-1 edges. Combined with the initial edges in **S**, less than **V**-1 edges will be picked, so **T1** may not be an MST. This modification of **Prim's algo does NOT work** for this question

Kruskal's algorithm works as the algorithm does not use whether a vertex is visited or not to determine if it is on one or the other side of the cut. Rather, cycles are avoided using checking whether both vertices have the same representative in the UFDS, i.e. findSet(u) and findSet(v) within isSameSet(u, v). Therefore, as long as the UFDS is kept accurate, there will not be a problem picking edges all over the graph while building the Spanning Tree. **Modification of Kruskal's algo works** for this question.

Questions 13 and 14 refer to the following problem:

13. Mr. Zak wants to implement a Friends network among a group of N persons which is modeled as a graph and stored in an adjacency list. The vertices are labeled from 1 to N and each person is mapped to only one of these vertices. Two vertices i and j are connected through an undirected and unweighted edge if the persons corresponding to the vertices i and j are friends.

    2 operations are required by Mr Zak:

    1. The Friends network is dynamic, i.e., the users i and j can friend/unfriend, resulting
    in the insertion/deletion of the edge between the vertices corresponding to persons i and j, so a corresponding operation called $change(i, j, x)$ is required where i and j are the two users and x can be either 0 or 1, where x = 0 means i and j will unfriend each other if they are already friends and x = 1 means i and j will friend each other if they are not already friends.

    2. To check whether i and j are directly/indirectly connected, Mr Zak requires another operation $connected(i, j)$ that will answer this by returning 0 if i and j are not directly or indirectly connected, 1 if i and j are directly connected by an edge (a path of 1 edge), and 2 if i and j are indirectly connected by a path with > 1 edge and not connected by path with 1 edge.

Mr Zak wants to make use of a UFDS (as taught in CS2040) to create disjoint sets of vertices such that all vertices that corresponds to either directly or indirectly connected users are placed in a single set.
He will create this UFDS by going through all edges (x,y) in the adjacency list of the input graph and calling the UnionSet(x,y) operation to create this UFDS.

Claim: Using only the above created UFDS (without any augmentation to the UFDS and no other DSes, not even the input adjacency list) and operations on the UFDS (as taught in CS2040) Mr Zak believes that he can implement the 2 operations $change(i, j, x)$ and $connected(i, j)$ in $O(\alpha(N))$ time.

[2 Marks] The claim is correct.

a. True
b. False

Ans: False

14. **[3 Marks]** Give the rationale for your answer to the previous question.

Ans:

There are multiple reasons for the claim being **False**.

If the Friends network is static, i.e., if there is no unfriend option, then the UFDS can created as a preprocessing step. The UFDS can be used to group the friends that are connected either directly or indirectly. However, the friends network varies with time. For example, we need to delete an edge from the graph if a person unfriends another person. UFDS as taught in CS2040 does not support this delete operation.

Also, just deleting an edge is not enough to remove a vertex from the disjoint it belongs. All edges adjacent to that vertex needs to be removed, before a vertex is disconnected from the disjoint set it belongs to. The number of edges adjacent to a vertex is only obtained from the adjacency list (which we cannot use).

Moreover, just using the UFDS we cannot determine if 2 vertices in the same disjoint set are directly connected (connect by an edge) or indirectly connected (connected by a path with more than 1 edge). We have to make use of the adjacency list in order to get such information.

Grading Scheme:

1 **3 marks**: As long as the student state either:

- UFDS does not support the un-union of two sets for change(); OR
- UFDS is unable to handle when exactly a person is going to be disconnected from the set after unfriending; OR
- UFDS is unable to support checking if two person are directly or indirectly friends

2 **0 marks**: If the answer ONLY:

- Considered only friending, but did not consider unfriending i.e. change(i, j, 0); OR
- Did not consider unfriending may not result in a disjoint where it initially belongs;
- (C) Did not consider checking for directly / indirectly connected friends;
- (D) Misunderstood the question by assuming usage of additional / other data structures;

If the answer is the combination of *1* and *2*: will get **3 marks**.

