CSCI 570 – ANALYSIS OF ALGORITHMS HOMEWORK – 2

1. In the SGM building at USC Viterbi, there is a need to schedule a series of n classes on a day with varying start and end times. Each class is represented by an interval $[start\_time, end\_time]$, where $start\_time$ is the time when the class begins and end_time is when it concludes. Each class requires the exclusive use of a lecture hall.

   (a) To optimize resource allocation, devise an algorithm using binary heap(s) to determine the minimum number of lecture halls needed to accommodate all the classes without any class overlapping in scheduling. (7 points)

   (b) Analyze and state its worst-case time complexity in terms of n. (3 points)

ANS:

a) ALGORITHM USING BINARY HEAPS:

STEP 1: Sort all the classes in the order of the increasing start time (earliest class first) – For n classes i.e., run-time complexity will be $n \log n$.

STEP 2: Initialize a count variable Number_of_Halls_Required = 0 to keep a track of number of halls we need as per the given start times and end times of the classes.

STEP 3: Create an empty min-heap to store end times of the given classes.

STEP 4: Now we need to traverse through our sorted classes (sorted in increasing order of start times – Step 1) to allocate the halls as required.

STEP 5: Check in order for each class if the min-heap is empty or the root of the min-heap (earliest ending class) has an end time less than or equal to the start time of the current class.

This is to analyze if we can utilize that lecture room [with end time in root] for the current class and optimize the allocation of halls.

STEP 6: If this condition is met (i.e., Current Class Start Time <= Earliest Ending Class End Time [root]), remove the root of the min-heap (as we will be utilizing that lecture hall for our current class) and make no changes to our Number_of_Halls_Required count variable.

If the condition is not met (i.e., Current Class Start Time > Earliest Ending Class End Time [root]). Allocate a new lecture hall for the current class and increment (+) the Number_of_Halls_Required count variable by 1.

Add the end time of the current class to the min-heap irrespective of whether the condition is met or not.

STEP 7: After traversal is completed through all the sorted classes, return Number_of_Halls_Required as our final answer.

The size of our min-heap will give us the number of lecture halls required to accommodate all the classes without any overlap.

b) <u>WORST-CASE TIME COMPLEXITY IN TERMS OF N:</u>

The time complexity of the sorting step is $O(n \log n)$, where ' n ' is the number of classes – Step 1.

Initialization takes $O(1)$ and iteration through the sorted classes takes $O(n)$ time.

In worst-case, if all the n classes are overlapping. Then building a min-heap with n insertions at most take $O(n \log n)$.

Therefore, the algorithm has a worst-case time complexity of $\approx O(n \log n)$ in terms of 'n'.

2. The Thomas Lord Department of Computer Science at USC Viterbi is working on a project to compile research papers from various departments and institutes across USC. Each department maintains a sorted list of its own research papers by publication date, and the USC researchers need to combine all these lists to create a comprehensive catalog sorted by publication date. With limited computing resources on hand, they are facing a tight deadline. To address this challenge, they are seeking the fastest algorithm to merge these sorted lists efficiently, taking into account the total number of research papers (m) and the number of departments (n).

   (a) Devise an algorithm using concepts of binary heap(s). (7 points)

   (b) Analyze and state its worst-case time complexity in terms of m and n. (3 points)

ANS:

Given that all the research papers in n departments are in sorted order of their publication dates (earliest first) respectively.

So, our input will be n sorted arrays (one sorted array of research papers for each department) and the Total number of research papers = m

As the items are in sorted order respectively in each category, we can determine that by comparing an item x to the first item in the array → if x < first element of the array. Then it indicates x < all the individual elements in the array. This implementation is similar to that of merge sort (2-pointer concept) and efficient as we are using binary heaps.

a) ALGORITHM USING BINARY HEAPS:

STEP 1: Create a min-heap which will serve as a data structure used for comparisons of publication dates.

STEP 2: Initialize our Comprehensive_Catalog array to store the final merged list of all the m research papers.

STEP 3: Initialize n pointers, one per department to keep a track of current research paper index in its sorted list.

STEP 4: For each department, insert the first research paper (smallest by publication date) into the min-heap along with an identifier of the department.

It implies inserting tuples or pairs where the first element is the research paper and the second element is the identifier (or label) of the department. Each tuple should be structured this way so that you can keep track of which department each research paper belongs to during the merging process.

Example: If research paper dates are in DD-MM-YYYY format, then min-heap = [(20-03-2016, 'Research Paper 1 Title' 'Computer Science'), (03-11-2017, 'Research Paper 2 Title' 'Electrical')] etc.

STEP 4: While min-heap is not empty, Extract the root i.e., research paper with the earliest publication date from the min-heap and add it to the Comprehensive_Catalog (resultant array).

