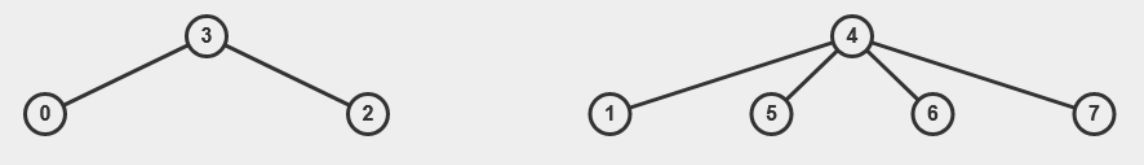
CS2040 2021/2022 Semester 2 Final

MCQ

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

Do all questions in this section.

1. You are given the following UFDS, which uses union by rank and path compression:



Initially, the UFDS consists of 8 disjoint sets of 1 item each, before a series of unionSet/findSet calls were made to form it into the UFDS above. What is the rank of the set containing item 6?

a. 1
b. 2
c. 5
d. The UFDS is invalid, or there is more than 1 possible rank for the set containing item 6

Ans:

It is not possible to determine for sure what the rank of the set containing 6 is. While it is clear that the height of this set is 1, the rank could be 1 or 2, depending on how the set was made.

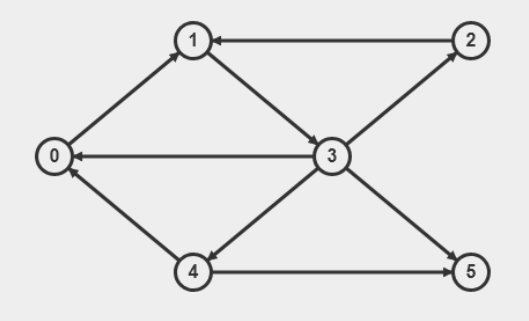
A possible sequence of events (assuming 8 disjoint sets numbered from 0-7) to create the set containing 6 (ignoring the other set on the left) with rank 1:

unionSet(1, 4), unionSet(4, 5), unionSet(4, 6), unionSet(4, 7)

A possible sequence of events to create the set containing 6 with rank 2:

unionSet(1, 4), unionSet(5, 6), unionSet(6, 4), unionSet(4, 7), findSet(5)

2. You are given the following directed graph:



You are asked to pick exactly one edge from the answers below to be removed from the graph, such that there will not be any cycles in the graph after this edge is removed. Which of the following edges should you remove?

a. 0 -> 1
b. 1 -> 3
c. 3 -> 4
d. 4 -> 0

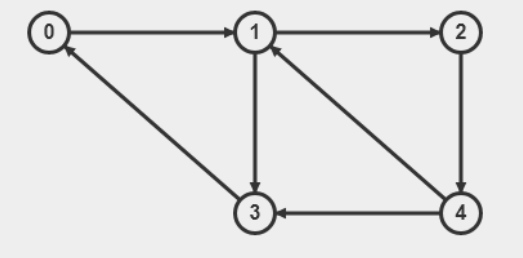
Ans:

There are 3 cycles in the graph:

(0, 1), **(1, 3)**, (3, 4), (4, 0)
(0, 1), **(1, 3)**, (3, 0)
(1, 3), (3, 2), (2, 1)

By removing the edge (1, 3), we are able to break all cycles in the graph. To further prove this, if the edge (1, 3) is removed from the graph, we can obtain the valid toposort (3, 2, 4, 0, 1, 5) for the resulting graph.

3. You are given the following directed graph:



The graph is intended to be weighted, but the actual weights of each edge is unknown. However, the following is known:

1. The correct minimum distances from vertex 0 to all other vertices are as follows:

Vertex	0	1	2	3	4
Distance	0	5	3	6	2

2. All edge weights are distinct integers in the range [-2..5] (ie. from -2 to 5, both ends inclusive)
3. No negative cycles are present in the graph.

Determine the largest possible weight that the edge 1 -> 3 can have.

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

Ans:

No negative cycles exist in the graph. Therefore, all shortest paths from vertex 0 should not involve the same vertex more than once.

We can deduce the exact weights of the following edges:

0 -> 1 has weight 5 (from dist(1))
1 -> 2 has weight -2 (from dist(2) – dist(1))
2 -> 4 has weight -1 (from dist(4) – dist(2))

The remaining edges can take on a range of different weights:

4 -> 1 must have a weight of at least 3 (otherwise a negative cycle is formed)

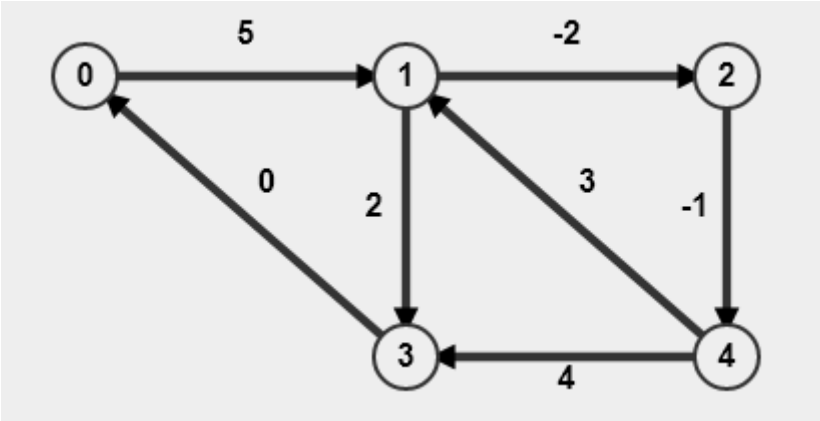
Now, depending on which vertices lie along the shortest path from 0 to 3, other edges may take on different weights:

If the shortest path is 0 -> 1 -> 3, then:

1 -> 3 must have a weight of 1

If the shortest path is 0 -> 1 -> 2 -> 4 -> 3, then:

4 -> 3 must have a weight of 4
4 -> 1 must therefore have a weight of 3
Therefore, the largest edge weight that is left is 2, which can be assigned to edge 1-> 3. One such graph is as follows:



Edge 3 -> 0 can have either weight 0 or weight 1.