Some answers are incomplete or ambiguous but are on the right track (e.g. "change() cannot be done by only using unionSet() and findSet() operations in the UFDS"), will get **1 mark**.

Some answers assumed no usage of heuristics used in UFDS such as path-compression (most commonly implied through answers in the form of "achieving connected(i,j) through isSameSet(i,j)"), these will get **1 mark** IF the solution also revolves around solving correctness of operations by explicitly stating that the 2 heuristics are removed, but no marks will be awarded for assuming standard UFDS does not have path-compression heuristic.

A substantial number of answers involved the initial adjacency list, which is not what the question is asking. A few also mentioned about running Floyd-Warshall, an algorithm that doesn't run on UFDS nor adjacency list.

Questions 15 to 16 refer to the following problem

15. Given a non-empty BST containing unique integer values, a new value Y (also unique from current values in the BST) which is just bigger than value X in the BST and just smaller than value Z in the BST is inserted (X and Z are guaranteed to exist in the BST).

    Note 1: This is a BST and not a balanced BST.
    Note 2: The ancestor of Y is any node along the path from Y to the root of the tree.

    [2 marks] Claim: X and Z will both be ancestors of Y in the BST after it is inserted.

    a. True
    b. False

    Ans: True

16. **[3 Marks]** Give your rationale for your answer to the previous question.

    Ans: The caim is true.

    This is because if Y is just bigger than X and just smaller than Z, it must means before Y is inserted X is just smaller than Z and thus X is the predecessor of Z:

    Thus there are only 2 possible arrangement of X and Z in the BST before the insertion of Y:

```
    Z
   /
  .  ← the . simply means there is a path
 .
X  ← X is the largest in the left subtree of Z
```

    or

```
  X
   \
    .
     .
      Z ← Z is the smallest in the right subtree of X.
```

Once Y is inserted, there are also only 2 possible arrangements:

```
    Z
   /
  .
 .
X
 \
  .
   .
    Y  ← Y is the smallest in the right subtree of X (any other position is wrong
    given
         the current arrangement of X and Z since Y is just larger than X and just
         smaller than Z)
```

or

```
  X
   \
    .
     .
     Z
    /
   .
  .
 Y ← Y is the largest in the left subtree of Z (any other position is wrong given the
     current arrangement of X and Z since Y is just larger than X and just smaller
     than Z)
```

As you can see either ways, X and Z must be ancestors of Y.

Grading Scheme:

3 marks – To get full marks, students can choose to explain one of the following:

X and Z must be in the same subtree

- Since Y is just bigger/smaller than X/Z, there cannot be a value that is greater than X and less than Z prior to the insertion of Y. Otherwise, that value will be the element that is just bigger/smaller than Y.
- This implies that there cannot be a node that can split X and Z into two different subtrees (i.e. a node that have X as its left child and Z as its right child).
- Since Y is just bigger/smaller than X/Z, it will be inserted to the subtree that contains both X and Z.
- As Y will be inserted as a leaf node, both X and Z will be an ancestor of Y.
-

Property of successor and predecessor / insertion

- In a BST, a node will be inserted as a leaf node.
- Since the newly inserted value Y has no child, its successor is either:
    - its parent (if Y is a left child), or
    - its “first right ancestor” (if Y is a right child)
- Similarly, since Y has no child, its predecessor is either:
    - its parent (if Y is a right child), or
    - its “first left ancestor” (if Y is a left child)
- It’s clear that both X and Z, which are Y’s predecessor and successor, are ancestors of Y.

2 marks – incomplete/erroneous proofs that include:

- Mentioned the relationship between X and Z.
- Only considered 1 scenario, e.g. X must be the parent of Z and Z must be the parent of Y.
- Mentioned that X is Y’s predecessor and Z is Y’s successor and Y is a leaf node.

1 mark:

- Gave example to show it’s possible X and Z are Y’s ancestor instead of proving the claim.
- Any answers that mention some elements of the proofs above but lacking in coherence.