STEP 5: Determine the department of the research paper added to the Comprehensive_Catalog using the label.

STEP 6: If the respective pointer of that department is not pointing to the last element, Move the pointer of that department to the next research paper in its list and add the research paper (publication date & department label) to the min-heap.

If the pointer has reached to the last element already, it means that all the research papers in that department are already inserted into the min-heap.

STEP 7: Continue this process until we have all the m research papers in the Comprehensive_Catalog (resultant array).

b) <u>WORST-CASE TIME COMPLEXITY IN TERMS OF M and N:</u>

Inserting the first research paper from each department into the min-heap requires $O(n)$ to fetch and $O(n \log n)$ to re-order, where 'n' is the number of departments

The main loop, where our dominant step is iteratively extracting the research paper with the earliest publication date for adding into resultant array takes $O(m)$ and inserting the next papers from the respective departments takes $O(m-n)$ to fetch and $O((m-n)\ log\ n)$ for re-structuring the heap i.e., as every time, we extract an element from min-heap, it restructures it takes O $(log\ n)$ and we do it $m-n$ times i.e., O $((m-n)\ log\ n)$.

Therefore, the overall worst-case time complexity of the algorithm is $\approx \boldsymbol{O(m\ log\ n)}$ in terms of 'n' (number of departments) and 'm' (total number of research papers).

3. In an interstellar odyssey, a spaceship embarks on a journey from a celestial origin to a distant target star, equipped with an initial fuel capacity of `'currentFuel'` units. Along the cosmic highway, there are space refueling stations represented as an array of `'spaceStations'`, each defined as `[distanceToStationFromOrigin, fuelCapacity]`. There are `'n'` space stations between the celestial origin and the target star.

The objective is to determine the minimum number of refueling stops required for the spaceship to reach the target star, which is located `'targetDistance'` light-years away. The spaceship consumes one unit of fuel per light-year traveled. Upon encountering a space station, it can refuel completely by transferring all available `'fuelCapacity'` units from the station. The challenge is to calculate the `'refuelStops'` needed for a successful voyage to the target star or return -1 if reaching the destination remains unattainable with the available fuel resources.

(a) Devise an algorithm using concepts of binary heap(s). (7 points)

(b) Analyze and state its worst-case time complexity in terms of n. (3 points)

ANS:

a) ALGORITHM USING BINARY HEAPS:

STEP 1: Initialize **currentFuel** = **initial fuel capacity** and **refuelStops** = 0.

currentFuel is a variable that tracks the number of light-years that the spaceship can travel with the remaining fuel given the statement that 1 unit of fuel is required to travel 1 light-year.

target is the total number of light-years spaceship has to travel from the origin

If the **currentFuel** is ≥ **targetDistance**, return refuelStops. If not, proceed to next step.

STEP 2: Sort all the space stations in the order to increasing distance from the origin (earliest station first) – For n stations i.e., run-time complexity will be $n\ log\ n$. If the space stations S_1 to S_n are already in the order of the increasing distance from origin, we can skip this step.

STEP 3: Create an empty max-heap to store the max fuel capacity of all the space stations that can be reached using the **currentFuel**.

STEP 4: Iterate through the space stations in the order they are encountered along the cosmic highway assuming they are in a straight line (as mentioned as highway).

STEP 5: Check what are all the possible space stations we can visit with the currentFuel given that 1 unit of fuel is required to travel 1 light-year i.e., we can visit all the space stations that have **distanceToStationFromOrigin** ≤ **currentFuel**.

Insert all the **fuelCapacity** values of each of these reachable space stations (**distanceToStationFromOrigin** ≤ **currentFuel)** into our max-heap if they are not added already.

Now, extract the root (max fuelCapacity refill) and add it to our **currentFuel** and increment (+) **refuelStops** by 1.

STEP 6: If the **currentFuel** is ≥ **targetDistance**, return **refuelStops**. If not, repeat STEP 5 until **currentFuel** is ≥ **targetDistance** or our max-heap becoming empty.

STEP 7: If max-heap becomes empty, return -1 which indicates that the target star is unreachable with the available fuel resources.

b) <u>WORST-CASE TIME COMPLEXITY IN TERMS OF N:</u>

Initialization of variables take $O(1)$ (Step 1).

S_1 to S_n are not in the order of the increasing distance from origin, then the complexity for sorting n items in order takes $O(n\ log\ n)$ (Step 2).

Building max-heap using insertion takes $O(n \log n)$ in worst-case as maximum number of space stations that are in our reach is n.