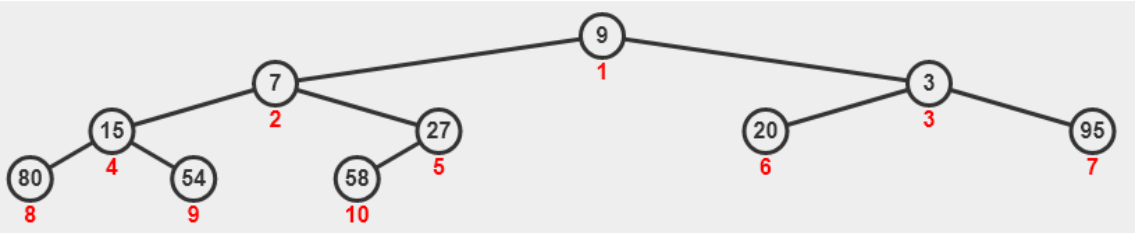
4. Given a BST (which may not be balanced) with **n** elements, we want to output all elements in the BST in sorted order. This can be done in worst case:

a. O(n) time
b. O(n log n) time
c. $O(n^2)$ time
d. $O(n^2 \log n)$ time

Ans:

Just run inorder traversal on the BST. The total time taken is O(n), and does not differ based on the height of the BST.

5. You are attempting to run an O(n) createHeap() to create a minimum binary heap. The following is how the heap looks before any shiftDown() operations are called.



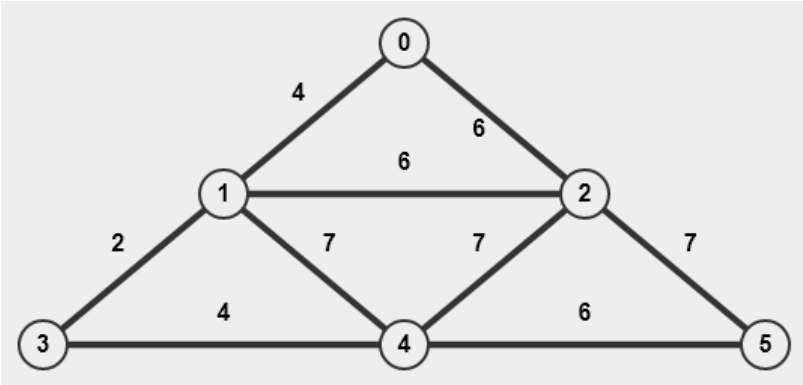
How many swaps will occur after all shiftDown() operations have been called?

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

Ans:

Only one swap will occur (9 with 3).

6. How many distinct Minimum Spanning Trees are there in the following graph?



(there are no edges with weights 5 (five) or 8 (eight) in this graph; if you are seeing this, it is most likely an edge with weight 6 (six)).

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

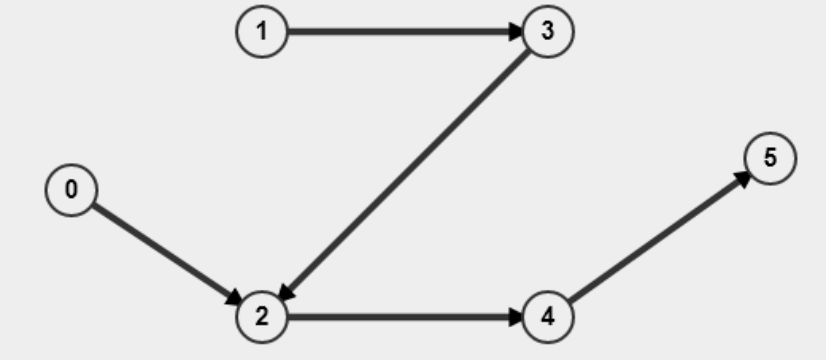
Ans:

There are two possible MSTs in this graph:

(0, 1), (1, 3), (3, 4) and (4, 5) are in both MSTs of the graph.

The final edge can be either (0, 2), or (1, 2).

7. You are given the following directed acyclic graph:



You are asked to pick exactly one edge from the answers below to be inserted into the graph. After this edge is inserted, it should result in the fewest number of valid toposorts. Which of the following edges should you insert? A cyclic graph is considered to have 0 valid toposorts for this question.

a. 3 -> 4
b. 3 -> 5
c. 0 -> 3
d. 1 -> 2

Ans:

The initial graph (before any edges are inserted) only has 3 valid toposorts:

0, 1, 3, 2, 4, 5
1, 0, 3, 2, 4, 5
1, 3, 0, 2, 4, 5

To see how many of these toposorts are still valid after inserting an edge (u -> v), we check to see if there are any toposorts where v is before u in the toposort. If so, this toposort is no longer valid. The option which removes the most toposorts in this manner is the correct answer.

3 -> 4 removes 0 toposorts.
3 -> 5 removes 0 toposorts.
0 -> 3 removes 1 toposort (1, 3, 0, 2, 4, 5)
1 -> 2 removes 0 toposorts.

Therefore, 0 -> 3 is the correct answer.

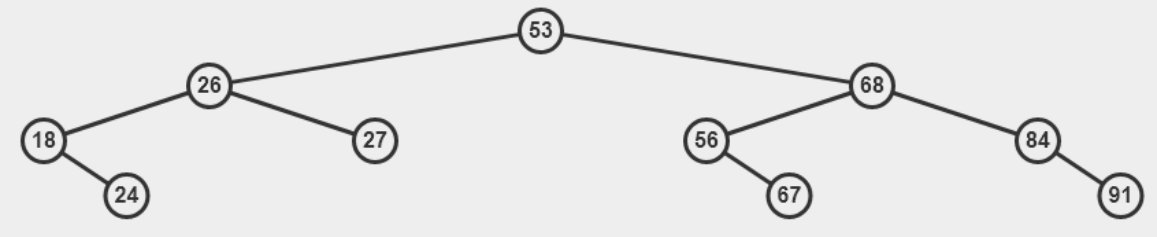
8. Given both a min binary heap and a max binary heap containing the same elements, we can obtain a sorted list of these elements in worst case:

a. O(1) time
b. O(log n) time
c. O(n) time
d. O(n log n) time

Ans:

There are no shortcuts involving a min and max heap that would allow for finding a sorted list of elements in faster than the default O(n log n) time.

9. You are given the following AVL tree:



How many of these elements would, if deleted (lecture implementation of delete) on its own, result in at least one rotation?

a. 0
b. 1
c. 2
d. 3

Ans:

Deleting either 26 or 27 would result in at least one rotation.

10. Which of the following may give incorrect SSSP results on a directed graph (having no bi-directed edges) with negative edge weights but no negative cycles?

a. Modified Dijkstra's
b. Bellman Ford's
c. Floyd Warshall's
d. Depth First Search

Ans:

DFS only gives correct results for SSSP under very specific circumstances (if the graph is a tree).

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.