Common mistakes

- Giving an example where a node splits X and Z into two different subtrees.
- Arguing that Y will be the parent of X and/or Z.
    - Giving an example where X or Z is the one being inserted last.
    - Assumed that the tree is a self-rebalancing binary search tree.
- Wrong definition of ancestor or mixing up the concept of predecessor/successor. For instance, saying that Z is a successor and not an ancestor.
- Confusing BST with binary heap, mentioning shifting up/down.
- Wrong understanding of BST property:
    - Stating that Y must be X’s right child, and Z must be Y’s right child.
    - Stating that Y must be the parent of both X and Z because $X < Y < Z$.
    - Giving an example where a left child is greater than the root, or where the right child is smaller than the root.
- Wrong assumption about the ordering of X, Y, and Z. Some assumed $Y < X < Z$, others assumed $X < Z < Y$.

- Stating that the statement is true if X is Z ancestor or vice versa and false otherwise.
- Stating that Y can be “on the same level” as X or Z.
- Stating that Y must be “lower” than X and Z because there is no rotation.
- Showing that it’s possible for both X and Z to be ancestor of Y instead of proving that the claim is true for all possible cases.

Structured Questions: 4 questions

This section is worth 42 marks. Answer all questions.

Write in **pseudo-code** for the written questions.

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. **[6 marks]**

City Z has **A** places numbered 0 to (**A**-1) inclusive, as well as **R** roads ($0.1\mathbf{A}^2 < \mathbf{R} < 0.2\mathbf{A}^2$) which each join two different places together. Moving from one place **a** to another place **b** consumes energy $\mathbf{E_{ab}}$ (a positive floating-point number) which is also the same as $\mathbf{E_{ba}}$. It is guaranteed that the roads are laid out such that it is possible to travel between any pair of places.

Jerry has a vehicle which he uses to move around city Z. Jerry's vehicle can have a battery of capacity **C** (a positive floating-point which Jerry is unsure of), and the battery is **always fully charged at place 0** and **always fully recharged** (will have **C** units of energy) **at every place before Jerry moves** along a road. As Jerry moves from one place **a** to another place **b**, the energy consumed along a road drains the battery by that amount, i.e. new energy in the battery at place **b** = old energy at place **a** - $\mathbf{E_{ab}}$ before being recharged.

To be safe, Jerry wants to ensure that the battery never goes below half its capacity at any time. Help Jerry figure out, for every place **p** in 1 to **A**-1 inclusive, the battery capacity $\mathbf{C_p}$ required such that he can travel between place 0 and place **p** with battery never going below half capacity.

At least one of these 4 options outlines part of the high-level approach that solves the problem correctly. Which one of these is the best option, that is correct and most efficient?

a. Model this as a graph problem. Find the topological ordering of the graph and run one-pass Bellman-Ford so that the SSSP from place 0 to everywhere else is computed.
b. Model this as a graph problem. Run Bellman-Ford algorithm so that the SSSP from place 0 to everywhere else is computed.
c. Model this as a graph problem. Run Prim's algorithm so that the MST is computed, but use the variant (as described in lecture) that does **NOT** use a Priority Queue.
d. Model this as a graph problem. Run Kruskal's algorithm so that the MST is computed.

Ans: c

The problem description shows that we can model a dense connected undirected weighted graph where vertices are places and roads are edges.
Since the battery is recharged at each place, the best path to take from vertex 0 to another place should minimize the maximum edge along the path. Therefore this is a minimax path problem.

The sum of the weights along the path does not matter at all, so this is NOT an SSSP problem.
One possible minimax path from (0, place b) can be found along the path from 0 to b on the MST of the graph.

Since the graph is dense, running Prim's algorithm WITHOUT a PQ will run in O($\mathbf{A}^2$) == O(V^2) time, while running Kruskal's algorithm will take O($\mathbf{A}^2$ log **A**) == O(E log E) time.