Maximum number of re-fuels stops i.e., popping elements from max-heap is n. Hence, worst-case time complexity for this step is $O(n \log n)$

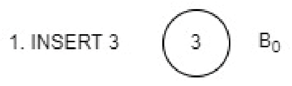
Therefore, the algorithm has a worst-case time complexity of $\approx \boldsymbol{O(n\ log\ n)}$ in terms of 'n' irrespective of whether Step 2 is required or not required.

4. You are tasked with performing operations on binomial min-heaps using a sequence of numbers. Follow the steps below:
   (a) Create a binomial min-heap $H1$ by inserting the following numbers from left to right: 3, 1, 13, 9, 11, 5, 7, 15. (2 points)
   (b) Perform one `deleteMin()` operation on $H1$ to obtain $H2$. (2 points)
   (c) Create another binomial min-heap $H3$ by inserting the following numbers from left to right: 8, 12, 4, 2. (2 points)
   (d) Merge $H2$ and $H3$ to form a new binomial heap, $H4$. (2 points)
   (e) Perform two `deleteMin()` operations on $H4$ to obtain $H5$. (2 points)

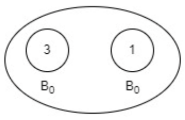
   Note: It is optional to show the intermediate steps in your submission. Only the five final binomial heaps ($H1$, $H2$, $H3$, $H4$, and $H5$) will be considered for grading. So, please ensure that you clearly illustrate your final binomial heaps ($H1$, $H2$, $H3$, $H4$, and $H5$). You can use online tools like draw.io for drawing these heaps.

<u>ANS:</u>

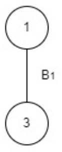
a) Create a binomial min-heap H1 by inserting the following numbers from left to right: 3, 1, 13, 9, 11, 5, 7, 15



