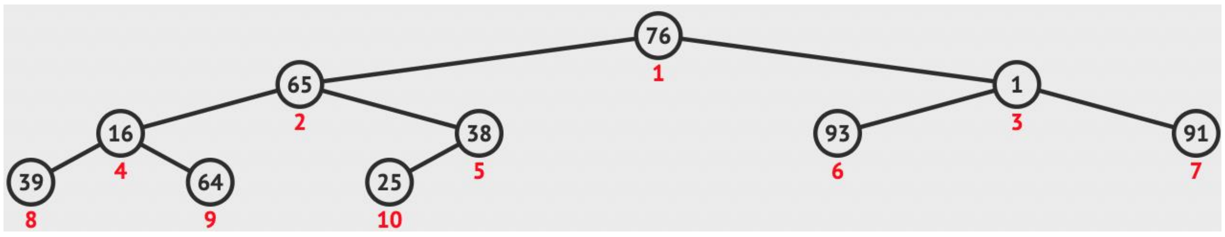
CS2040 2023/2024 Sem1 Final Assessment – Sample Solution+Grading Scheme

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

1. You are attempting to run an O(N) createHeap() to create a max 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: d

The following swaps will occur (the first number shown in the swaps is always that of the parent):

16 -> 64
1 -> 93
76 -> 93
76 -> 91

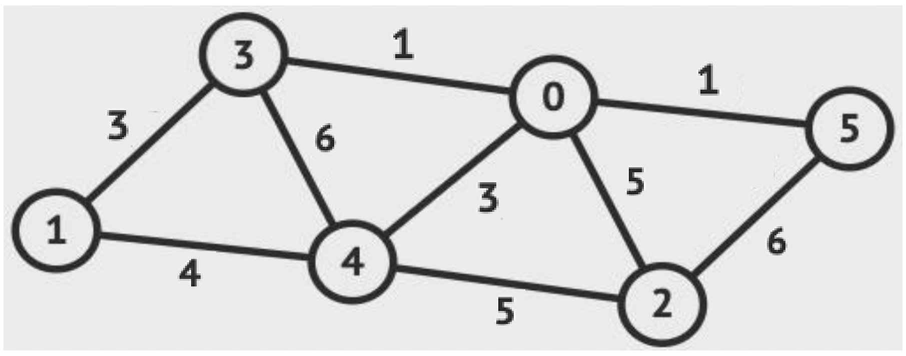
2. Given an array of N integers representing a min heap. What is the worst-case time complexity of the most efficient algorithm to convert it to a max heap?

a. O(1)
b. O(log N)
c. O(N)
d. O(N log N)

Ans: c

We can perform the usual heapify() operation on the given array in O(N) to convert it into a max heap. It does not matter if the given array is a min heap or not.

3. You are given the graph below.



   How many distinct minimum spanning trees are there in the graph?

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

Ans: b

The two distinct MSTs are: {(0, 5), (0, 3), (1, 3), (0, 4), (0, 2)} and {(0, 5), (0, 3), (1, 3), (0, 4), (2, 4)} with the total weight of 13.

4. You are given a complete undirected graph G on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5 and 6. What is the maximum weight that a minimum spanning tree of G can have?

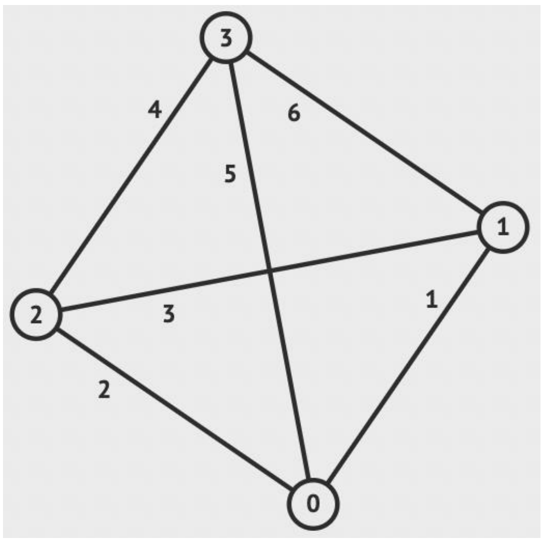
a. 6
b. 7
c. 8
d. 9

Ans: b

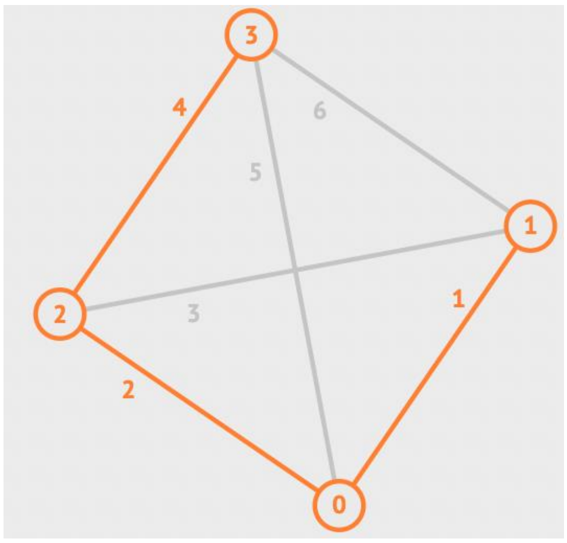
From Kruskal's algorithm, we know that the edge weights are considered in increasing order of 1, 2, 3, 4, 5, 6. Edge weights of 1 and 2 have to be included as 2 edges cannot form a cycle yet.

Since we want to maximize the weight of the MST, we assign edge weight 3 in G such that it forms a cycle. Hence, based on Kruskal's algorithm, we can ignore that edge and consider the next edge weight, which is 4. There is no way to form another cycle with edge weight 4. Therefore, it is included in the MST and the maximum weight that an MST of G can have is 7.

An example of G satisfying the condition above with its corresponding MST is shown below:



Original graph G



MST of G

Questions 5 and 6 refer to the following graph:

Consider an undirected, complete graph with 6 vertices. It is known that every edge has weight either 1 or 2. Given 2 vertices S and D:

5. How many simple paths are there from S to D?

a. 46,656
b. 720
c. 65
d. 24

Ans: c

For paths that use i vertices between S and D, there are $\binom{4}{i}$ ways to choose the i vertices and then $i!$ ways to permutate them. Since the graph is complete and every vertex is directly connected to every other vertex, we are guaranteed all paths are valid. The total number of simple paths is thus:

$$\binom{4}{4}\cdot 4! + \binom{4}{3}\cdot 3! + \binom{4}{2}\cdot 2! + \binom{4}{1}\cdot 1! + \binom{4}{0}\cdot 0! = 65$$

6. What is the maximum number of shortest paths from S to D possible?

a. 1
b. 4
c. 5
d. 6

Ans: c

Now since the graph is complete, S and D are directly connected. If this edge has weight 1, this path is the shortest and no other shortest path can exist since every other path must have weight > 1. So there is only 1 SP in this case.
On the other hand if this edge has weight 2, then every path that uses 2 edges can also be a shortest path if we assign each of them weight 1. There are $\binom{4}{1}$ paths which uses 2 edges. This gives us a total of 5 possible shortest paths, and is the maximum possible number of SPs.