18. **[9 Marks]**
John has created an unweighted directed graph and saved it in a text file. The format of the text file is the familiar way in which a graph is stored.
That is, the first line contains 2 integers "V E" representing the number of vertices and the number of edges respectively.
Following that will be E lines, each containing 2 integers "X Y" which represent the edge X$\rightarrow$Y.

John is certain he has created at least 1 cycle (simple or not) that includes all the vertices in it. However, the next day he checks his file and realizes that there are only E-1 lines after the 1st line, meaning that there is actually 1 edge missing.

Assuming the graph is already read into an adjacency list AL, help John out by giving the most efficient algorithm you can think of in terms of worst case time complexity to detect if there is still at least 1 cycle (simple or not) that includes all the vertices.

Ans:

If there is a cycle that connects all the vertices it means there is only 1 SCC in the graph. So just run Kosaraju's algorithm to count number of SCCs in the graph. If it is 1 then there must be such a cycle else there is no such cycle. Time complexity is O(V+E).

Grading Scheme:

Correct Solutions:

9 marks: O(V+E) using Kosaraju's algorithm or something similar: 9 marks

6 marks: O(V^2) by running DFS/BFS from each vertex as source and check if all other vertices are reachable

Incorrect solutions:

4 marks: *use Kosaraju's to detect if number of SCC < number of vertices, if so there is a cycle that includes all vertices*. This is not correct since this only means there is a cycle and not that the cycle includes all vertices in it.

3 marks: other solutions that detect the presence of a cycle but not whether there is a cycle that includes all vertices in it.

2 marks: solutions which are only able to detect presence of cycle in undirected graph (e.g counting component algo and not Kosaraju's algo) and not directed graph.

1 mark: Other attempted but wrong solution or incomplete solution.

Deductions for mistakes:
-1 mark for mistakes or important steps in the solution with no details (is vague).

1. The marks your solution will get based on the category it falls into will be usually be given in the comments.
2. The deductions for mistakes will also be given in the comments.

Your total marks for this question will be the sum of 1. and 2.

19. **[15 marks]**
The management of a busy shopping center has plans to renovate and upgrade their parking lot.
In this upgraded parking lot, each parking space is numbered from 1 to X (X > 1000000 .. yes it is a super huge shopping center ...).
Cars coming into the parking lot will be given a parking space which is the **smallest numbered parking space that is free**.
Cars are identified by their license number which is a string of length 8 and unique.
To implement this upgraded parking lot, the management has asked you to write a program with the following 3 operations:

1.) **assign(c)**: car c (c is the string containing the license number of the car) is coming into the car park, so assign it to the smallest numbered parking space that is currently free and return that parking space number.
If there is no empty space, do nothing and simply return -1.

2.) **free(c)**: car c (c is the string containing the license number of the car) is moving out of the parking lot, so free its parking space.

3.) **init(x)**: Initalize the carpark with x parking spaces (x is guaranteed to be > 1000000) which are all free.

Using what you have learned in CS2040, determine the data structure(s) required to implement **assign(c)** and **free(c)** in worst case ≤ O(log x) time and **init(x)** in worst case ≤ O(x) time and give the algorithm for all 3 operations.

Ans:

Use both a PQ (min heap) H and an AVL T:

H will contain as entries the numbers of the current free parking spaces.
T will contain as entries a pair (x,y) where x is the license number of a car and y is the parking space it occupies. T is ordered by x.

```
init(x)
 use fast heap create to create H initialized with 1 to x // O(x) time
 Create an empty AVL T // O(1) time

assign(c)
  if H is not empty
    y = H.extractMin() // O(log x) time since H is at most size x
    T.insert((c,y)) // O(log x) time since T is also at most size x
    return y
  else
    return -1
free(c)
  p = T.search(c)  // find and return the entry in T where license number is c. O(log x) time since
                   // T is at most size x
  H.insert(p.y) // since parking space p.y is now empty insert it into H. O(log x) time since H is
                // at most size x
```

*can also replace the heap with another AVL BUT you have to give details on how to insert 1 to x into the AVL in O(x) time and not O(x log x) time to get full marks.