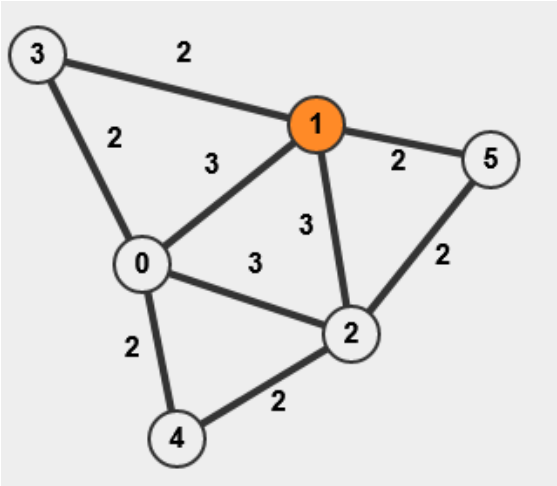
Correct explanation is worth 2 marks. Partially correct explanation worth 1 marks.

Do all questions in this section.

11. In an undirected connected graph where there exist at least 1 cycle with multiple edges having the same largest edge weight in the cycle, then there always exist multiple MSTs for the graph.

False.

It is possible that even if there are multiple edges with the largest edge weight in a cycle, all those edges are also involved in other cycles where they are the only largest edge and thus have to be removed. The following is an example, the edges of weight 3 forms a cycles but each are also involved in another cycle where they are the largest, thus they all have to be removed in the MST and so there is only 1 possible MST.



Marking Scheme:

2 marks: Giving a counterexample where all largest edges in one cycle are removed.

0 marks: Giving an incorrect counterexample (eg. a graph where more than 1 unique MST exists, or where the MST identified in the graph is incorrect).

12. For an undirected connected graph with no cycles (standard definition of cycle as given in lecture notes) but where all the edge weights are negative, running modified Dijkstra (as described in the lecture notes) on such a graph (stored in adjacency list) from a source vertex **s** will give the correct SSSP solution from **s** to every other vertex.

False.

This is because an undirected edge (i,j) is always represented in both directions from i->j and j->i in adjacency list resulting from it being indistinguishable from a directed edge from i->j and a directed edge from j->i. Thus if (i,j) is negative and reachable from the source vertex, both i->j and j->i will keep being enqueued in the PQ when running modified Dijkstra resulting lower and SP cost for i and j. Modified Dijkstra will need to be modified again to handle this kind of cases.

Marking Scheme:

2 marks: Showing that an infinite loop occurs for Modified Dijkstra's in such a graph.

1 mark: Mentioning both "infinite loop" and "exponential time" in your answer. "Exponential time" is not the same thing as "infinite loop" (see below)

0 mark: Mentioning only "exponential time" (the most common case being to refer to the example shown in lectures). The algorithm will still produce the correct result in that case, but it may take a long (but finite) time to do so.

13. We can always re-construct a BST (exact same structure) given the in-order sequence of the keys in the BST and which key is the root in the BST.

False.

Given an in-order sequence of the keys in a BST, you can possibly reconstruct multiple correct BSTs and not the BST from which the in-order sequence is obtained. An example is 1,*2,3,4 from the BST (Fig 1) below and you are given that 2 is root. However from the sequence, you can obtain either the original BST or another valid BST (Fig 2).

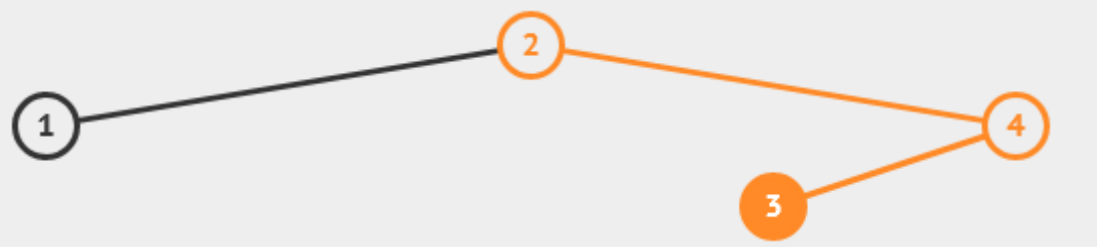


Fig 1

Fig 2

Marking Scheme:

2 marks: Showing an example where multiple valid BSTs exist for a given inorder traversal/root combination.

1 mark: Showing an example where multiple valid BSTs exist for a given inorder traversal, but the roots are different.

1 mark: Showing an example where multiple valid binary trees (not BSTs) exist for a given inorder traversal.

14. Given an unsorted array of N non-repeated integers and two integers a and b from the array, where **a** < **b** and there is at least 1 integer in the array bigger than **a** and smaller than **b**, it is possible to output the M^{th} smallest integer between **a** and **b** (excluding **a** and **b**), where $1 \leq M \leq K$ (K is the number of integers in the array bigger than **a** and smaller than **b**) in <u>worst case</u> O(N+MlogK) time.

True.

1. Scan through the array and copy all integers c, where a < c < b into another array H starting from index 1. ← O(N)
2. Perform fast heap create on H to create a min heap.← O(K)
3. Extract min M times to the M^{th} smallest integer between a and b. ← O(MlogK)

In total, time complexity = O(N+K+MlogK) = O(N+MlogK) since K <= N

Marking Scheme:

2 marks: Giving the above algorithm that runs in O(N + M log K) time .

1 mark: Giving an algorithm that runs in O(N + K log K) time.

1 mark: Giving an algorithm that runs slower than O(N + M log K) time, but has the first part of the answer done correctly (ie. filtering out integers which are > a and < b in O(N) time).

0 mark: Giving algorithms that fit into none of the above.

Some common misconceptions resulting in slow runtime complexity:

- Claiming that quickselect runs in worst case O(N) time. Quickselect runs in expected O(N) time, but worst case $O(N^2)$ time.
- Claiming that building an AVL tree runs in worst case O(N) time. This is possible if you are provided a sorted list of integers, but otherwise runs in O(N log N) time.
- Claiming that radix sort runs in worst case O(N) time. Radix sort runs in worst case O(dN) time, where d is the number of digits in the integer. For this question, it was not stated that the number of digits in the integer is fixed.
- Claiming that some other sorting algorithm (eg. quick sort, insertion sort) runs in worst case O(N) time. These algorithms run in best case O(N) time, but worst case $O(N^2)$ time.
- Claiming that merge sort can somehow be modified to run on the array containing the K elements in O(M log K) time, by stopping merge sort once the first M elements are in sorted order. There is no way to tell that the first M elements are in sorted order until the very last iteration of merging, at which point the total time taken would have been O(K log K) already.