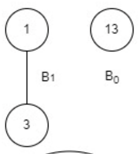
2. INSERT 1



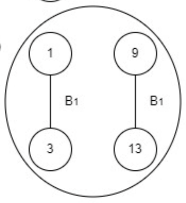
3. MERGE



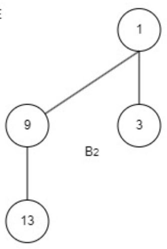
4. INSERT 13



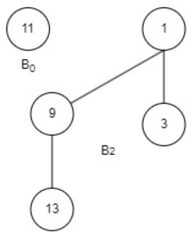
5. INSERT 9



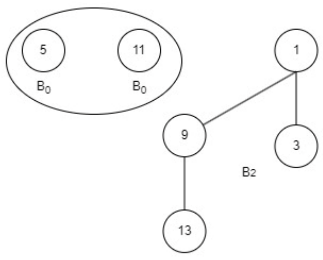
6. MERGE



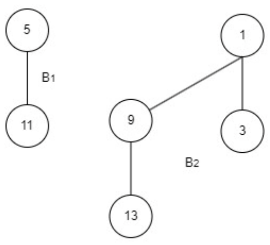
7. INSERT 11



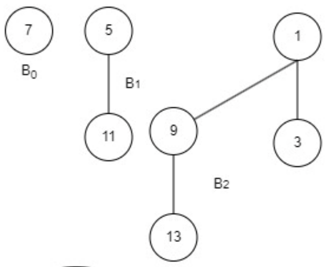
8. INSERT 5



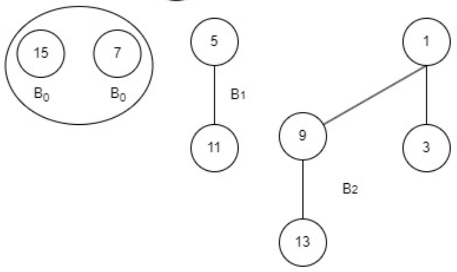
9. MERGE



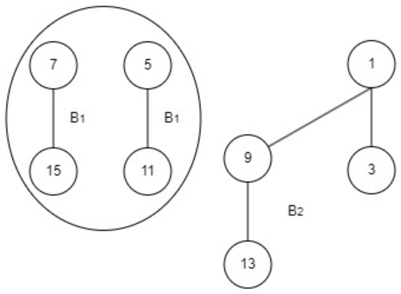
10. INSERT 7



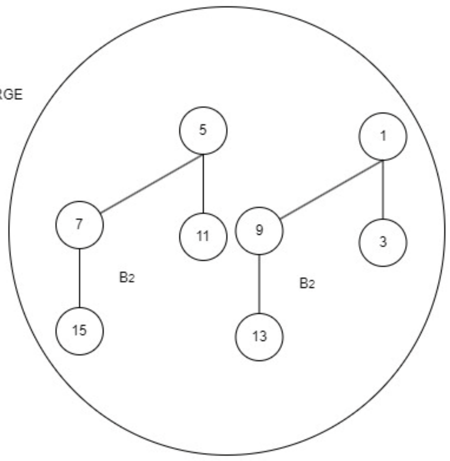
11. INSERT 15



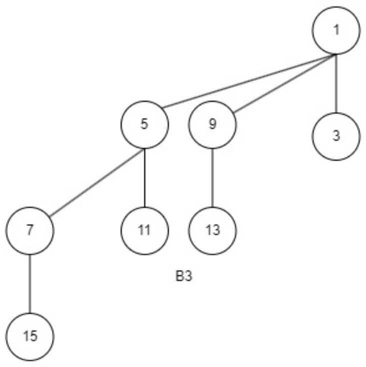
12. MERGE



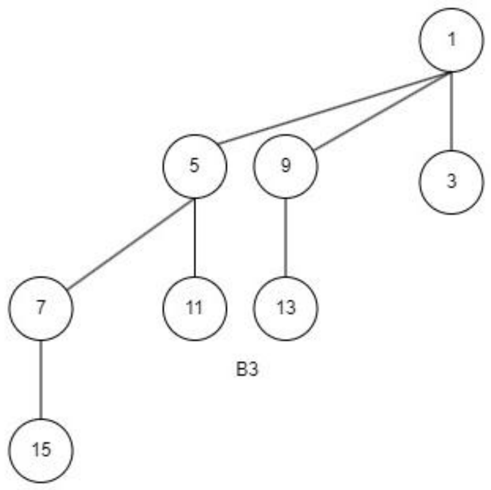
13. MERGE



14. FINAL H1

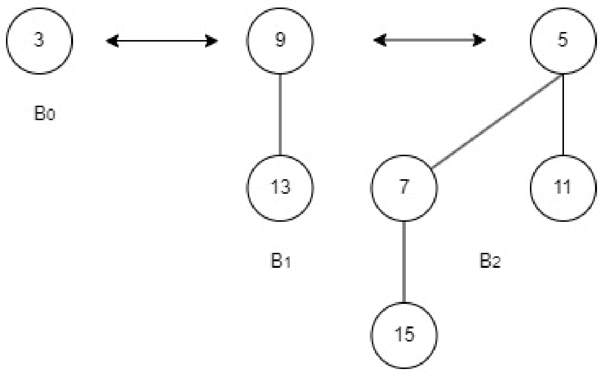


H1:



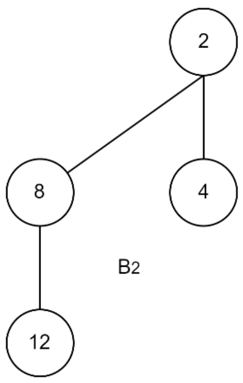
(b) Perform one deleteMin() operation on H1 to obtain H2

H2:



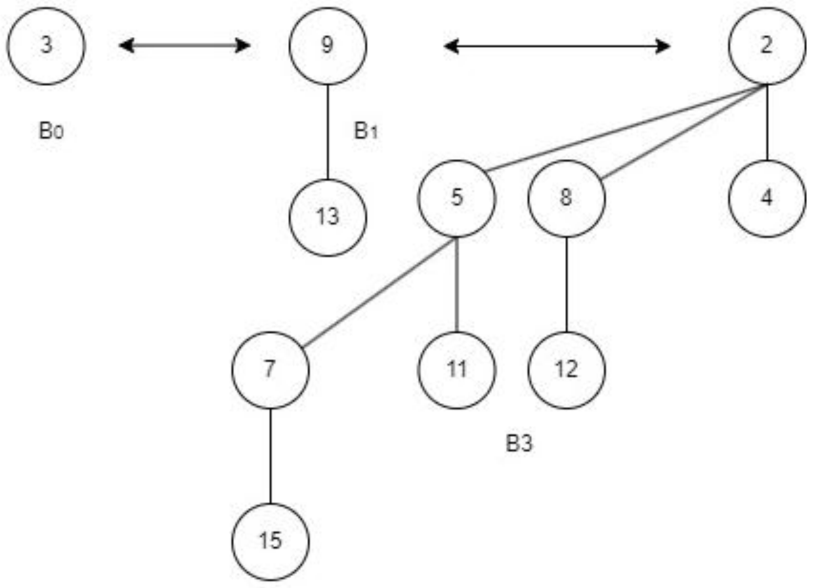
(c) Create another binomial min-heap H3 by inserting the following numbers from left to right

H3:



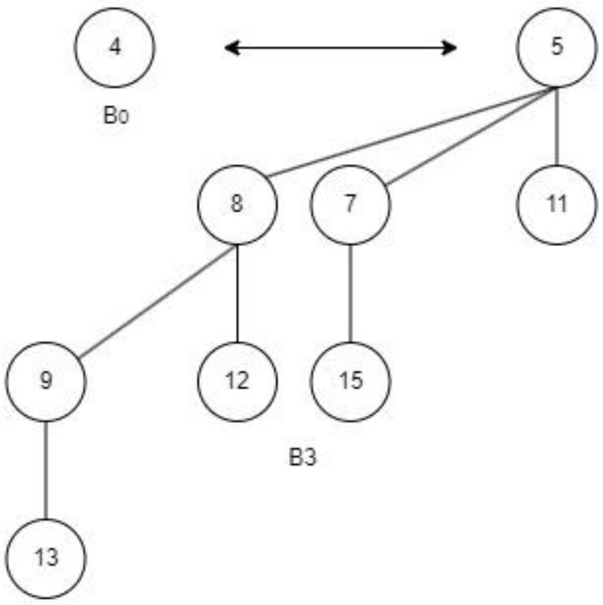
(d) Merge H2 and H3 to form a new binomial heap, H4

H4:



(e) Perform two deleteMin () operations on H4 to obtain H5.

H5:

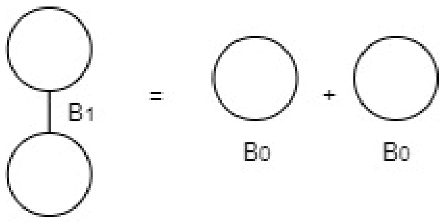


5. If we have a k-th order binomial tree (B_k), which is formed by joining two B_{k-1} trees, then when we remove the root of this k-th order binomial tree, it results in k binomial trees of smaller orders. Prove by mathematical induction. (10 points)

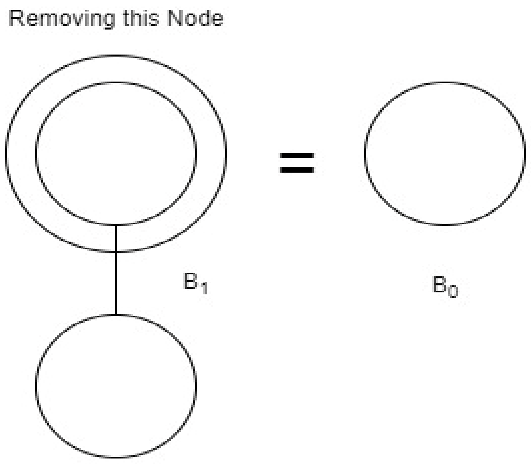
<u>ANS:</u>

Required to prove that if a k-th order binomial tree (B_k) is formed by joining two B_{k-1} trees, then when we remove the root of this $k-th$ order binomial tree, it results in k binomial trees of smaller orders.

<u>Base Case:</u> For $k = 1$



$$B_1 = B_0 + B_0$$



By above statement, B_1 is created by joining two B_0 nodes and removing the root node from the binomial tree of order B_1 we got 1 binomial tree of smaller order i.e., B_0. Hence, for Base Case the above statement is holds true.

Induction Hypothesis:

Let us assume that it is true for Binomial heap of k-th order i.e., if a Binomial heap B_k is created by joining two B_{k-1} trees, then if we remove the root node from the binomial tree of order B_k, we get k binomial trees of smaller order i.e., $k-1, k-2 \ldots..0$

=> B_k – Root Node = $B_{k-1} + B_{k-2} \ldots. + B_0$

$=>$ $B_k = B_{k-1} + B_{k-2} \ldots. + B_0$+ Root Node ……… (1)

Now, let us prove the same for $k+1$

=> $B_{k+1} = B_k + B_k$ …….. (2)

$=>$ $B_{k+1} = (B_{k-1} + B_{k-2} \ldots. + B_0$+ Root Node) + $(B_{k-1} + B_{k-2} \ldots. + B_0$+ Root Node) as per our hypothesis……… From (1)

By (2), if we merge Binomial trees of same order n we get one resultant tree with order $n+1$ applying this to merge corresponding terms in each () of the above equation

$=>$ B_{k+1} = ($B_k + B_{k-1} \ldots. + B_1$ +2 Root Nodes)

We need to prove that if we remove the root node for binomial tree of order B_{k+1}then it should yield $k+1$ binomial trees of smaller order.

$=>$ B_{k+1} = ($B_k + B_{k-1} \ldots. + B_1$ +2 Root Nodes)

$=>$ B_{k+1} – Root Node = ($B_k + B_{k-1} \ldots. + B_1$ + Root Node [B_0])

Hence proved that when a root node is removed from binomial tree of order B_{k+1} , it results in $k+1$ binomial tress of order $k, k-1, \ldots.., 0$.