Grading Scheme:

init(x): total 5 marks for correct O(x) solution
assign(c): total 5 marks for correct O(log x) or less solution
free(c): total 5 marks for correct O(log x) or less solution

Deductions:
- 1 mark each for **assign** and **free** if the solution is O(x)
- 2 marks each for **assign** and **free** if the solution is O(x log x)
- 1 mark for initialization of AVL tree in O(x log x)

-1 mark for unclear logic and ambiguous reasoning for time complexity.

Analysis of student solutions:

- Majority of the students identified min heap to store the free parking slots and hash table to store the assigned parking slot along with the Hash table. Hashing without collision is not possible for the car license number that contains only 8 characters. It takes O(x) in worst case for assign and free.

- Some students suggested to use parking slot as the key for Hash table. License number is the input argument to free function. To free the parking slot, we need to search for the license number in hash table values. It takes O(x) in worst case.

- Single AVL tree is used to store parking slot and car license number. The tree is ordered by parking slot number. If the tree is ordered by parking slot number, then free(c) takes O(x) in worst case.

- Some students suggested BST for storing car license info along with assigned parking slot. Searching a BST is O(x) in worst case.

- Arrays/Linked list/double linked list to store parking slot numbers and license information. The worst case complexity is O(x).

- UFDS to store the occupied parking slots with a separate logic to track the minimum free parking slot and the logic to implement free function. However, the reasoning for worst case time complexity is lacking.

- Naïve UFDS without the logic to implement free function. Removing an element from disjoint sets is not feasible. So this is not correct answer.

20. **[12 marks]**

Tom is playing a game in which he needs to score points. You are NOT concerned with how he scores points, but are concerned with calculating the penalty he gets. **Penalty** is an integer which starts off at 1, can increase or stay the same, but **can NEVER decrease**. Obviously, less penalty is desired.

To decide on Tom's penalty, Tom's character may transition through various penalty states based on the penalty state his character is currently in, as well as the action Tom performs. There are **S** such penalty states numbered 0 to (**S**-1) inclusive, and **T** transitions (**S** < **T** < 4**S**) between penalty states in total. Tom starts off having a penalty of 1 at penalty state 0, and can end at any penalty state i.e. all penalty states are reachable from penalty state 0.

When Tom's character transitions from one penalty state to another, Tom's penalty is **multiplied by a positive real number > 1** which we will call the **multiplicative factor**. It turns out that this multiplicative factor will always be one of {$\mathbf{c^2}$, $\mathbf{c^3}$, $\mathbf{c^5}$} for some positive real number **c** > 1 (c itself is not given only the multiplicative factor is given). For example, if Tom's character undergoes 4 successive transitions where the multiplicative factors are 4, 8, 32, 4 respectively (here we see that **c** = 2 but it is NOT given), then the incurred penalty will be 4096 == 2^{12}, since the penalty progresses as follows: 1 -> 4 -> 32 -> 1024 -> 4096.

You are given the integers **S**, **D** (0 ≤ **D** < **S**), as well as a list **L** that contains **T** transitions in the form (u, v, w), u and v being the 2 penalty states and w is the multiplicative factor to get from u to v.
Your task is to give the most efficient algorithm in terms of the worst case time complexity to **find the lowest possible penalty** Tom's character can have incurred **at penalty state D** starting out from **penalty state 0**.