Claiming that since the maximum value of K would be approx. N, and the maximum value of M is K, an O(N log N) algorithm would therefore be an O(N + M log K) algorithm. This is not how the time complexity of an algorithm is deduced.

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 CS2040 must be described, there must be no black boxes.

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

Note:

1. If you are using the UFDS data structure, you may assume that all operations take worst case O(1) time.

2. If you want to solve the question as a graph problem, give the graph modeling first (if the required graph has not been described in the question). That is, state what are the vertices, what are the edges and how are the vertices linked by the edges.

***Missed out in the section writeup that hashtable operations are taken as O(1) worst case**

15. John who lives in city X is planning to visit his friend Mary who lives in city Y by driving there from city X.

    Every time John has to enter a city on his way to visit Mary, he will be stopped and will be deducted half of the amount of money that is left in his "toll card" before he is allowed entry. Once the amount of money is < 1 dollar he will be rejected entry into the city. Note that since John is already from city X at the start, money will not be deducted there.

    Given a graph G of N vertices and E edges representing the road network (vertices = cities and edges = bi-directional roads linking the cities) stored in an adjacency list, give the most efficient algorithm in terms of worst case time complexity you can think to determine the minimum amount of money that John needs to have in his "toll card" in order to drive from city X to city Y to visit Mary. (Announced during exam: You may assume there is always a way to get from X to Y)
    [8 marks]

    Key observation: The fewer cities the less money is required since amount of money grows exponentially with number of cities that John pass through. Thus we simply a path that passes through the minimum number of cities to get from city X to city Y.

    1. Simply to perform BFS SP using X as source.
    2. Minimum amount of money required = $2^{D[Y]-1}$

Marking Scheme:

For correct solutions:

O(N+E) BFS = 8 marks <-- Can be modelled as an **unweighted** graph problem!

O(N^2) by using AM + BFS = 7 marks

O((N+E) log N) Dijkstra SSSP from X/Y = 6 marks

O(N^2 log N) by using AM + Dijkstra = 6 marks

O(NE) Bellman-Ford SSSP = 4 marks

Common deductions:

-1 mark for wrong power, typically 1-2 hops away

-1 mark for dijkstra which multiply instead of add, but did not specify initial money

-1 mark if mentioned finding/updating node in PQ without any explanation of how this can be done efficiently

-2 marks if only count number of edges from X to Y along SSSP

-3 marks if SSSP from X to Y but use global counter to count cities

16. Mankind in the year 4XXX has begun to colonize other planets outside the solar system. One such planet is planet Y.

Due to a recent gigantic planetquake, most of the cities were ruined. In order to rebuild, one urgent matter would be to rebuild the road network linking the cities which have also been destroyed. Since the central database containing the road network was destroyed too, the last resort was to refer to ancient physical blueprints of a proposed network of roads linking the cities.

In the blueprints, there are N cities (labeled from 0 to N-1) and M proposed roads ($N-1 \leq M \leq \frac{N(N-1)}{2}$) linking the cities such that there is always a way to get between any pair of cities via the road network (announced during exam: the roads are two way roads). Due to the way the cities are built, the proposed roads in the network are all the same length**.** Thus building each road would normally cost the same amount C (for some C > 1,000). However, because different roads go through 3 different types of terrains, the cost of the roads are multiplied by a factor of 1, 2 or 3, so cost of the roads can be C, 2C or 3C.

Given a list A of the M proposed roads where each road is expressed as a triple (x,y,z) where x and y are integers representing the 2 cities connected by the road and z is the cost of building the road, use the most appropriate DS(es)/ADT(s) and give the most efficient algorithm <u>in terms of worst case time complexity</u> you can think of to determine the set of roads to be actually built such that <u>there is still a way to get between any pair of cities and the cost of building all the roads is **minimum**</u>.
[12 marks]

Quite obvious from the description that this is a MST problem. However the caveat is that there are only 3 possible weights on the graph: 1*C, 2*C, 3*C. In this case, we can use Prim's algo but we don't even need a PQ, only 3 normal queues, each for storing one of the 3 different edges.

1. each edge being processed is simply enqueued into the corresponding queue instead of a PQ.

2. when dequeuing, priority is queue for weight C edges, if empty then queue for weight 2C edges, if empty then queue for weight 3C edges. If all 3 queues are empty then end the while loop.

Here enqueueing and dequeuing takes O(1) time instead of O(logN) time. Thus total time complexity is O(M).

Another solution is to use Kruskal's, but again enqueue all edges into 3 queues as above, then simply go through all queues from C queue to 3C queue. Enqueueing all edges take O(M) time and dequeueing all the edges takes O(M) time. Thus time complexity is O(M).

Yet another solution is to transform the graph by dividing the edge weights by C, so you have only edge weight of 1,2 or 3. Now an MST of this transformed graph will be an MST of the original graph since cost of any ST in the transformed graph will simply be multiplied by C for the original graph. Thus now you can simply perform radix sort on the edge list in O(M) time and perform standard Kruskal's.

Marking Scheme:

Correct Graph modeling = 2 marks

Correct Solution:

O(M) solution as given or the equivalent = 10 marks

$O(N^3)$ Floyd Warshall = 4 marks

$O(N^2)$ Prim's variant or O(ElogV) standard Prim's/Kruskal's = 7 marks

Solution that performs radix sort on edges with the original edge weight = 5 marks. Since C is not bounded radix sort can do even worse than O(MlogM).

Wrong Solution:

1. There are quite a few solutions which first divide all edge weights by C then further transform the graph again by using extra vertices and edges so that all edges are now weight 1 (similar to solution for tutorial 11 3d), then run BFS/DFS on the transformed graph to get a ST. The rational is that in a graph with only weight 1 edges any ST is a MST. However, the problem is that a MST of the transformed graph in not necessarily a MST of the original graph. 3 marks

Deductions (capped at 10 marks of deduction):

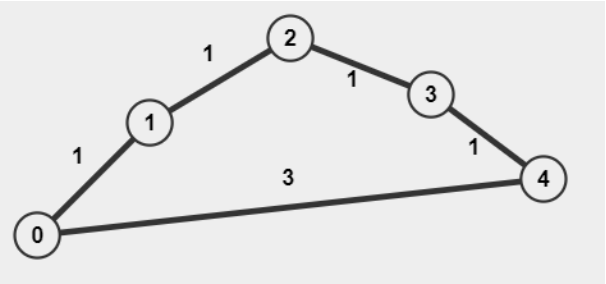
1. 2 marks for every major mistake

2. 1 mark for every minor mistake

Comments:

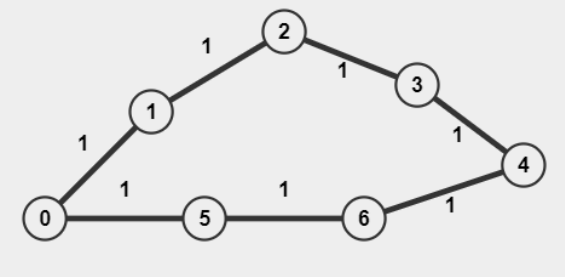
1. If you do not mention radix sort specifically when you solution allows for O(M) radix sort, it will be taken as O(MlogM) sort

2. Regarding the wrong solution mentioned above. Below is an example



If you have a graph as above (edge weight already divided by C), then the MST would simply be the removal of edge (0,4).

