CSCI570 Homework 2

Name: Saunak Sahoo   USC ID: 3896400217

Ans 1

To schedule classes such that we have minimum number of halls utilized, we need to maximize the number of classes that can be scheduled parallelly. So, in our algorithm we would be finding out the maximum number of parallel classes that can be scheduled which would also be the minimum number of halls utilized

a) Algorithm:

```
1. Insert all classes into an array and sort them in ascending order based on start_time
2. Initialize 2 variables parallelClassCount = 0, maxParallelClasses = 0
3. Initialize an empty MinHeap
4. For each class in array of classes:
    a. If MinHeap is empty:
        i. Insert class.end_time into MinHeap
        ii. parallelClassCount  = parallelClassCount  +1
    b. Else If MinHeap is not empty:
        i. if root of MinHeap (end_time) ≤ class.start_time
            1. deleteRoot() from MinHeap
            2. parallelClassCount = parallelClassCount - 1
            3. Go to 4a
        ii. Else If root of MinHeap (end_time) > class.start_time
            1. Insert class.end_time into MinHeap
            2. parallelClassCount = parallelClassCount  + 1
    c. If maxParallelClasses < parallelClassCount
```

        i. maxParallelClasses = parallelClassCount

5. Return maxParallelClasses

b) Time Complexity

Time complexity would be calculated on the most expensive operations of the algorithm.

The sorting in step 1 takes O($n \log n$) time

For the loop in step 4, in the worst case we may have the operations deleteRoot() and insert into MinHeap that run for each of the n classes. These operations have time complexity O(log n)

Hence time complexity = O($n \log n$) + $nX2XO(\log n)$ = O($n \log n$)

Ans 2

Assume there are n arrays where each array contains the research papers published by a department/institute ordered in ascending order. We are also assuming that each research paper would have information that would make it possible to point back to the department that published it

a) **Algorithm:**

1. For each of the n arrays, pick out the first research paper and push it into a MinHeap
2. Initialize an array to store the resultant sorted research paper. Let us call this array **Result** and initialize it to size m
3. While MinHeap is not empty:
    a. Perform deleteMin() on the MinHeap and push this research paper into the Result array
    b. Check the array of the department from which the above research paper originated and push the next research paper into the MinHeap (if it exists)
4. Return the Results array as the sorted list of research papers from all departments

b) Time Complexity:

Time complexity of the algorithm is determined by the most expensive step. In our algorithm, the loop in step 3 would determine the overall time complexity of the algorithm

The loop would run once for the insertion of all m research papers. We know that the time complexity for deleteMin() for a heap is O(log n)

Hence, time complexity = m X O(log n) = O($m \log n$)

Ans 3

We will traverse through all the space stations until we reach a space station that cannot be reached with the amount of currentFuel. In such a scenario we shall see what was the station that is reachable that had the highest amount of fuel and make a stop there

a) **Algorithm:**

```
1. Sort the array of space stations in ascending order of
   distanceToStationFromOrigin
2. Initialize refuelStops = 0 and filledFuel = currentFuel
3. Initialize an empty MaxHeap
4. For each spaceStation in array of space stations:
   a. If spaceStation.distanceToStationFromOrigin ≤ filledFuel (i.e. the space station
      is reachable with the current amount of fuel):
      i. Insert spaceStation.fuelCapacity to the MaxHeap
   b. Else If spaceStation.distanceToStationFromOrigin > currentFuel:
      i. If the MaxHeap is empty (i.e. there are no stations that can be reached with
         the current amount of fuel):
         1. return -1
      ii. Else:
         1. filledFuel = filledFuel + MaxHeap.root
         2. deleteRoot() from MaxHeap
         3. refuelStops = refuelStops + 1
```

```
            4. Go to 4a with the current spaceStation (to check whether this
               spaceStation can be reached with the currentFuel or whether we'll have
               to stop at a next highest fuelCapacity space station)
5. If targetDistance > currentFuel:
    a. If the MaxHeap is empty (i.e. there are no stations that can be reached with the
       current amount of fuel):
        i. return -1
    b. Else:
        i. filledFuel = filledFuel + MaxHeap.root
        ii. deleteRoot() from MaxHeap
        iii. refuelStops = refuelStops + 1
        iv. Go to 5

   Else If targetDistance ≤ currentFuel:
       return refuelStops
```

b) Time Complexity

Time complexity would be calculated on the most expensive operations of the algorithm.