Questions 7 and 8 refer to the following graph:

Consider a connected, weighted, undirected graph G with no negative edge weights.

7. Given a special vertex X in G, we want to answer Q queries of the form f(A,B) where f(A,B) denotes the shortest path from any vertex A to any vertex B **that contains vertex X** (there may be cycles in such a SP). What is the most **efficient** and **correct** algorithm to answer all Q queries among those listed? Vertex X remains the same for all the queries.

   Note: Some details of each algorithm may be missing, but based on what is given decide which is the most efficient and correct.

a. Run DFS once as a pre-processing step then answer each query in O(1) time for a total of O(V+E+Q) time
b. Run original Dijkstra once as a pre-processing step then answer each query in O(1) time for a total of O((V+E)log(V)+Q) time.
c. Run original Dijkstra once for each query for a total of O(Q(V+E)log(V)) time
d. Run Bellmanford once for each query for a total of O(QVE) time

Ans: b

We first run original Dijkstra once as a pre-processing step using X as the source to get the shortest distances from X to all vertices which takes O(V+E)log(V) time. This precomputation gives us all values of $\delta(X,v)$ where $\delta(X,v)$ denotes the shortest distance between X and v. Now, the shortest path from A to B that uses X is just:

$$f(A,B) = \delta(X,A) + \delta(X,B)$$

Thus, each query can now be answered in O(1) time and all Q queries can be answered in O(Q) time.

8. Instead of being given a special node X, you are now given a special edge Y, we want to answer Q queries of form g(A,B) where g(A,B) denotes the shortest path from any vertex A to any vertex B **that passes through edge Y** (there may be cycles in such a SP). What is the best possible time complexity to answer all Q queries including the time required for any pre-processing among those listed? Y will remain the same for all queries.

a. O(V+E+Q)
b. O(Elog(E)+Q)
c. O(Q(E)log(E))
d. O(QVE)

Ans: b

For special edge $Y = y_1 \to y_2$, we run original/modified Dijkstra twice as a pre-processing step. Once with y_1 as the source and another time with y_2 as the source to get their SSSP results D_1 and D_2 respectively. Time taken is O(ElogE). Note that since it is a connected graph E >= O(V) thus time complexity of original and modified Dijkstra is the same.

Then the shortest path from A to B that passes through edge Y is the minimum of $A \to \cdots \to y_1 \to y_2 \to \cdots \to B$ and $A \to \cdots \to y_2 \to y_1 \to \cdots \to B$. We thus have:

$$g(A, B) = \min(D_1[A] + D_2[B],\quad D_2[A] + D_1[B]) + \text{weight of } Y$$

Thus each query can be answer in O(1) time, and so total time taken is O(ElogE+Q).

9. You want to create an **AVL tree** by adding the elements 1, 2, 3, 4, 5, 6, 7, 8, 9 into the tree one at a time, in any order except that:
   - The first element added is 6
   - The second element added is 3
   - **NO rotation / rebalancing operation** must occur at any time

   How many different AVL trees of 9 elements can be produced at the end? Choose the interval that contains the correct answer

a. 0 of such trees
b. 1 to 2 of such trees
c. 3 to 5 of such trees
d. 6 or more of such trees

Ans: c

6 is the root, 3 is the left child of root. Using BST property, we can divide the elements into 3 subtrees.
The right subtree of 6 has to contain {7, 8, 9}. There is only 1 such orientation that satisfies the AVL property, 8 being the root of the subtree.
The left subtree of 3 has to contain {1, 2} while the right subtree of 3 has to contain {4, 5}.

There are 2 possible orientations for an AVL tree of {1, 2}, and 2 possible orientations for an AVL tree of {4, 5}.

Check if all 4 possible BSTs rooted at 6 satisfy the AVL property, i.e. are balanced. In any case, the node at 3 will have balance factor of 0 (perfectly balanced) and the node at 6 will have left subtree taller by 1. Therefore, all these 4 different AVL trees are possible.

10. There is a UFDS of **N** items, with **both** union-by-rank heuristic and path compression, each item being its own disjoint set initially. Ivan then performs **some** (zero, one or more) number of findSet, isSameSet and/or unionSet operations.

    You pick an item from the UFDS and examine the characteristics of that item.

    Which of the following options is true?

a. If a representative item is picked, its rank must be equal to its height
b. If an item picked is NOT a representative item and NOT a leaf, its rank must be equal to its height (the height of the subtree rooted at the picked item)
c. If a leaf is picked, its rank must be equal to its height
d. An item's rank can NEVER be equal to its height
e. None of the other options are true

Ans: e
Rank increases whenever the tree could possibly grow taller. It is difficult to find the actual height since the height may decrease due to path compression. Hence rank serves as an upper bound for the tree height. Rank NEVER decreases.
Counter-example for (a) – Create a UFDS of 1 set with the representative having rank and height 2 and then call findSet on the items at the bottom-most level
Counter-example for (b) – Create a UFDS with many nodes. Union 8 nodes one at a time to get a set with representative having height 3. Next, create the counter-example for (a). Finally unionSet the 2 sets' representatives together. The former representative of the second set would have differing rank and height
Counter-example for (c) – A leaf could have non-zero rank (was previously not a leaf), but after path compression on all of its descendants, it is now a leaf, i.e. has height 0
Counter-example for (d) – The item picked is itself a disjoint set (of 1 item), having rank 0 and height 0.

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

11. Claim: Given a general directed graph G with V vertices, running Bellman ford algorithm on a source vertex s of G for only k (k < V-1) iterations of the outermost for loop (k passes of relaxation) will only find the shortest paths from s to vertices where the shortest paths are up to k edges long. For vertices with shortest paths from s having more than k edges, no valid path can be found yet.