Now the transformed graph for the above would be as follows



with the additon of vertices 5 and 6 and edgs (0,5),(5,6) and (6,4) to replace the weight 3 edge (0,4). Now if BFS/DFS starts from vertex 0, then depending on how neighbors are picked, (3,4) might very well be the edge not picked to be in the BFS/DFS ST. Now this will indeed be a MST of the transformed graph. However this is not a MST of the original graph, since removal of (3,4) means in the original graph the cycle is broken by removing an edge of weight 1 and not the edge of weight 3.

17. In the future there is a popular game played among N+1 robots that are placed at integer coordinates (x,y) (x,y ≥0) around a large field. The distance between 2 robots at (x,y) and (x',y') can be calculated as $\sqrt{(x-x')^2+(y-y')^2}$. Each of these robots will hold a ball, called the owned ball at the beginning of the game.

    To play the game, there will be 1 selected robot and the other N robots will each need to pass the owned ball it is holding at the beginning of the game to the selected robot in turns (starting from robot 1 and going to robot N+1, excluding the picked robot itself). A robot can start to pass its owned ball only after the robot before it has finished passing its owned ball to the selected robot or the robot before it has forfeited its turn (more on this in the description below).

    Each robot can throw a ball it is holding to another robot who will catch it, so the catcher can have up to 2 balls, the caught ball and the catcher's owned ball if that has not already been passed to the selected robot. However, each robot can only throw to some other robot that is within a radius D of itself. To successfully get the owned ball to the selected robot, each robot needs to plan a path for the owned ball to be passed among the other robots until it reaches the selected robot in the shortest time possible (the longer the distance traveled by the ball the longer the time it takes).

    If there is no way for a robot to pass the owned ball to the selected robot, the robot will forfeit its turn. All robots that can pass their owned ball to the selected robot are called valid robots. You may assume there is 0 lag time between a valid robot having passed its owned ball to the selected robot and the next valid robot starting to pass its owned ball. In addition, the robots are always placed so that **no more than maximum(100, N)** valid robots can pass their owned ball to any selected robot.

    Given an array A containing the coordinates of the N+1 robots, use the most appropriate DS(es)/ADT(s) and give the most efficient algorithm in terms of worst case time complexity you can think of to determine the selected robot such that it satisfies the following 2 conditions (condition 1 is more important than condition 2)

    1. The most number of valid robots can pass their owned ball to the selected robot
    2. Sum of the distance taken to pass the owned balls of all valid robots to the selected robot is minimized.

    As an example if there are N+1=6 robots placed at coordinates (0,0), (1,1), (5,0), (5,1), (6,2) and (7,0) respectively and the distance every robot can throw is D = 2, then the robot at (5,0) or (5,1) should be the selected robot, with only (0,0) and (1,1) not being able to pass their owned ball to the selected robot, and the sum of the time taken to by the valid robots to pass their owned ball to the selected robot is minimized.
    [12 marks]

Model this as a graph G.

Vertices = coordinates of each robot

Edges = there is an undirected edge between every pair of vertices (i,j) and (i',j') where sqrt((i-i')^2+(j-j')^2) <= d

Number of vertices V = N

Number of edges E can range from O(N) to O(N^2)

Building the graph will take O(V^2) time (need to check each pair of vertices to see if they can be linked with an edge)

1. To satisfy the 1st condition, you need to determine the largest component in G. Run Counting Components algorithm and label all vertices according to their component number, keeping track of the size S of the largest component and the component number L. ← O(V+E)

2. To safisfy the 2nd condition, you need to find the robot V in the component L such that the sum of the SP cost from V to every other robot is minimum. To determine V

    a. Using a Hashmap H to remap the vertices in component L to vertices numbered from 0 to S-1. ← O(S)
    b. Initialize a D matrix of size S*S using only the vertices in L and run Floyd Warshall. ← O(S^3)
    c. Go through the matrix and sum up each row r to determine which row r has the minimum sum. Select H.get(r) as the robot. ← O(S^2)

Total time complexity = O(V^2)+O(V+E)+O(S)+O(S^3)+O(S^2) = O(V^2+V+E+S^3). Now since S can only be at most 100 based on the problem description, S^3 is therefore bounded and can be considered a constant. Thus total time taken is O(V^2+E) = O(V^2) = O(N^2).

Marking Scheme:

The maximum marks awarded are as follows (before further deductions due to mistakes):

12 marks: Giving the above algorithm, or some other algorithm which runs in worst case $O(N^2)$ time (inclusive of time taken to construct the graph).

10 marks: Giving an algorithm that runs slower than $O(N^2)$ time, but correctly determines the selected robot (ie. has the maximum number of valid robots, and ties broken by the minimum total distance). The most common way to do so is to run Floyd Warshall's on the entire graph at once.

8 marks: Giving an algorithm that selects a robot only based on number of valid robots. If you attempted to select a robot based on total distance as well, but did so incorrectly (eg. solving via MST), you may be awarded **9 marks** instead (barring any further mistakes).

6 marks: Giving an algorithm that selects a robot only based on total distance.

6 marks: Giving an algorithm that only considers direct throws to the selected robot, rather than paths consisting of possibly more than 1 edge.

4 marks: Attempting to solve a different question. The most common one is assuming the selected robot is already given, and finding the number of valid robots/total distance.

Some guiding questions you may wish to consider for the review session:

- Are the vertices and edges in your graph clearly stated?
- Does your algorithm run in longer than $O(N^2)$ time? (for cases listed as 8/6/4m)
- Is the constraint of throw distance (D) correctly handled?
- Is your program clearly returning a selected robot? (some answers return a list of selected robots if they are tied for both valid robots/distance; this is fine too)
    - Some answers return a 1D/2D distance array, which is not the expected result
- Are the conditions for the selected robot handled correctly? (largest number of valid robots, then smallest total distance)

18. In the kingdom of Wadanka with a population of N people, each person is given a unique integer id at birth. This id is useful in the following way.