6. Given a weighted undirected graph with all distinct edge costs. Design an algorithm that runs in $O(V + E)$ to determine if a particular edge e is contained in the minimum spanning tree of the graph. Pseudocode is not required, and you can use common graph algorithms such as DFS, BFS, and Minimum Spanning Tree Algorithms as subroutines without further explanation. You are not required to prove the correctness of your algorithm. (10 points)

ANS:

To determine if a particular edge e is contained in the minimum spanning tree (MST) of a weighted undirected graph with all distinct edge costs, we can use a modified version of Kruskal's algorithm. Kruskal's algorithm is typically used to construct the MST, but it can be adapted to check for the presence of an edge 'e' as well.

ALGORITHM TO DETERMINE PARTICULAR EDGE *e* is in MST:

STEP 1: Create a list of all edges in the graph sorted in increasing order of their weights.

STEP 2: Initialize a variable total_weight = 0, which will keep track of the total weight of the MST.

STEP 3: Initialize an empty set 'MST_Edges' to store the edges of the MST.

STEP 4: Iterate through the sorted edges:

STEP 5: For each edge (u, v) with weight 'w'

STEP 6: If adding this edge to the MST does not create a cycle, add it to "MST_Edges" list and increment total weight by w.

STEP 7: After iterating through all the edges, if the edge 'e' that has to be found is in the 'MST_Edges' set, then it is present in the MST.

This algorithm runs in $O(V + E)$ time, where V is the number of vertices and E is the number of edges, because it essentially constructs the MST using Kruskal's algorithm while checking for the presence of the edge 'e' in the process.

7. Given a weighted undirected graph with all distinct edge costs and $E = V + 10$. Design an algorithm that outputs the minimum spanning tree of the graph and runs in $O(V)$. Pseudocode is not required, and you can use common graph algorithms such as DFS, BFS, and Minimum Spanning Tree Algorithms as subroutines without further explanation. You are not required to prove the correctness of your algorithm. (10 points)

ANS:

GIVEN: A weighted undirected graph with all distinct edge costs such that E = V + 10 and each of the edges have distinct undirected weights for each of the edges.

ALGORITHM THAT OUTPUTS MINIMUM SPANNING TREE IN $O(V)$:

Find the minimum spanning tree using Prim's algorithm. As per the Prim's algorithm, we will traverse though all the vertices in the worst-case which is gives the run-time complexity of $O(V)$.

In reality, we only keep adjacent vertices i.e., Left and Right edges for each vertex. We may have several cycles, but for finding a spanning tree if we encounter a single cycle, we remove the largest edge in the cycle and we continue with the same procedure for rest of the cycles respectively (if there are any).

For each cycle, the size of the heap is only 2 vertices and hence it's a constant time $\approx O(1)$ to deal with each of such case as the size the heap is very small. And the number of cycles will be very small compared to number of vertices V.

Thus, we can conclude that we can conclude that the run-time complexity to dish out MST is linear time $\approx O(V)$ irrespective of the number of Edges whether it is $V + 1\ or\ V + 2\ or\ \ldots . V + 10$ etc as it takes 1 complete traversal using Prim's algorithm to find the MST and deal with cycles in constant time.

8. There are N people with the i-th person's weight being w_i. A boat can carry at most two people under the max weight limit of $M \geq \max_i w_i$. Design a greedy algorithm that finds the minimum number of boats that can carry all N people. Pseudocode is not required, and you can assume the weights are sorted. Use mathematical induction to prove that your algorithm is correct. (10 points)

ANS:

ALGORITHM FOR FINDING MINIMUM NUMBER OF BOATS REQUIRED TO CARRY *N* PEOPLE:

Given that we can assume that weights of N people are already in sorted order. Maximum people a boat can carry at a time is two, given that max weight limit is $M \geq max_i w_i$

STEP 1: Initialize two pointers $i_1 = 0$ *and* $i_2 = N - 1$, one in the beginning (lightest person) and one at the end (heaviest person). Also initialize a variable Number_of_Boats_Required = 0 to track the number of boats required to transport *N* people.

STEP 2: Do the following:

1. Check if the sum of weights of the two people $(w_{i1} + w_{i2})$ at the current positions of the pointers does not exceed the maximum weight limit M. If it doesn't exceed, it means these two people can share a boat, so move both pointers towards the midpoint of the list i.e., $i_1 = i_1 + 1$ *and* $i_2 = i_2 - 1$.

2. If the sum exceeds M, it means the heaviest person alone cannot share the boat with anyone, so you can only send the heaviest person in a separate boat. Move only the heavier person's pointer towards the center of the list. $i_1 = i_1$ *and* $i_2 = i_2 - 1$

For either of the case increment (+) the Number_of_Boats_Required by 1.