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

    a. True because the first pass will find the SPs to vertices that are 1 edge long, the 2nd pass will find the SPs to vertices that are 2 edges long ... so the kth pass will find the SPs to vertices which are k edges long.

    b. False because even though after k passes only vertices where SPs are up to k edges long will have their SP found, for vertices with SPs longer than k edges, if there are some other paths from the source to those vertices with less than k edges, a valid path (though not the shortest) can still be found after k passes.

    c. False because it depends on the order of edge relaxation, given a good ordering you may encounter a situation where you will need k or less iterations before finding the SPs to all vertices even though the SP with the most number of edges can be more than k.

    d. False because you may need more than k passes of relaxation to even find SPs that are k edges long.

    e. False because you can only converge on the SSSP solution using less than V-1 passes of edge relaxations if the graph is a DAG.

    Ans: c

    Even if the graph is not a DAG, if a good ordering is given to the edges to be relaxed (whether or not such a good ordering can be computed efficiently), you will need less than k pass.

    In general a good ordering of the edges is one where the edges in each of the SPs from the source vertex are relaxed the order they appear in the SP from source to ending vertex (this allows the SP to each vertex to be discovered in 1 pass of relaxation).

    An example is given in lecture notes 16 slide 8 (where the graph is not a DAG but given a good ordering of the edges you can converge on the solution in 1 pass).

12. It is known that DFS generally **cannot** be used to find shortest distance from vertex **x** to **y** in an unweighted graph.

    Now you are given a directed **unweighted acyclic** graph **G** with **V** vertices and **E** edges, along with valid vertex numbers **x** and **y**.

    There is a **modified** recursive DFS algorithm, in which the algorithm does **NOT store, mark, or check for visited vertices**. When exploring each neighbouring vertex, the distance estimate is 1 greater than the distance estimate from **x** to the current vertex. Recursively visit a neighbouring vertex if the distance estimate is improved over the previous estimate.

    Claim: This modified recursive DFS algorithm called on vertex **x** can successfully compute the shortest distance to vertex **y**

    Psuedo-code of the modified DFS matching the description above:

```
modifiedDFS(visited, distances, adjList, currVertex) {
   for neighbour in adjList.get(currVertex):
      if 1 + distances[currVertex] < distances[neighbour] then
            distances[neighbour] = 1 + distances[currVertex]
            modifiedDFS(visited, distances, adjList, neighbour)
}

SSSPOnDAG(adjList, x, y) {
   V = adjList.size()
   Create visited[V], init all to false
   Create distances[V], init all to +INFINITY except 0 for x

   modifiedDFS(visited, distances, adjList, x)
   return distances[y]
}
```

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

    a. False. Shortest path is not defined on a DAG

    b. False. Shortest path is not defined on an unweighted graph

    c. False. The graph may not be connected

    d. False. The algorithm will not terminate since there is no check for vertices being already visited

    e. True. There are no cycles. However, the time complexity may be much worse compared to using BFS

    f. True. Each vertex will be visited at most twice, and the distance estimate can only decrease. The big-O time complexity is the same as that of BFS since O(2**V** + 2**E**) = O(**V** + **E**)

g. True. Since there are no cycles, we will never hit a vertex twice, so there is only 1 possible distance estimate (other than +INFINITY) on a vertex when relaxing an edge. The algorithm will perform as good as BFS, or possibly better

h. True. Since the graph is a DAG, DFS, **even when unmodified**, will always take the shortest path to the destination.

Ans: e

On any (unweighted) directed graph, acyclic graph, OR unconnected graph, there is a distance(x, y) which is the number of edges needed to get from x to y, or infinity if not reachable. This eliminates (a)-(c).

When DFS hits a sink (a vertex of out-degree 0) it cannot recurse any further along that path, and needs to backtrack. Since there is no cycle, the algorithm will have eventually explored all possible paths and will terminate. This eliminates (d).

There is no reason why each vertex can be visited at most twice. Also, if it is possible to travel from a to b using 2 different paths and b to c using 2 different paths, there will be at least 4 different paths from a to c, not 2. The effect is multiplicative not additive. This eliminates (f).

DFS(x), modified or not, may still hit a vertex y twice as there may be a cross edge (think of why 3 states is needed for directed cycle detection in tutorial), so there may be multiple distance(x, y) estimates. This eliminates (g).

The unmodified DFS will not re-explore neighbours of already visited vertices, so on the first visit to vertex y, if the optimal distance(x, y) is not found then the correct distance will not be used in computing distance(x, y's neighbour). This eliminates (h).

Exploring all possible paths can take exponential time, see https://visualgo.net/en/sssp > Example Graphs > Corner Case > CP4 4.21 (Dijkstra's Killer) BUT ignore edge weights. Imagine vertices are labelled from right of the diagram to left of the diagram. This eliminates (g) again and supports (e).

Questions 13 and 14 refer to the following problem:

13. Alice and Bob are working at a company in the human resources department and they keep a record for all their employees. Every employee's performance is evaluated once a year and every employee receives a score. The 5 top-scoring employees are recognized and receive a small gift every year at the company's year-end party.

Alice and Bob use a max priority queue with all the employee scores so that they easily can determine the 5 top scorers.

Recently the company has realized that some of the lowest scores may be due to external factors, such as family situations, etc.

Now Alice and Bob's boss has asked them to report to him the 5 lowest scorers in the company. Alice is suggesting to Bob to make a copy of the max priority queue (a max heap), dequeue each score from the max heap and for each dequeued score enqueue it into a min priority queue (a min heap). Then the 5 lowest scorers can easily be extracted.

However, Bob thinks this is too slow.

He claims that all that is needed is to copy the entire 1-base compact array storing the max heap **in reverse** into another 1-based compact array. The new 1-based compact array will now become a min heap. This will only take O(N) time.

[2 Marks] Bob's claim is correct.

a. True
b. False

Ans: False

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

Ans:

A priority queue (a heap) is not completely sorted. It is only partially sorted top to bottom, but not fully, left to right. Therefore, a reversal of the array implementation is not guaranteed to be a valid min heap.

A simple counter-example is the following max heap 5,2,3 when you reverse it, you get 3,2,5 which is not a valid min heap since 3 is root but it is not the smallest.

Grading Scheme:

2 marks for correct answer stating heap property violation in Bob's approach

1 mark for stating the heap operation time complexity correctly

Notes:

- 1 mark is deducted for incorrect time complexity analysis.
- No mark is given if assume that the max heap is completely sorted in descending order.