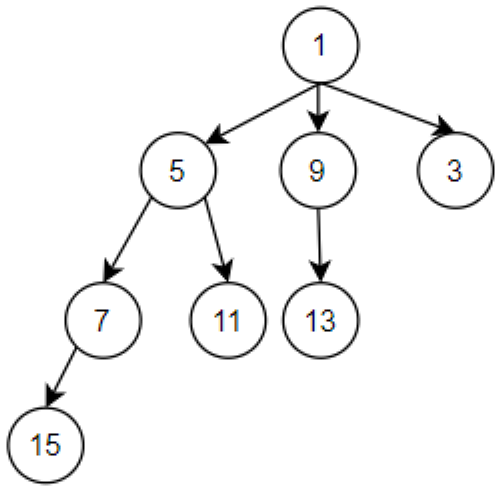
The sorting in step 1 takes O($n \log n$) time

For the loops in steps 4 and 5, in the worst case our space craft would stop to refuel at every spaceStation. So in this scenario, we would stop at every n space stations. We would have to run insert and deleteRoot() operations in our MaxHeap n number of times with each operation taking O($n \log n$) time.

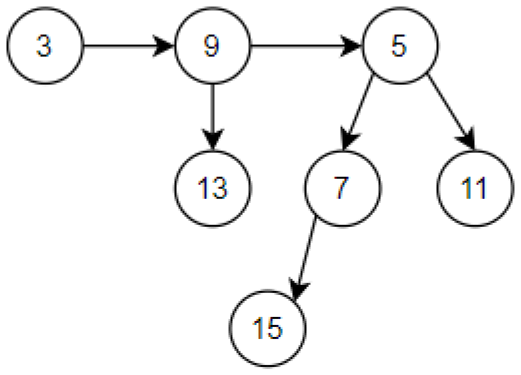
Hence time complexity = O($n \log n$) + 2 X n X O($n \log n$) = O($n \log n$)

Ans 4

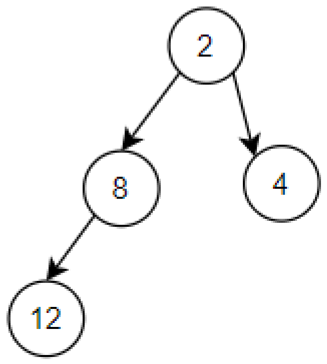
a) Binomial heap H1 after insertions:



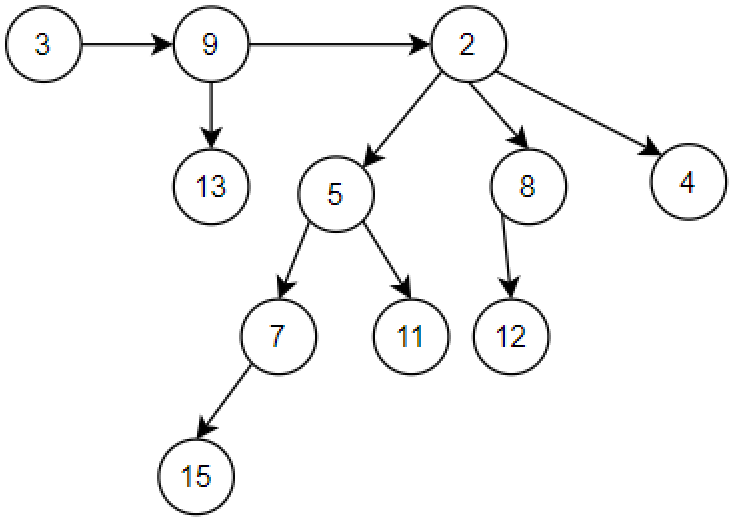
b) Binomial heap H2 after deleteMin() on H1



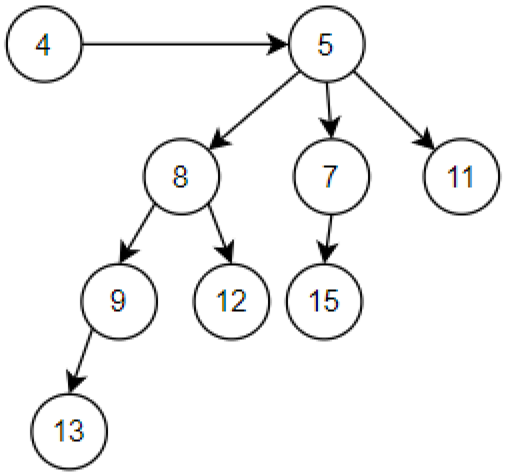
c) Binomial heap H3 after insertions:



d) Merging H2 and H3 to form H4



e) Binomial heap H5 after 2 deleteMin() operations on H4



Ans 5

Proof by Mathematical Induction:

Base case:

For k=0, B_0 is a single node with no edges. On deleting the root node (i.e. the only node) we will get 0 binomial trees = k. Hence the base case holds true

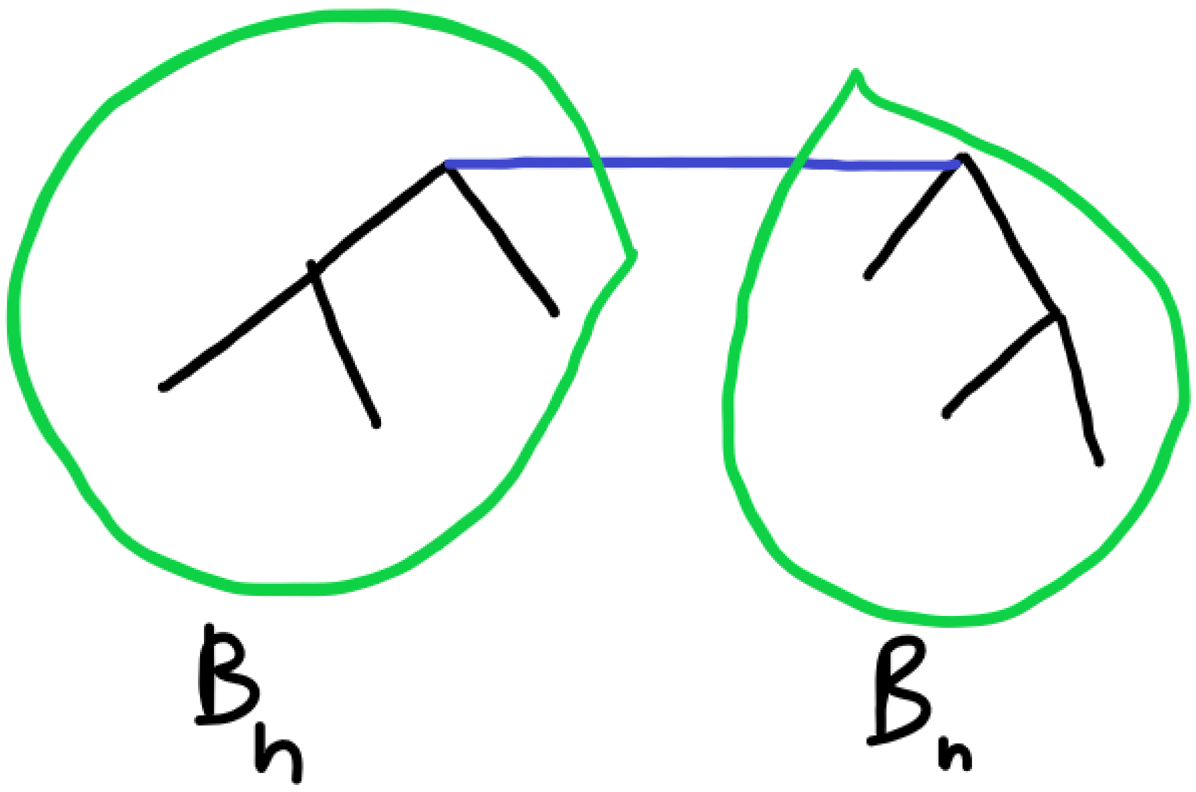
Inductive Hypothesis:

Say the algorithm holds true for k=n i.e. B_n breaks into n trees of smaller orders

Inductive Step:

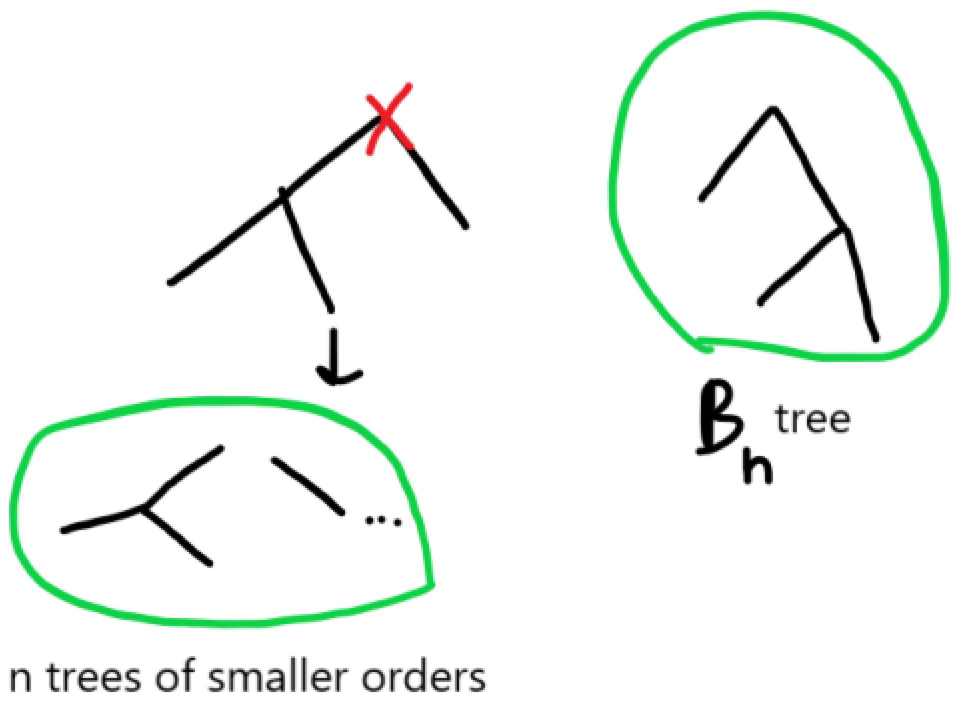
To prove: B_{n+1} breaks into n+1 trees of smaller orders

A B_{n+1} tree is made by joining 2 B_n trees:



When we delete the root of the B_{n+1} tree, it breaks the 2 B_n trees into:

- One B_n tree AND
- The other B_n tree would break into n trees of smaller orders (from our Inductive Hypothesis)



Total number of trees = n trees of smaller orders + One B_n tree = n + 1

Hence we have proven our Inductive Step and the statement holds true

Ans 6

We know that for an edge e to be part of the MST, it has to be part of the minimum weighted edges that can connect all vertices without creating cycles

Algorithm:

1. Delete all edges from the graph whose weight is greater than the weight of edge e
2. Run a traversal algorithm such as DFS on the graph to check whether there are vertices that are now disconnected
3. If graph is now disconnected:
   Edge e is part of the MST, because it is required to keep the graph connected

   Else If the graph remains connected:
   Edge e is not part of the MST, as there is some other edge smaller than e which is keeping the graph connected in an MST

In this algorithm the traversal algorithm DFS has time complexity O(V+E)

Ans 7

We are given a graph with E = V+10. We know that an MST would include the minimum weight edges such that all vertices are covered without any cycles, so for an MST E=V-1

So for our graph to become an MST, we have to delete (V+10) - (V-1) = 11 edges

A graph with E=V+10 is going to have cycles. We know that an MST cannot have cycles, hence we have to delete 11 edges from the cycles that are contained in the graph

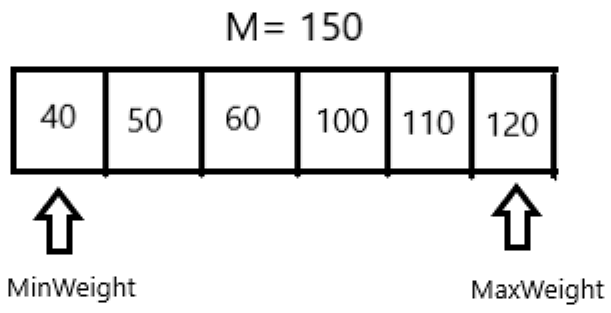
In order to detect cycles in the graph in linear O(V) time, we have to use DFS or BFS traversals. Here we shall use DFS to detect a cycle by running DFS from every unvisited node and storing all vertices visited in an array. If there is an edge to an already visited node, we record that there's a cycle.