STEP 3: Repeat STEP 2 until the two pointers coincide or overlap. And the final value of the Number_of_Boats_Required is the number of boats required to transport following the constraint $M \geq max_i w_i$.

Now, let's prove the correctness of this algorithm using mathematical induction:

Base Case ($N = 1$): If there's only one person, he/she must go in one boat. The algorithm correctly returns 1 boat. Hence, algorithm holds for Base Case $N = 1$.

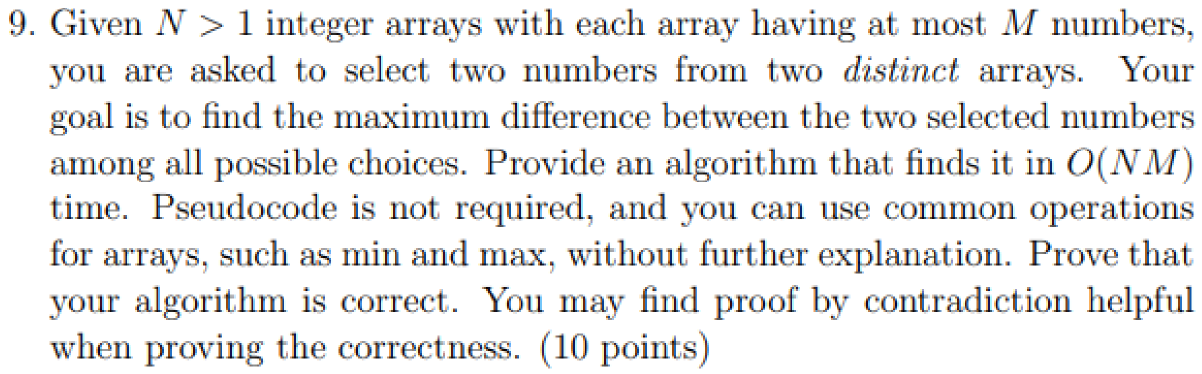
Inductive Hypothesis: Assume that the algorithm correctly determines the minimum number of boats for k people i.e., **($N = k$)**

For **($N = k + 1$)**

If the heaviest person with weight w_{i2} $(initially\ at\ k + 1, w_{k+1})$ can share a boat with the lightest person weight w_{i1} $(initially\ at\ 1, w_1)$, the algorithm will correctly identify this and move both pointers towards the center. This preserves the correctness by the inductive hypothesis.

If the heaviest person cannot share a boat with anyone (i.e., their weight $w_{i1} + w_{i2}$ exceeds M), the algorithm will correctly send the heaviest person in a separate boat, incrementing the boat count. This also preserves the correctness by the inductive hypothesis.

In either of the cases, the algorithm ensures to handle the $(k + 1)th\ case$ correctly. Therefore, by induction, the algorithm correctly determines the minimum number of boats needed to carry all k+1 people. Since it holds for the base case and the inductive step, it is correct for all values of N.



9. Given $N > 1$ integer arrays with each array having at most M numbers, you are asked to select two numbers from two *distinct* arrays. Your goal is to find the maximum difference between the two selected numbers among all possible choices. Provide an algorithm that finds it in $O(NM)$ time. Pseudocode is not required, and you can use common operations for arrays, such as min and max, without further explanation. Prove that your algorithm is correct. You may find proof by contradiction helpful when proving the correctness. (10 points)

ANS:

Given that we already have the array of elements, let's assume that the arrays be Given number of integer array = N.

Maximum number of elements possible in each array = M

Required to find: Maximum difference between 2 selected numbers provided the numbers are from the different array.

Given that we are already provided with the array of elements.

Let the arrays are represented as A_1, A_2,…, A_n.

Maximum or minimum element of particular array of size M can be found in O(M) time complexity.

All the respective maximums and minimums of the all the array can be found in O(MN) time complexity.

We will be taking the 4 variables. If the first_max and first_min are from the same array, we can obtain the maximum difference by taking the max(abs(first_max-second_min), abs(second_max-first_min))

ALGORITHM TO FIND MAXIMUM DIFFERENCE BETWEEN TWO SELECTED NUMBERS IN $O(NM)$:

STEP 1: Initialize variables first_max, second_max, first_min, second_min.

STEP 2: All the above variables are in the format (value, A_n)

STEP 3: Initialize the first_max, second_max to (-infinity, -infinity) and first_min, second_min to (infinity, infinity).

STEP 4: Initialize the value of i to 1. array_i corresponds to the ith array.

STEP 5: If i is less than or equal to N, iterate STEP 6 to 10.
Else go to STEP 11

STEP 6: Find max_i and min_i of array_i

STEP 7: If max_i is greater first_max, then second_max is equal to first_max and first_max is equal to max_i. Update the elements according to the array.