Common mistakes:

- Claiming that a heap is completely sorted either in ascending or descending order. This is not correct since a heap is only partially sorted top to bottom, but not fully, left to right. Therefore, a reversal of the array implementation is not guaranteed to be a valid min heap.
- Assuming that heap is a form of BST where an in-order traversal would give an increasing sequence where the smallest element is the leftmost leaf and the largest element is the rightmost leaf. This is not true for the same reason as the bullet point above.
- Focusing on explaining how to solve the problem in O(N) time and what Bob should have done but did not explain whether Bob's claim of "copying the entire 1-base compact array storing the max heap in reverse into another 1-based compact array will now become a min heap" is correct or not.
- Claiming that "copying the entire 1-base compact array storing the max heap in reverse into another 1-based compact array" includes extractMax and other heap operations. However, what the question means is that Bob is simply reversing the array representation of the max heap and he claims that the resulting array is a min heap.
- Mistakenly interpret queue and dequeue operations in Alice's approach to pure queue operations rather the heap (priority queue) operations.

Questions 15 to 16 refer to the following problem

15. **[2 marks]** Given an AVL tree **T** that contains **N** numbers and an array **A** of length **N**, Pepega wants to fill **A** with the elements from **T** such that the numbers in **A** will be in ascending order. Pepega remembers that removing an element from an AVL tree takes O(log **N**) time, and repeated removal of the minimum from **T** to fill **A** will thus take O(**N** log **N**) time.

    Claim: Filling **A** with all elements in **T** such that **A** is now in ascending order takes worst case O(**N** log **N**) time.

    True/False

    Ans: False

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

Ans: No need to perform removal of keys from T to fill A, just traverse **T** in in-order fashion, filling **A** in sequence for each BST vertex visited. Pre-/In-/post-order traversal moves to each node at most 3 times, so only O(**N**) time is needed.

(The paragraphs below need not be given in your answer)
As in the Tutorial on BST, even if In-order traversal is performed iteratively with nested loops (repeatedly finding successor(node) which repeatedly walks through the tree), the previous paragraph is still true.

Due to BST property, a number X would only be visited (i.e. added to the array) when all numbers < X have already been visited, so A will contain a sorted sequence.

As for Pepega's argument that sorting using a balanced BST takes O(N log N) time at least, bear in mind that T already stores elements in sorted sequence, so more time is not needed to find the correct position of every element.

Grading Scheme:

2 marks	Correctly stating algorithm with worst case $O(N)$ run time. These include: • Inorder traversal where we directly copy into the back of Array A • FindMin() and continually calling Successor() which uses node reference instead of node value • Using lecture's ListSorted() as a subroutine and then copying into Array A
1 mark	Correctly stating worst case of $O(N)$ is achievable but not stating the algorithm Stating a correct algorithm but not mentioning the new worst case runtime Incorrectly analyzing a correct algorithm, e.g stating inorder traversal runs in $O(\log N)$
0 marks	Justifying Pepega's algorithm or time complexity Stating Incorrect algorithm

Common mistakes:

- Small number of students misunderstood that Array A also had some elements in arbitrary order. Array A is initially empty.
- Doing inorder traversal of an AVL tree already gives us the ascending sorted ordering. There is no need to perform any deletions, just simply copy them into the back of Array A as you process the nodes. There is also no need to perform any further sorting on Array A. This also applies if we are repeatedly calling Successor().

- Common misinterpretation: the question is ***not*** asking if Pepega's algorithm runs in worst case $O(NlogN)$, it is instead asking whether the best possible worst case is indeed $O(N\log N)$. This is not true because there are algorithms that run with worst case $O(N)$. Still, many students who misunderstood the question also incorrectly analyzed Pepega's algorithm so it is worth mentioning it:

  1 rotation does indeed take $O(1)$ time, and so does 1 rebalancing. However, a deletion might cause *multiple* nodes to require rebalancing, in the worst case $O(\log N)$ of them. Rebalancing aside, just finding the minimum node in the tree is already $O(\log N)$. This is because the minimum node in the AVL tree is the leftmost leaf which at least at a depth of $O(H-1) = O(\log N - 1)$ away from the root of the tree. Deleting the minimum thus node takes $O(\log N)$ time for a tree with N nodes. In general after removing i elements, there are $N - i$ remaining elements and removing the next element takes $O(\log(N-i))$ time. Summing over all elements we indeed get:

$$O(\log N \ + \log(N-1) + \ \log(N-2) + \ldots + \log 1) \ = \ O(\log N!) \ = \ O(N\log N)$$

- A small number of students mentioned an algorithm to put the root into the middle of array A, dividing A into a left and right subarray. Then, recursively calling the left child with the left subarray and right child with the right subarray. This is however incorrect as the root of any subtree is not guaranteed to be the middle element of its subarray. A valid AVL tree might have a root with a minimal AVL tree of height x as the left child and a complete binary tree of height $x+1$ as the right. Such a tree will not have the root as the middle element. This algorithm is probably inspired from the inverse problem of constructing an AVL tree given a sorted array in O(N) time.

Structured Questions: 5 questions

This section is worth 44 marks. Answer all questions.

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

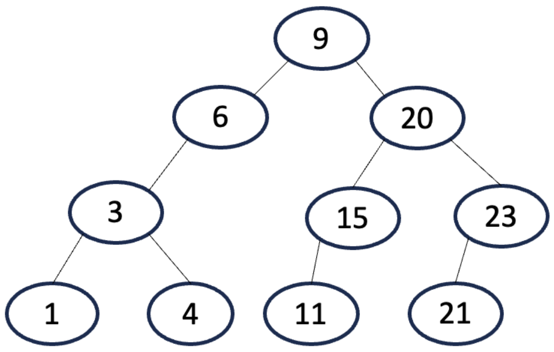
Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes.

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

17. **[6 marks]** Given the root of a **not necessarily balanced** binary search tree (not augmented with the size and height information, just the standard BST vertex) containing n integer values and *k* (1 <= *k* <= n) as input, find *k*-th largest element in the BST i.e., find the value of the (*n* – *k+1*)-th smallest element.

    Assumption: You are given the BST, but you are not given *n*, i.e., you don't know how many elements are in the tree.

    For example, in the following BST, if k = 3, then the output should be 20, and if k = 7, the output should be 6.



Select the **worst case time complexity** of the most efficient algorithm from the options listed.

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

Ans: c

Solution 1: You can use rank operation to find the k-th largest element, but since it is not a bBST/AVL and not augmented with size and height info, you must convert the BST into a bBST before calling rank, and that will take O(nlogn) time.

Solution 2: Since you don't know n, you can't find the (n-k+1)th smallest element. What you can do is first perform in-order traversal to get all the elements and add them to an arraylist. This takes O(n) time. Once this is done, you can get the (n-k+1)th smallest element from the arraylist in O(1) time. Total time taken is O(n).