The social ranking of each person in the kingdom is determined by their id. The higher the id the higher the social ranking. The person with social ranking = 1 also has the largest id and the person with social ranking = N also has the smallest id. (Note: social ranking = 1 is the highest social ranking)

Wadanka is made up of many tribes. The leader of the tribe is the person with the highest social ranking and therefore the largest id in the tribe. Whenever 2 tribes in Wadanka encounter each other, they "fight" by comparing the total sum of id of all persons in their tribe. The tribe with the larger total sum will be the victor. If both tribes have the same sum then the leaders will compare their id and the one with the larger id will be the winner (note that every person's id is unique thus there will not be a tie here). The following will then happen

1. Let X be the leader of the winning tribe and Y the leader of the losing tribe.
2. If Y is the person with the current highest social ranking in Wadanka then X will have the id of Y added to his/her id.(e.g if X has id = 10 and Y has id = 15 and is currently the person with the highest social ranking, then new id of X is 25)
3. If Y is not the person with the current highest social ranking, then id of X and Y will be exchanged if id of X is smaller than id of Y. Otherwise nothing is done.
4. The 2 tribes will be merged, and the new leader is the person with the largest id in the new tribe.

Given that there are N starting tribes in Wadanka with 1 person each (the leader him/herself) and the id of these N persons are given in an array A, use the most appropriate DS(es)/ADT(s) and algorithms you can think of to implement the following 3 operations in the required time complexity:

1. **FightAndMerge(a,b)** -> the tribe that person with id = **a** belongs to and the tribe that person with id = **b** belongs to will fight to determine the victor (according to the rules stated above) and the 2 tribes will be merged. If both persons belong to the same tribe then nothing happens. This should run in **worst case ≤ O(logN) time**.

2. **Leader(a)** -> return the id of the leader of the tribe that person with id = **a** belongs to. This should run in **worst case < O(logN) time**.

3. **SocialRanking(a)** -> return the social ranking of person with id = **a**. This should run in **worst case ≤ O(logN) time**.

You may assume that the input arguments given to the 3 operations are always valid, and you can have a pre-processing step that takes **no more than O(NlogN)** time to populate the DS(es)/ADT(s) you use. **[12 marks]**

Use a combination of UFDS, AVL tree and Hashmap.
Let T be an AVL where each node corresponds to a person in Wadanka and stores the id of the person.

Let F be a UFDS of N items where item i corresponds to person at index i in A, and each disjoint set represents a tribe in Wadanka.
Let H be a hashmap that maps a person current id to item number in UFDS of the person. Initialize with (A[i],i) for all i from 0 to N-1.
Let SNL be an array of N items. For representative item i of each disjoint set in F, SNL[i] contains a pair (s,lid) where s = sum of ids of all person in same set as i, and lid = id of the leader in the same set as i.

FightandMerge(a,b) <- O(logN) since operation with highest time complexity is AVL delete and insert

```
x = F.findset(H.get(a)), y = F.findset(H.get(b))
F.unionSet(x,y)
r = F.findset(x)
max = max(SNL[x].lid, SNL[y].lid) // ensure leader has the max id
SNL[r] = (SNL[x].s + SNL[y].s,  max) // new merged tribe
if (SNL[x].s == SNL[y].s) // same sum, so do nothing
  return
if (T.findMax() != max &&
   ((SNL[x].s > SNL[y].s && SNL[x].lid < SNL[y].lid)
   ||
   (SNL[y].s > SNL[x].s && SNL[y].lid < SNL[x].lid))) // swap ids
   temp = H.get(SNL[x].lid)
   H.set(SNL[x].lid,H.get(SNL[y].lid))
   H.set(SNL[y].lid,temp)
if (T.findMax()== max) // one of the leader has largest id
  flag = false
  if (SNL[x].s > SNL[y].s and max == SNL[y].lid)
    ol = SNL[x].lid
    flag = true
  else if (SNL[y].s > SNL[x].s and max == SNL[x].lid)
    ol = SNL[y].lid
    flag = true
  if (flag)
    T.delete(ol)
    nl = SNL[x].lid+SNL[y].lid
    T.insert(nl)
    SNL[r].s += max
    SNL[r].lid = nl
    id = H.get(ol)
    H.delete(ol)
    H.set(nl,id)
```

Leader(a) <- O(1)

```
return SNL[F.findset(H.get(a))].lid
```

SocialRanking(a) <- O(logN)

```
Return N+1-T.rank(H.get(a))
```

Marking scheme:

For solution that does its job correctly (maximum marks still capped at 12):

	FightAndMerge()	Leader()	SocialRanking()
O(N)	2 marks	1 mark	1 mark
O(log N)	(max 8 marks)	(max 3 marks)	(max 3 marks)

Deductions (except for the 1st deduction listed below, other deductions are applied only after getting >= 8 marks from the table above):

- -2 marks for assuming ids are from 1..N <- always applied regardless of marks
- -1 mark for each minor mistake (failing to handle a whole case is NOT minor)
  e.g. not swapping ids if necessary, not increasing ids, not updating a UFDS attribute in general

Let total marks for this be X.

Design marks: Award up to 4 marks for correct design that enables correct and efficient ops.

Let total marks for this be Y.

Final marks = max(min(X,12),Y)

Note:

- To qualify for some mark band in Leader() and SocialRanking(), must have reasonable attempt in FightAndMerge() to enable the correct operation with the same time complexity in Leader() and SocialRanking()
- In the comment box, if design is given 4 it means you get 4 marks. You should not be adding in the other marks in the comment box.