For example, if inputs are:

S = 6, **D** = 2, (not given **T** = 8, **C** = 2)

```
L = [   (0, 3, 4),
        (0, 5, 32),
        (1, 2, 32),
        (1, 4, 4),
        (3, 1, 8),
        (4, 1, 32),
        (4, 2, 4),
        (5, 4, 32)]
```

then the result should be 512, by:

moving from penalty states 0 -> 3 -> 1 -> 4 -> 2
with 4 transitions where the multiplicative factors are 4, 8, 4, 4 respectively.
causing the penalty to progress as follows: 1 -> 4 -> 32 -> 128 -> 512

IF you model this as a graph problem, do clearly state:

- what the vertices and edges are
- what important characteristics your graph has

- what graph representation to use
- the algorithm you use, which variant (if there are multiple), and any modifications you may make especially if any of these affect correctness or time complexity (otherwise the worse will be assumed)

Ans:

Each penalty state is a vertex, each transition is an edge. Build a **directed weighted** adjacency list **AL** out of **L**. The graph will be a sparse graph. This is an SSSP problem. The result is the shortest distance from vertex 0 to vertex **D**.
There are 2 general approaches to handle distance being edge weights multiplied together instead of summed:

A – Replace addition with multiplication during edge relaxation

B – $\log_c$-transform all edge weights, i.e. edge weights are just the power, which will be one of 2, 3, or 5, and finally the result is $\mathbf{c}^{\text{distance}}$ instead of just distance

Solution 1: Run Bellman-Ford(0). Takes O($\mathbf{S}^2$) time

Solution 2: Run Dijkstra(0) dense-graph variant which does NOT use a PQ as in Tutorial 11. Takes O($\mathbf{S}^2$) time unfortunately

Solution 3: Run Dijkstra(0) which uses a PQ and lazy update (or TreeSet as a PQ). Takes O(**S** log **S**) time. Dijkstra's algorithm (modified with either A or B) works because the distance along a path can NEVER decrease. That means when Dijkstra's algorithm picks a vertex with the minimum distance estimate among all unvisited vertices, that picked vertex will NEVER need to be visited again

However, the best algorithm runs in O(**S**) time:
Since powers can only be 2, 3 or 5, split each edge (u, v, $\mathbf{c}^{\mathbf{x}}$) into **x** edges of multiplicative weight **c**, creating **x**-1 dummy vertices in between u and v. Then run BFS(0), where relaxation multiplies distance by **c** to an unvisited neighbour, and still read off dists[**D**]. Takes O(**S**) time as |V| < 5**S** and |E| < 5**T** < 20**S**

Equivalently, if approach B is used instead of approach A, split each edge (u, v, w) into **w** unweighted edges, creating **w**-1 dummy vertices in between u and v. Run the standard BFS_SSSP(0) algorithm and the result will be $c^{\text{dists}[\mathbf{D}]}$

Regardless of solution, if approach B is chosen, a non-trivial part of the problem is to first determine the value of **c**. This can be done in O(**S**) time by adding all weights encountered in **L** to a set **W**, which will contain up to 3 elements. Sort **W** into a list **Y**. Finally:

```
If Y has 2 elements and pow(smaller element, 5) == pow(larger element, 3)
    then c is cubeRoot(smallest element)
else c is sqrt(smallest element)
```

Grading Scheme:

If the answer is completely correct (regardless of efficiency), these will be given:

- O(S) is capped at **12 marks**
- O(S^2 log S) or worse time capped at **5 marks**
- O(S^2) time capped at **7 marks**, or capped at **6 marks** without correct modification / with no modification
- O(S log S) time capped at **9 marks**

Otherwise, the 3 components below are used to score partial marks while the **cap above still applies**:

1. Graph modelling (capped at 6 marks)

Note: Marks are awarded for this component if either explicitly mentioned, or can clearly be inferred by details in the algorithm (but ignoring the algorithm name)

Vertices and edges (capped at 2 marks)
Penalty states are vertices
Transitions are edges

Graph characteristics (capped at 2 marks)
Directed
Weighted (or unweighted if properly done)
Cyclic
Sparse
(Weakly-)connected

Graph representation (up to 1 mark)
Adj list (since graph is sparse and we're enumerating neighbours frequently)
(For Bellman-Ford, Edge List is also alright)

Nature of problem (up to 2 marks)
SSSP - 2 marks
MSSP/APSP - 1 mark
MST - 0 marks for this section

2. Correct algorithm name for the correct problem is worth **2 marks**. Less efficient but correct algorithms like Bellman-Ford, Floyd-Warshall will get **1 mark**

3. Modification to allow multiplicative effect of edge weights is worth **4 marks**. Will only get **2 marks** if student assume the value of c is given rather than computing it themselves from the weights given.