Solution 3: You could do in-order traversal but recursively traverse the right subtree first before processing the current node and then recursively traverse the left subtree so that you will go through the elements in descending order, to ensure that the k-th node you process is the one you want. However again because the BST may be unbalanced and totally skewed to the right, you will still need to go make a recursive call on all nodes before you start processing the largest node, the 2nd largest node etc. So time taken is still O(n) in the worst case.

Solution 4:
1. call findMax() to get a reference to the node with the largest element ← O(n) time
2. call predecessor() k-1 times starting from the node with the largest element (each time starting from the current node instead of from the root) ← O(k) time.

Total time is still O(n).

18. **[6 marks]** You are given a weighted directed graph G modeling a chemical plant.
The vertices represent the state of a chemical reaction, and directed edges going from 1 vertex to another vertex represent the transition of the state of the chemical reaction to another state.
In order to transit from one state to another, the chemical reaction will give off some positive amount of energy. This amount is the weight of the edge representing the transition. You can assume this amount is an integer value > 0.
The maximum amount of energy given off by any transition in the graph is at most an integer value M and at least an integer value M'.
The total energy given off is the sum of the edge weights of the shortest path from a starting state S to an ending state S' in the graph.

The chemical plant cannot control the chemical reaction if the total energy given off is too much, so one way is to supply reagents into the chemical reaction at S and this will cause the energy given off by **all transitions** to be lowered.
If K amounts of reagents are introduced, it will cause the energy given off to be reduced by exactly K. You can assume K is an integer value and (1 <= K <= M').
For example if there is a path from state 1 to state 4 as follows: 1->2->3->4, if K amounts of reagents are used, then the weights of all the edges/transitions 1->2,2->3 and 3->4 respectively will be reduced by K.

Given the graph G, with V vertices and E edges, the values M and M', a positive integer value T, a starting state/vertex S and an ending state/vertex S', you want to answer the query of what is the least amount of reagent to use so that the **total**

energy given off is <= T. If the amount of reagent required is > M', the answer to the query should be "impossible".

Note that the chemical reactions cannot transition from one state x to another state y and then go through 1 or more transitions and come back to state x , as this means there is the ability to perpetually give off energy which breaks the laws of physics.

Select the worst case time complexity of the most efficient algorithm (using what you have learned in CS2040) to answer the above query.

a) O(ElogE)
b) O(VE)
c) O(M'ElogE)
d) O(M'VE)
e) O(M'(V+E))
f) O((V+E)logM')

Ans: f

Since the chemical reaction cannot give off energy perpetually, it means there are no positive cycles in graph, since the edge weights are only positive values, it means there are no cycles in the graph at all. The graph is a DAG.

In order to find what is the minimum amount of reagent required, you start with K = 1 and run one-pass bellman ford from S, and each time when relaxing an edge, first deduct the edge weight by K. At the end, check if D[S'] <= T. If it is still bigger, increment K by 1 and repeat until K == M'. If at some point D[S'] <= T, that will be the minimum amount require so output it. Otherwise if D[S'] > T when K == M', simply output impossible. This algorithm will take O(M'(V+E)) time.

A faster way is to do a binary search for the minimum required amount of re-agent.

```
1. Let Left = 1, Right = M' and K = (Left+Right)/2
2. Run one-pass bellman using K amount of re-agent
3. If meet requirement:
   Right = K // need to include K since that might actually be the min amount required
   If Left >= Right output K and terminate
   Else K = (Left+Right)/2, repeat step 2
4. If does not meet requirement:
   Left = K+1
   If Left > Right output "impossible" and terminate // even using M' is not enough
   Else K = (Left+Right)/2, repeat step 2
```

Since each time you run one-pass bellman ford, the range of K to search for is halved (starting at 1 to M'), you can only run it log(M') times before you arrive at the answer. Total time taken is O((V+E)logM').

19. **[11 Marks]** In a manufacturing project, there are **T** tasks and **R** pre-requisite rules, $0 \leq \mathbf{R} \leq \mathbf{T}^{0.5}$. Assume the tasks are numbered 0, 1, 2, … (**T**-1) for simplicity. You are given as input – **not yet read** into any data structure – the value of **T**, as well as **R** lines each containing 2 integers **x y** for $0 \leq \mathbf{x}, \mathbf{y} < \mathbf{T}$ and $\mathbf{x} \neq \mathbf{y}$ showing that task **x** can only be started after task **y** completes.

    Find if there is a case (just answer INVALID or VALID) where the rules are problematic, i.e. when the rules require some task **k** to be completed before task **k** may be started.

    *Remember, for this problem, O(**R**) is much better than O(**T** + **R**)*
    If you solve the problem in O(**R**) average time or better, you will get the full 11 marks
    If you solve the problem in O(**R** log **R**) time, you will get 10 mark
    If you solve the problem in O(**T** + **R**) time, you will get 9 marks

    [**Advice: If you are not confident of getting the top 2 time complexity bands, aim to get the 3rd band CORRECT]**

    If you are answering using a description, be very clear to explain how to model the problem, any data structure(s) and algorithm(s) you require. If you do make changes to any design or algorithm shown in lecture, you have to clearly explain in detail what the changes are.
    Give only one answer, do not give alternatives, or the lowest band marks will be awarded

Ans:

The rules are INVALID if there is a directed cycle, but VALID otherwise

O(**T**+**R**) as in tutorial
Create a directed unweighted graph, each task being a vertex and each rule an edge **y** -> **x**
Store graph in an adjacency list (AL of LLs of Integers)
* For each unvisited vertex 0 to (**T**-1), run DFS to find directed cycle, sharing the visited array across calls
If any one such cycle is found, output INVALID and terminate
If the loop completes, output VALID

O(**R** log **R**) worst using tree sets, O(**R**) average using hash sets
Model the problem similarly and use adjacency list BUT let the outer dimension be a set instead of ArrayList.
Run a similar algorithm to find directed cycle, BUT:

1. iterate over vertices in the set that is the outer dimension of the adjacency list instead of all **T** vertices
2. Use another set to mark visited vertices instead of an array

Conceptually, using the second approach ignores vertices of degree 0, such that the subgraph modelled has at most 2**R** vertices, which is an improvement since the graph is so sparse, **R** << **T**. Analyzing the algorithm, O(**R**) vertices and **R** edges are visited, so the time taken is the time taken to mutate / access the set O(**R**) times.

***** - Alternatively, run Kosaraju's algorithm to count the number of SCCs in the graph/subgraph. Again you can replace the arraylist of arraylist with set of arraylist, and use a set to mark visited vertices. If the number of SCCs is less than the number of vertices in the graph/subgraph, then there must be a directed cycle, hence INVALID. This will also run in O(R) time.