Comments:

- Almost all students implicitly made some assumption about the ids being in 1..N or 0..(N-1) by not mapping person ID to the person's index within the UFDS
- Marking scheme for 2. Leader() was relaxed to allow O(log N) solutions to get full marks for that function. Otherwise, those who drop union-by-rank (while still using path compression) and place winner-on-top or leader-on-top may not get "practically O(1) worst case" as UFDS should (HashMap is also needed besides TreeSet/Map)

- Many students focused on algorithm, but algorithm is limited by the design aka data structure(s) and what is stored in them; e.g. if you don't store leader (array) in each UFDS element, then you can't just simply record a swap between leaderA and leaderB after a union
- Either being modular, or realizing that the weird rules on id apply to "winner" and "loser" may help you simplify your code. First union (by rank), maintain necessary attributes (leader id, sum) on only the **new representative**, maintain other data structures, then decide if any special cases apply (only "leader ID increasing due to losing leader being having the largest ID" needs to be a special case, but "swapping leader ID due to losing leader having larger ID" may not even need to be done)

Common mistakes:

- Storing rank is a bad idea, whether in array form or in an AVL tree
- Attempting to aggressively change the leader/tribe of every person when 2 tribes merge. UFDS is meant to balance the time needed to union and find items by modifying/accessing as few nodes as possible
- Assuming that the representative node will be the leader. Since all nodes in the set have the representative node as the topmost ancestor, place the leader as an attribute there. This way, it is easy to change the id of the leader of that set/tribe as you like

19. You are a treasure hunter and have found yourself trapped in the pyramid of king Tat while trying to discover treasure within!

The pyramid is a maze that is split into N (N > 10) rooms that are labeled from 0 to N-1. There are M (N-1 ≤ M ≤ $\frac{N(N-1)}{2}$) doors which will allow you to pass from one room into another room. If a room is linked to another room, they can only be linked by at most 1 door.

At the start all the M doors are shut. As you approach a shut door in a room, it will automatically open, and you can go into the next room. You can revisit rooms you have visited before, but you can only go through at most K (0 < K ≤ 5N) rooms (including repeated rooms and the room you start in) before all doors in the pyramid will shut trapping you permanently! Luckily, you have a map of the entire pyramid and so know how the N rooms are linked by the M doors.

From the map, you have created an array R that contains M descriptions, each of which is a pair (x,y) meaning that room x is linked to room y by a door.

You start off at room A and you need to get to a room B which has the only door that opens to the outside, thus allowing you to escape, and you are sure there is a way to reach B from A before you become trapped. Once you reach room B, all the other doors in the pyramid will immediately shut permanently and the only door exiting the pyramid will open.

Escaping the pyramid is not the problem, rather you are faced with a dilemma. Starting at room A, you will have T_A years added to your lifespan (which starts at 80 years). If you enter room i next, you will have T_i years deducted from your lifespan. If you enter room j after room i, you will now have T_j years added to your lifespan again. Thus, this addition and deduction will keep alternating as you move from room to room (you will always start off with addition of lifespan then alternate). In fact, you can even reach -ve lifespan! But the effect will kick in only upon you exiting the pyramid.

Again you have created an array T of size N where T[i] will contain the amount of years to be added/deducted for room i (depending on when you visited it).

Given array R and T, model the problem as a graph (clearly state what are the vertices, what are the edges and how the vertices are linked by edges) and give the most efficient algorithm you can think of to start off at room A and reach room B with the most amount of lifespan possible. If you end up with -ve lifespan output "impossible to escape", otherwise output the lifespan.

As an example, if there are N = 6 rooms and the M = 6 doors linking the rooms are as follows (0,1), (1,2), (1,3), (3,4), (2,5) and (5,1) and the lifespan added/deducted for each room is as follows room 0 = 2, room 1 = 1, room 2 = 5, room 3 = 3 and room 4 = 4 and room 5 = 5. Given A = room 0, B = room 4 and K = 9, the best path to take is 0->1->2->1->5->1->3->4 with total lifespan = 80 + 2 + (-1) + 5 + (-1) + 5 + (-1) + 3 + (-4) = 88.
[10 marks]

Graph modeling:

Vertices

1. For each room r, in order to differentiate the situation where lifespan is added and the situation where lifespan is deducted when we step into the room, model it as 2 vertices one to represent the case when we step into r and T[r] lifespan is added and the other to represent the case when we step into r and T[r]
2. Now since we can go through at most K rooms, we need to further augment our vertices with a parameter to indicate how many rooms we have go through so far before reaching the current vertex.
3. The final modeling of vertices is as follows: For each room r, we will represent it using a set of vertices having 3 parameters v,x,y where v is the room number, x = {0,1} where 0 represents the case T[r] is added and 1 represent the case where T[r] is deducted, y = {1,2..K} to indicate the vertex count, i.e how many vertices we have gone through from source until reaching the current vertex. Thus each vertex represents the current state that we can be in as we move from room to room
4. Thus we will have 2*K number of vertices per room for a total of 2*K*N vertices in the graph.
5. Each vertex completely encapsulates the current state you are in, and edges allow you to transition from one state to another. So the graph is called a state-space graph.

Edges

1. For each door (x,y)
    a. Let there be a directed edge of weight -T[y] from vertex (x,0,c) to vertex (y,1,c+1) for all 1 <= c < K
    b. Let there be a directed edge of weight T[y] from vertex (x,1,c) to vertex (y,0,c+1) for all 1 <= c < K
2. There are at most 2*K edges for each door for a total of 2*K*M edges in the graph

This graph can be constructed in O(2*K*N+2*K*M) = O(KN + KM) = O(N^2 + NM) time, since K = O(N) in the worst case.

Note that there cannot be any cycles in this graph as we can only go from vertices with lower vertex count to vertices with higher vertex count. Thus this is a DAG.

The problem we're trying to solve is then the SP on a DAG where the source vertex is (A,0,1). Here we initialized d[(A,0,1)] = 80+T[A] and the distance estimate of the rest of the vertices to infinity.

Run 1-pass bellman on the graph with the exception that when going through the vertices in topological ordering, whenever a vertex is related to room B do not relax its outgoing edges since you can no longer move once you reach room B.

Return the minimum SP cost among vertices (B,0,1) to (B,0,K) and (B,1,1) to (B,1,K). ← O(KN+KM) time

If the minimum is –ve return "impossible to escape" else return that minimum.