Common Mistakes

- Claiming the graph is acylic since the SPST graph is acyclic – While an SPST is indeed acylic, the input graph can have cycles, as seen from 1 -> 4 -> 1 in the sample input. The characteristics of the input graph to the algorithm affects it, one-pass-bellman-ford generally won't work on cylic input graphs

- Claiming the graph is undirected – The question says that Tom's character may take an edge (transition) based on the current vertex (penalty state his character is in) as well as the action Tom performs. There is no indication that Tom can take the same action at the other end of the edge (u, v) to move back to vertex u

- Claiming that the graph is dense since **S** < **T** – This just shows that the graph will definitely NOT be a tree if you ignore edge direction. In fact, the graph is sparse since **T** is in O(**S**) as **T** < 4**S**. On the other hand, a dense graph would be O($\mathbf{S}^2$) or close to that

- Thinking that dijkstra's algo runs equally well on a general edge list as compared to an adjacency list

- Assigning penalty to vertices / edges brute-forced even before running some algorithm – There could be many possible penalties at a vertex, and the resulting penalty on vertex v after taking an edge (u,v) is dependent on the penalty at vertex u. It is very unlikely that just a linear pass through all edges / all vertices on the original graph will give the min penalty at all vertices from vertex 0

- Claiming that taking fewest number of edges from 0 to **D** results in the lowest penalty

  Counter-example: -$\mathbf{c}^2$-$\mathbf{c}^2$- is better than -$\mathbf{c}^5$- but the latter has fewer number of edges

- Claiming that this is a minimax (minimum maximum) path problem

  Counter-example: -$\mathbf{c}^3$- is better than -$\mathbf{c}^2$-$\mathbf{c}^2$- under multiplication

- Using addition (or not mentioning modification of edge relaxation to handle multiplication) in a SSSP algo, then retracing path using predecessor array from **D** to 0 using multiplication

  Counter-example: -3^3- is better than -3^2-3^2- under multiplication but the latter is better under addition.

  It is also worth noting that additional information needs to be stored along with the predecessor array to efficiently find the edge weights while retracing the path

- Using a global penalty variable which is multiplied whenever an edge is relaxed, instead of reading off the distances array – Some relaxed edges may be an intermediate improvement, but still not the best, so these would not be on the SPST. Also, some edges on the SPST may not lead to **D**. There is a need to multiply only edges along the shortest (multiplicative) path from 0 to **D**, not on the whole SPST

- Thinking that this is an MST problem – In order to get lowest penalty at **D**, there may be vertices in which Tom's character should NOT visit. If Tom needs to visit all vertices (penalty states) and end up with the minimum penalty at **D**, that would still NOT be an MST problem, but would be a TSP (Travelling Salesman Problem) instead. Also, MST is not defined on directed graphs

- Assuming that **c** is known – Explicitly stated in the question that **c** is NOT given

- Claiming that **S** is the source – 0 is the source, **S** is the number of vertices (penalty states)

- Claiming that there is an upper-limit of **T** transitions – *It seems that "(not given **T** = 8, **C** = 2)" in the sample test case disappeared from the question in examplify?* Even so, since **S** < **T** and the max number of edges on the SSSP from 0 to **D** is **S**-1, there is effectively no limit on the number of edges than can be taken on the SSSP, so state-space graph is not required

- Taking negative log (using base > 1) of each edge weight (multiplicative factor) – This changes all positive weights into all negative weights, and possibly introducing negative-weighted cycles. Dijkstra's algo won't work, and even if the graph is acyclic and a brute-forced algorithm is used, you'd be attempting to find the (when untransformed) highest multiplicative penalty at **D**, since you are attempting to minimize the sum of negative-log-weights along the path.