STEP 8: Increment the i value. Continue to STEP 5.

<u>STEP 9:</u> If max_i is greater than second_max then second_max is equal to max_i. Update the elements according to the particular array

<u>STEP 10:</u> Increment the *i* value. Continue to STEP 5

<u>STEP 11:</u>
If first_max [1] is not equal to first_min [1], then return absolute (first_max [0]-first_min [0]) and End the algorithm.
Else go to the STEP 12.

<u>STEP 12:</u> Return maximum (absolute (first_max [0]-second_min [0]), absolute (second_max [0]-first_min [0])) and End the algorithm.

<u>Proof of Correctness by Contradiction:</u>

Assume that the algorithm is incorrect and produces an incorrect result for a specific input case.

Now, let's consider this incorrect result:

Case 1: first_max and first_min come from different arrays, but the algorithm claims they are from the same array.

Case 2: first_max and first_min come from the same array, but the algorithm claims they are from different arrays.

We will show that both of these cases lead to contradictions:

For Case 1, if first_max and first_min come from different arrays but the algorithm claims they are from the same array, it means that the algorithm has incorrectly updated these values during its iterations, violating the correctness of the algorithm. This is a contradiction because we assumed the algorithm is incorrect.

For Case 2, if first_max and first_min come from the same array but the algorithm claims they are from different arrays, it means that the algorithm has incorrectly identified the source of these values. This also violates the correctness of the algorithm, leading to a contradiction based on our initial assumption.

Since both cases lead to contradictions, we can conclude that our initial assumption that the algorithm is incorrect must be false. Therefore, the algorithm is correct and will produce the desired result for any valid input.

10. There are N cities (city 1, to city N) and some flights between these cities. Specifically, there is a direct flight from every city i to city $2i$ (no direct flight from city $2i$ to city i) and another direct flight from every city i to city $i-1$ (no direct flight from city $i-1$ to city i). Given integers a and b, determine if there exists a sequence of flights starting from city a to city b. If so, find the minimum number of flights required to fly from city a to city b. For example, when $N = 10$, $a = 3$, and $b = 9$, the answer is 4 and the corresponding flights are $3 \to 6 \to 5 \to 10 \to 9$. You are not required to prove the correctness of your algorithm. (10 points)

ANS:

ALGORITHM TO FIND THE MINIMUM NUMBER OF FLIGHTS FROM a to b:

STEP 1: Number of Cities = N

Source = City a

Destination = City b

Flights_Required = 0 (to keep a track of number of flights required)

STEP 2: If $a > N \ or \ b > N$, then return Flights_Required = -1 directly as there will be no valid flight path (given constraint that there are only N).

STEP 3: If $a = b$, then return Flights_Required = 0

STEP 4: If $a > b$, then return $a - b$ is the only way to traverse back in the form of i to $i - 1$ until the destination and finally

STEP 5: If $a < b$, execute the below

STEP 6: Initialize a flag i.e., flag = 0 to regulate the loop

STEP 7: Create a queue and insert b into the queue.

STEP 8: If size of the queue > 0, perform STEPS 9 to 19.

STEP 9: Increment (+) the Flights_Required by 1.

STEP 10:

If size > 0, Perform STEPS 11 o 17.

Else go to STEP 18.

STEP 11: Perform dequeue operation and store the value in a temporary variable called temp.

STEP 12:

If temp%2 = = 1 (is odd) and temp+1 ≤ N, go to STEP 13. Else, go to STEP 14.

STEP 13:
If temp + 1 = = a, make flag=1 and go to STEP 18.
Else enqueue temp+1 to the queue and Skip the STEPS 14 to 17.

STEP 14:

If temp%2 = = 0 (is even) and a = = temp/2, update flag = 1, go to STEP 18.
Else go to STEP 15.

STEP 15: Enqueue temp/2 to the queue.

STEP 16: If temp%2 = = 0 (is even) and temp+1 ≤ N, go to STEP 17.

STEP 17:
If a = = temp+1, update flag = 1 and go to STEP 18.
Else enqueue temp+1 to the queue. Decrement (-) the size by 1. Go to STEP 10.

STEP 18: Update the size of the queue to the new size. Size = queue.size().

STEP 19:
If flag = = 1 go to STEP 20.
Else go to STEP 8.

STEP 20:
If flag = 1, return Flights_Required.
Else return -1.

This algorithm computes the minimum number of flights required in < $\boldsymbol{O(N)}$

If we want to go for a standardized approach, we can construct a directed graph and perform BFS from $a\ to\ b$ to find the minimum number of flights required. If no paths, algorithm will return -1. But the time complexity of this graph approach will be $\approx O(N)$

******************************** **THE END** **********************************