Total time taken = O(KN+KM) time

Marking Scheme:

Correct graph modeling = 5 marks

1. marks distribution for graph modeling

- consider 2 vertices for each room, 1 representing deduction of lifespan when you step into the room and the other representing addition of lifespan. 1 mark
- correct assignment of directed edge & edge weights between the correct pair of vertices for each door linking room x and room y. 1 mark
- further splitting vertices into K vertices for each of the 2 vertices that represent a room so that each vertex has a extra attribute that is the counter for the number of rooms visited so far. 1 mark
- for every edge x->y in the original graph have an edge from x,i-> y,i+1 for i from 1 to K-1 in the transformed graph. 1 mark
- negate all the edge weight in the transformed graph since we want to find longest path / modify algo to perform stretching rather than relaxation. 1 mark

Correction solution based on correct graph modeling:

Floyd Warshall = 1 mark
Bellman Ford = 2 marks
Dijkstra = 3 marks
1-pass bellman ford = 5 marks

Correction solution will only be taken into account when graph modeling is mostly correct, at most making the 2 mistakes below

- forget that door links rooms bi-directionally
- forget to negate edges or the equivalent change to SSSP algorithm to get longest path

Other solutions:

1. Exponential time solution - modified DFS/BFS to search for all valid paths from room A to room B and find the maximum weighted one among them. 3 marks

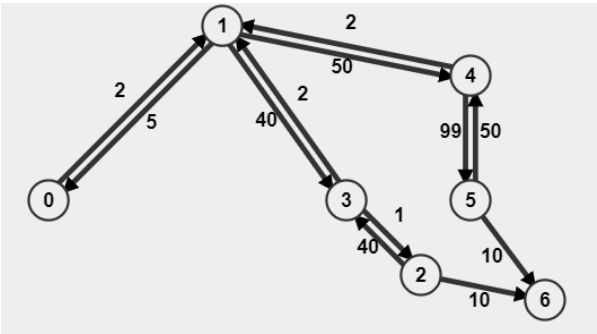
Deductions:

1. major mistakes. -2 marks
2. minor mistakes (e.g doors only link rooms in 1 direction). -1 mark

Comments:

1. Some answers use a global counter to keep track of the number of rooms visited when running Dijkstra and incrementing this counter whenever some vertex is dequeued from the PQ. This is not correct. For example if the counter is currently 4 and a vertex X with a path of 4 vertices is dequeued from the queue, the global counter will be incremented to be 5, next if a vertex Y with a path of 3 vertices is dequeued from the queue, the counter will still be incremented and it will be 6. If K = 6, then Dijkstra will terminate. However this is not correct since the for the current vertex the number of vertices has not exceeded 6.

2. Some answers will greedily pick the least weighted outgoing edge if the current turn is a deduction and greedily pick the largest weighted outgoing edge if the current turn is an addition. However this may not result in the best solution. An example is given below



The above graph, T[0] = 5, T[1] = 2, T[2] = 1, T[3] = 40, T[4] = 50, T[5] = 99, T[6] = 10. Assuming room A = 0, room B = 6 (that's why not outgoing edges from 6) and K = 5, then the best path will be 0->1->3->2->6 with a resultant lifespan of 80+5+(-2)+40+(-1)+10 = 132. However based on the greedy solution when you are at vertex 1, the next turn is a addition so you will pick the largest edge among all outgoing edges to process which is edge (1,4). Thus the path will be 0->1->4->5->6 with resultant lifespan 80+5+(-2)+50+(-99)+10 = 44 which is worse than the best path. You can observe that choosing the lesser weighted edge can lead to net increase later on due to a small deduction at (3,2), while choosing the larger weighted edge lead to a net decrease later on due to a huge deduction at edge (4,5).

3. Quite a number of students consider the number of rooms visited, the addition/deduction of lifespan for items in the PQ used in Dijkstra but did not model the graph accordingly. That is, the graph is still each room being one vertex. This can lead to the wrong solution. The following will show an example of this

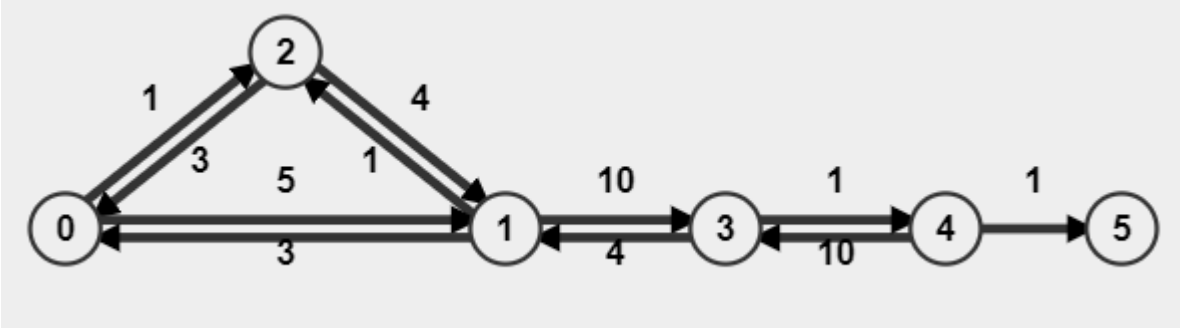
- Consider only addition and deduction being modeled as an extra boolean field t (assuming true being addition and false being deduction) for the items to be stored in the PQ, so that

for a vertex X dequeued, all its neigbors to be enqueued (if they can be relaxed) will have their t set to the opposite of X.

In the example below, T[0] = 3, T[1] = 4. T[2] = 1, T[3] = 10, T[4] = 1 and T[5] = 1. Assuming room A = 0 and room B = 5 (thus no outgoing edges) and K = 6. At the start of Dijkstra (83,0,true) will be inserted and then dequeued from the PQ. Next (82,2,false) and (80,1,false) will be enqueued. Next (82,2,false) will be dequeued since we want longest path. Now D[1] at this point is 80, but because the current turn is an addition D[2]+w(2,1) = 82+4 = 86 which is better than 80 so D[1] will be updated to 86 and the path is now 0->2->1. Subsequently you can observe that you can only move from 1 to 3 to 4 to 5 so the path from 0 to 5 given by Dijkstra will be 0->2->1->3->4->5 with total cost = 80+3+(-1)+4+(-10)+1+(-1) = 76.

However this is not the best path because even though 0->1 is not the best path to vertex 1 it will result in the best path from 0 to 5 later on, which is 0->1->3->4->5 with total cost = 80+3+(-5)+10+(-1)+1 = 88.

In fact 0->2->1 is the best path to 1 where you perform addition upon reaching 1 and 0->1 is the best to 1 where you perform a deduction upon reaching 1, and 0->1 is the best path to use to reach vertex 5 with the maximum amount of lifespan. However because only 1 vertex is used to represent a room (wrong graph modeling), the "lesser path" to 1 is never tracked.



- A similar argument can be given for keeping track of rooms visited by using an extra field for the items to be stored in PQ but not factoring this into the graph modeling.

4. Some students consider running the outer loop of bellman ford K number of times to limit the number of rooms visited to K. This is not correct. As an example, it is possible that there is a -ve cycle in the graph and the edges are ordered so that each iteration will relax all edges in the cycle. Thus after K iterations each vertex in the cycle will have been relaxed K times meaning you have visited each room corresponding to the vertices K time so total rooms visited = K*size of cycle = O(KN).