Grading Scheme:

General Notes:

- Minor typos are not penalized and the below scheme applies.
- It is important to recognize that the problem can be solved by constructing a directed, unweighted graph. The algorithm needs to check whether directed cycle(s) exist (INVALID) or not (VALID). Multiple algorithms/methods for directed cycle-detection exist.
- All the cycle-detection algorithms require an adjacency list (AL) as input. A standard AL results in complexity O(**T+R**) because there are T vertex IDs.
- In the given problem, the number of vertices that are involved in the rules (and thus are at the end of an edge) is much less than T (only 2R max.). So this is a very sparse graph. The complexity of the cycle-detection algorithms can be reduced to O(**R**) if only the vertices used in the rules are provided as inputs. This can be done, for example, by using a hash set/map, instead of a standard AL.

11m: The solution is correct and efficient with complexity O(**R**). The solution clearly mentions a method to reduce the algorithm's input from O(**T**) to O(**R**), for example, using a hash set/map.

10m: The solution is correct and efficient with complexity O(**R log R**). The solution clearly mentions a method to reduce the algorithm's complexity to O(**R log R**), for example using a tree set/map.

9m: The solution is correct and efficient with complexity O(**T+R**). The main idea is correct (checking whether there are any cycles, which means INVALID). Using a standard adjacency list results in O(**T+R**) complexity. There are multiple ways to check for directed cycles:

- Using Kosaraju's algorithm to count the number of SCCs in the graph/subgraph, which should be equal to T. If so, output VALID.
- Using Kahn's algorithm to compute a topological order. If it exists, output VALID.
- Using BFS/DFS (for directed graphs) to check for cycles. If no cycles, output VALID.

6m: The solution is somewhat correct in that it contains the main idea: the rules are INVALID if there is a directed cycle, but VALID otherwise (i.e., no cycle). However, the solution is lacking in some ways. For example, it has an efficiency of > O(**T+R**). This may be because the graph data structure is an adjacency matrix (O(**T²**)). Or it may be because the solution uses a UFDS data structure to detect cycles (the idea from Kruskal's), but note that in its general form, UFDS works on undirected graphs (e.g., MST), not on directed graphs. This is also the marks for solutions that try to use shortest path algorithms for cycle detection. This is not a shortest path problem.

4m: The solution has some right ideas, but also some major flaws. For example, the conflicting rules are not correctly detected. This may be because the algorithm does not check for directed cycles, rather it may only check for bi-directional edges (e.g., 1 $\rightarrow$ 2 **and** 2 $\rightarrow$ 1). Also, the problem/data structure was not recognized as requiring a directed, unweighted graph.

2m: Very incomplete, not understandable, solutions with many missing details. Partial solutions.

0m: No description or solution provided. Or the provided description cannot really be matched to the problem statement.

Common mistakes:

Quick Notes:

- A reminder: if you use a data structure or algorithm that has been covered in the course, you can just write down its name. No need to copy the code from your notes.
- Most students used an adjacency list as the data structure. This works fine and results in a O(**T+R**) solution because the list of vertices has a length of T. However, in the given problem most vertices have no edges (R << T). There are at most 2R vertices with edges. Thus, coming up with a data structure that excludes the vertices that have no edges can reduce the computational complexity.

Common Mistakes/Issues:

- Not understanding that the problem is about cycle detection in a directed, unweighted graph, i.e., not understanding how to model the rules as edges in a graph.
- Using an adjacency matrix of size T^2.
- Using a standard adjacency list for T vertices, but then claiming that the algorithm can run in O(**R**). If T vertices need to be checked for neighboring vertices it will take O(**T**) steps, even if most vertices have no neighbors.
- Using UFDS data structure to detect cycles (the idea from Kruskal's). However, in its general form using UFDS only works on undirected graphs (e.g., MST), not on a directed graph as in this problem.
- Checking only for very small cycles, i.e., 1 $\rightarrow$ 2 and 2 $\rightarrow$ 1. However, longer cycles must also be considered.
- Checking dependencies one-by-one, as the rules data is read/processed. Since the rules are listed in no particular order, checking the rules one-by-one without building a complete graph doesn't work. Also, we cannot assume that the vertex IDs appear in a specific order (e.g., lowest to highest) in the rules.

20. **[11 marks]** The valley of crystals is a strange place with M crystal formations of different colors growing on the ground. Each crystal formation is given a coordinates (r,c) where r and c are positive integer values.

    Energy bridges can be created between a pair of crystal formations to allow travel between the two, if the manhattan distance between the 2 is at most d.

    The manhattan distance between a pair of crystal formations at coordinates (r1,c1) and (r2,c2) is |r1-r2|+|c1-c2|.

    For any crystal formation in the valley, there is always another crystal formation within a manhattan distance of d.

    It is cheaper to create energy bridges between pairs of crystal formations of the same color due to resonance that allow minimal energy to be used. On the other hand it is more expensive to create energy bridges between pairs of crystal formations of different colors.

    You are given an array A of size M containing triples (r,c,k) describing the location (r,c) and color k (k is an integer >= 1) of each crystal formation in the valley, the value of d, and an integer value t.

    Formulate this as a graph problem and give the most efficient algorithm you can think of to answer the following query:

    <u>Determine if at most t bridges that connect pairs of crystal formations of different color need to be built so that a person can get from any crystal formation to any other crystal formation in the valley.</u>

    If this is possible return true else return false.

Ans:

Solution 1:
1. Number the crystal formation according to the index they are found in A.

2. Create a graph stored in an edgelist as follows:
For each pair of crystal formation x and y in A, if they are within manhanttan distance of d of each other
   a.) create an edge (x,y,0) if they are of the same color
   b.) create an edge (x,y,1) if they are of different colors
This takes O(M^2) time.
Let the number of entries in the edgelist be E.

3. Run Kruskal's algorithm and count the number of weight 1 edges used. If the count is > t return false, else return true. If count <= t & number of disjoint sets in the UFDS == 1 return true else return false. If there are more than 1 disjoint sets after Kruskal's is done, it means the modelled graph is not a connected graph to begin with so there is no way to go from any crystal formation to any other crystal formation.

Use Radix sort since the edge weights are only 0 or 1 so that sorting can be done in O(E) time. This causes Kruskal's algorithm to run in O(E*alpha(M))

Total time taken is O(M^2+E*alpha(M)). If E = O(M^2) then the algorithm is O(M^2*alpha(M)) else if E < O(M^2) then the algorithm is O(M^2).

Solution 2:
1. Create connected undirected unweighted graph with each crystal as a vertex. Store graph as an adj list by brute forcing every pair of crystals (x, y) in A, only add edge if dist(x, y) <= d. Run counting components algorithm. If there is more than 1 component return false since it means the graph is not connected, else go to step 2

2. Create another connected undirected unweighted graph with each crystal as a vertex. Store graph as an adj list by brute forcing every pair of crystals (x, y) in A, only add edge if dist(x, y) <= d and both x and y have same colour. Each CC can be connected with bridges within without affecting the quota t, but needs #CCs - 1 bridges to connect the entire graph

2. Label connected components by running DFS on each unvisited vertex from 0 to M-1, sharing the visited array and increasing the component number (initialized to 0, so the first component number is 1) for each unvisited starting vertex. Record the final component number, which is the #CCs

3. Return #CCs - 1 <= t

This solution is slightly faster since it requires exactly O(M^2) whether the graph is dense or not.

Solution 3:
1. Create connected undirected unweighted graph with each crystal as a vertex. Store graph as an adj list by brute forcing every pair of crystals (x, y) in A, only add edge if dist(x, y) <= d.

2. Run Prim's but instead of PQ use 2 standard queues, q1 to enqueue edges of weight 0 and q2 to enqueue edges of weight 1. To dequeue, always deque from q1 if it is not empty else dequeue from q2. This has the same result as using a PQ but will make enqueue and dequeue operations O(1) time. Thus total time taken for Prim's is O(V+E).

3. At the end, check taken array to see if all vertices taken. If not return false since it means the graph is not connected, else if cost(MST) <= t return true else return false

All 3 solutions will get full marks.

Note that even though it is stated that there is always a another crystal formation within a Manhattan distance of d from any crystal formation, it does not mean the graph formed is a connected graph. For example I can have 2 crystal formations with d distance of each other, then another 2 crystal formations with d distance of each other, thus fulfilling the requirement, but the 2 groups of crystal formations are not within d distance of each other so there is no way to get from 1 group to the other.

Grading Scheme:

Correct solution:

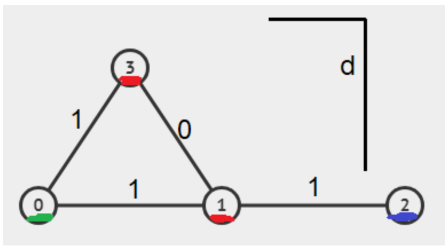
- 11 marks - O(M^2) or O(M^2 alpha(M))
- 9 marks - O(M^2 log M)

Additive (when the solution is not completely correct):

- 2 marks - Considers all candidate edges in O(M^2) time
- 2 marks - Differentiates between candidate edges with distance <= d and >d (as a filter, not just having edges sorted or distances sorted) successfully
- 2 marks - Differentiates (not just counts) between edges of same coloured crystals vs diff coloured crystals successfully (can enable a correct algorithm to work on the graph very easily)
- 2 marks - Prioritizes (first, not as tiebreaker) same-coloured reachable edges / groups them into a CC in O(M^2) or O(M^2 log M) time
- 3 marks - Other aspects of the algo and time complexity
    - including the need to check for case where it is impossible even with infinite bridges (if this is missing, max 9 marks, typically -1 mark)

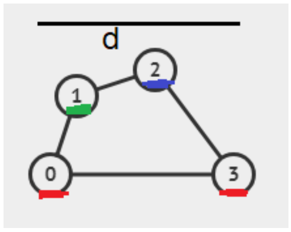
Common mistakes:

- Confusing vertex and edge attributes - one colour is a vertex (not edge) attribute, distance is an edge (not vertex) attribute (however, distance can be stored with the vertex if used in SP algorithms to represent accumulated distance from source, or in Prim's algorithm to represent an edge)
- Claiming that a coordinate grid where vertices can be placed is an adjacency matrix - an adjacency matrix of a grid of coordinates would be mat[r*c][r*c], not grid[r][c]
- Assuming that "there is always another crystal formation within a manhattan distance of d" means the graph of crystals that are reachable from each other is connected - it may not be
    e.g. M=4, v0-v1 and v2-v3 but {v0, v1} very far from {v2, v3}
- Answering query by using SSSP on graph of crystals and reachable distances between them, firstly there is no source here, also there is no reason to accumulate distances from some source
- Answering query by using SSSP of correctly modelled graph, which is to examine whether <= t edges to crystals of different colour are taken on some **path** instead of <= t such edges in entire graph
  Even if you use the algorithm to attempt to generate an MST, such an SP Spanning Tree may or may not give you an MST, so it is a wrong algo

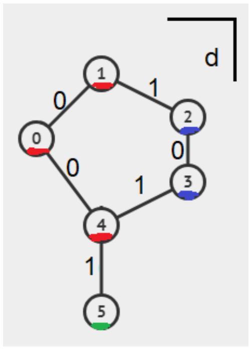


This example graph may also help you to see the importance of confirming that an edge is in the MST only when you confirm/process a vertex (dequeuing in Prim's), and NOT when you relax an edge

- Treating query as a longest-shortest-path (diameter) problem on correctly modelled graph - there may be other edges of different colour that need to be taken (to build expensive bridges) in the graph aside from the diameter (i.e. on a path from some vertex a to some vertex b having less edges than that of the diameter)
- Treating query as an MST problem but ordering edges by distance between crystals - if crystals are further apart but (still within distance d and of the same colour) we should prioritize it (distance doesn't matter at all after filtering edges by it)



- Sorting (comparison-based) edges with weights only 0 and 1 - can just split them into 2 buckets, or make 2 passes through the edge list
- Attempting to count different-coloured edges that must be taken by using DFS which prioritizes same-coloured edges in each call - DFS tries to go as deep as possible down a path, so it may hit a dead end and not have explored the entire connected component of same-coloured reachable vertices before taking a bridge to a different-coloured vertex



- Grouping all vertices of the same colour together - cannot neglect distance as some crystals of the same colour may be very far apart, needing bridges to other colours to connect them together
- Not explaining what vertices and/or edges are, characteristics of the graph (weighted? directed? if applicable, acyclic? connected?)
- Not explaining what data structures are used (besides those within standard algorithms) e.g. adj list / edge list to store graph
- "x at most y" means x <= y, not x < y
- CC is a term typically used for undirected graphs (or at least ignores edge direction), while SCC is a term used for directed graphs as it takes edge direction into account

21. **[10 marks]** We are back to playing dominos from THA4B. You are directly given the layout of N domino tiles (numbered from 0 to N-1) stored in an adjacency list AL.
To re-cap, if knocking over tile i will also cause tile j to be knocked over, i->j will be an edge in AL. Thus there are N number of vertices in AL and M edges where each edge models the above relationship.

This time you will be given a query **max_num_knocked_over(z)** where given a tile z return the maximum number of tiles that can be knocked over (excluding z itself) if z was knocked down by hand.

There are Q such queries.

Each query should be answered in worst case O(1) time. <- NOT EASY!
If your query runs in worst case O(V+E) time. You will get 30% of the marks.
If your query runs in > worst case O(V+E) time, you will get no marks.

Your solution may involve a pre-processing step that should run in time at most worst case O(V+E).
If your pre-processing step takes more than O(V+E) your marks that you get for the query operation will be halved.
If needed, you may assume you have a DFS algorithm **DFSLabel(B,visited,v,c)** that will take in an integer array B of size v, a visited array (visited[i] = 0 means vertex i is not visited, visited[i] = 1 means vertex is visited), a source vertex v and an integer value c. It will perform DFS from v as source vertex and also label v and all unvisited vertices v' reachable from v with c (by setting B[v] = c and B[v'] = c).

Ans:
The expected most efficient O(V+E) preprocessing + O(1) query solution is not correct and is currently no achievable, but for the sake of learning it will be posted here.

Preprocessing step:
Step 1: Run Kusaraju's alrogithm to label vertices with SCC number of the SCC they belong to as the SCC are being discovered. <- O(V+E) time

The SCC number starts from 1 and goes to M (where M is the max number of SCCs in the graph).
Note that the sequencing ordering of the SCC numbers from 1 to M is a topological ordering of the SCCs if we view the SCC as a super vertex.
For each starting vertex of a new SCC being discovered by Kusaraju's add it to the back of an arraylist A'.

The modification to the 3rd step of Kosaraju's (counting SCC on the transposed graph G') to perform the above is as follows:

```
Create visited array, array B of size N, create arraylist A’
SCC = 0
for v = 0 to N-1
  visited[v] = 0
for all v in K from last to first vertex
  if visited[v] == 0
     A'.add(v) // add v to the back of A'
     DFSlabel(B, visited,  v, SCC)
     SCC = SCC + 1
```

Step 2: Re-initialize the visited array. ← O(V)

Step 3: Go through A' from index M-1 to index 0 and perform DFS on each vertex v in A' as the source if v has not been visited (you are DFSing on the graph G). Keep track of the number of vertices reachable from v using DFS. Can re-use A’ for this purpose. ← O(V+E)

If a vertex u with a different SCC labelling as v is a neighbor of v, the SCC belonging to u must have already been visited (since we are going in reverse topological ordering of the SCC), so don't visit it but instead add the number of vertices reachable from u to the number of vertices reachable from v (so you only visit each vertex once).

The algo is as follows:

```
for i from M-1 to 0
   A'[i] = DFS(A'[i])
  // DFS will return the number of vertices reachable from vertex A'[i] and each vertex in
  // the same SCC as A'[i].

// Modification to DFS is as follows
int DFS(v)
  visited[v] <- 1
  int count = 1
  Let TN be a integer array of size SCC initialized to 0
  // TN ensures the number of vertices from a neighbor SCC is only counted once, since
  // there may be many outgoing edges from vertices in one SCC to another SCC.

  for all u adjacent to v
    if visited[u] = 0
      count += DFSrec(u)
    else if visited[u] == 1 and TN[B[u]] == 0
      count += A'[B[u]]
      TN[B[u]] = 1
      // u must belong to an SCC that has been explored (a larger SCC number) and the max
      // number of vertices reachable from those vertices already counted and stored in A'
     // Imagine this is a neighbor SCC of the SCC that v belongs to so add it's count to the
     // count for the SCC that v belongs to
   return count
  Since each vertex is visited only once in this step the total time taken is still O(V+E).
```

Query:

```
int max_num_knocked_over(z)
  return A'[B[z]]
```

Answer in O(1) time per query.

The problem is that in the pre-processing step, the number of vertices knocked over for any starting vertex z can be overcounted. A simple example is given below:

Assuming there are 3 SCCs, labelled 1,2,3 and they are connected as follows:

1->2
2->3
1->3

now every vertex in each SCC can knock every other vertex in the same SCC, also every vertex in SCC 1 can knock over every vertex in SCC 2 and SCC 3, while every vertex in SCC 2 can knock over every vertex in SCC 3. Assuming the number of vertex in each SCC is as follows: SCC 1 = 10, SCC2 = 20, SCC 3 = 30.

Now we will process the SCCs in reverse topological ordering:

we start from SCC 3 and calculate A'[3] = 30.
we move on to SCC 2 and since 2->3, A'[2] = 20 + A'[3] = 20 + 30 = 50
we move on to SCC 1 and since 1->2 and 1->3, A'[1] = 10+A'[2]+A'[3] = 10 + 50 + 30 = 90

You can see that number of vertices in SCC 3 is counted twice for SCC 1 due to SCC 1 have edges to both SCC 2 and SCC 3.

Currently the state of the art algorithm can only achieve ~O(V^2.3) preprocessing + O(1) query, thus it would be unfair to ask students to come up with a better algorithm on the spot.

Apologies for the oversight in not checking the solution properly.

Grading Scheme:

Due to the expected solution being unachievable, the maximum marks for this question has been reduced to 5 marks (so total marks is 95 not 100).

Correct solutions:

O(V(V+E)) preprocessing + O(1) query: 5 marks

Simply run DFS using each vertex v as source and count number of vertices visited (the reachable vertices which is the max number knocked over by v), and save this in an array. Then for query simply return the value in the array for index z.

no preprocessing, O(V+E) per query: 3 marks
Do the above for each call of max_num_knocked_down(z).

deduction for correct solutions:
-1 mark for minor mistakes.
-2 marks for major mistakes.

wrong solutions that try to solve the correct problem:
wrong O(V+E) preprocessing + O(1) query: 2 marks
wrong O((V+E)*alpha(V)) preprocessing + O(1) query: 2 marks

As long as there is a complete and detailed enough solution with the above 2 time complexities you will be given 2 marks. This is to make up for those who spent time trying to figure the most efficient solution expected.

wrong O((V+E)*alpha(V)) preprocessing + O(V) query: 1 mark
other worse time complexities for preprocessing + >= O(1) query : 1 mark

wrong solutions that solve a different problem (misunderstand the question): 0 marks

Incomplete solutions: 0 marks

Vague/unclear solutions: If the solution is vague (not enough details given), and the time complexity cannot be determined but there is some valid ideas presented then 1 mark, else 0 mark.