Algorithm:

1. Run the following 11 times to delete 11 edges:
    a. Run DFS on the graph and detect a cycle
    b. For the above cycle find the edge with the highest weight and delete it

Ans 8

To minimize the number of boats required, we should maximize the weight that each boat can carry. To do this we would use greedy algorithm to pair 2 people such that their combined weight is as close as possible to M (i.e. the maximum weight capacity of the boat)



To achieve this we sort the N people in increasing order of their weights into an array **Arr** (such as the example given above). The array has 2 pointers:

- **MinWeight:** Initially this would point to the 0th index of the array i.e. the person with the minimum weight
- **MaxWeight:** Initially this would point to the last index of the array i.e. the person with the maximum weight

Additionally we will also have a variable counter **BoatCount** initialized to 0

Algorithm:

While MinWeight < MaxWeight perform the following steps:

1. If MinWeight = MaxWeight:
   This is the scenario in which both the pointers are pointing to the same person. This person will be assigned 1 boat.
   MinWeight pointer will be moved to the right (MinWeight = MinWeight + 1) and MaxWeight pointer will be moved to the left (MaxWeight = MaxWeight - 1)

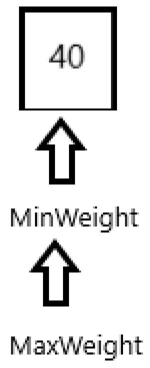
   Else If Arr[MinWeight] + Arr[MaxWeight] > M
   In this scenario the person with the MaxWeight cannot be paired with any other person who has a suitable MinWeight. This person with the MaxWeight will be assigned 1 boat.
   The MaxWeight pointer will be moved to the left (MaxWeight = MaxWeight - 1)
   {For the example above, 40 + 120 > 150, so the person with weight 120 will be assigned 1 boat}

   Else if Arr[MinWeight] + Arr[MaxWeight] ≤ M
   The 2 persons having MinWeight and MaxWeight will be paired up and assigned 1 boat. MinWeight pointer will be moved to the right (MinWeight = MinWeight + 1) and MaxWeight pointer will be moved to the left (MaxWeight = MaxWeight - 1)
2. Increment **BoatCount** (BoatCount = BoatCount + 1)

Proof by mathematical induction:

Base Case:

For N=1: There is only one person in the array in this scenario



Both the **MinWeight** and **MaxWeight** pointers will be pointing to the 0th index.
Here **MinWeight = MaxWeight** and so this person will be assigned 1 boat. **BoatCount** will be incremented (BoatCount = BoatCount + 1 = 1)
MinWeight = MinWeight + 1 = 1; **MaxWeight = MaxWeight** - 1 = -1

Now since, MinWeight > MaxWeight, the algorithm would terminate with **BoatCount** = 1 which is the optimal solution for N=1

Inductive Hypothesis:

Assume that the algorithm gives the optimal solution for ‘n’ number of people

Inductive Step:
To prove: The algorithm gives the optimal solution for n+1 people

We already know from the Inductive Hypothesis that the algorithm gives the optimal solution for n number of people, so for the (n+1)th person there may be 2 cases:

Case 1: If all of the n previous people have been assigned boats (i.e. the algorithm has reached the stage MinWeight = MaxWeight), the (n+1)th person would be assigned the last boat

Case 2: There is one other person left to be assigned a boat. In this case, this person would pair up with the (n+1)th person and be assigned the last boat

In both cases we get the most optimal solution for n+1 people

Ans 9

Say we have N arrays of at most M elements. The maximum difference between any 2 elements in any 2 arrays would be the maximum difference between the max and min elements of any of the arrays. Hence, we can solve this using greedy algorithm

Algorithm:

1. For each of the N arrays:
    a. Find max element and store is in an array **MaxArr**[]
    b. Find min element and store is in an array **MinArr**[]
2. Find max element of MaxArr[] and store it as **MaxElement**. Also store its index as **maxIndex**
3. Find min element of MinArr[] and store it as **MinElement.** Also store its index in **minIndex**
4. If **maxIndex ≠ minIndex**
    a. Return maximum difference as MaxElement - MinElement
    [Since the MaxElement and MinElement belong to distinct arrays, our answer would be this difference]

    Else if **maxIndex = minIndex**:

    proceed with remaining steps
5. Assign MaxArr[maxIndex] = $-\infty$ and MinArr[minIndex] = ∞ (we have done this step so that the element present at maxIndex and minIndex are no longer considered while computing the max and min of MaxArr and MinArr respectively)
6. Find highest element of MaxArr[] and store it as **MaxElement2**. Compute MaxElement2 - MinElement and store it as **Difference1**
7. Find second lowest element of MinArr[] and store it as **MinElement2**. Computer MaxElement - MinElement2 and store it as **Difference2**

8. If Difference1≥Difference2
   Return maximum difference as Difference1

   Else
   Return maximum difference as Difference2

We had to perform steps 5-8 for the case where the MaxElement and MinElement belonged to the same array (since we want maximum difference from *distinct* arrays)

Proof by contradiction:

Assume that there exist 2 numbers x and y in two distinct arrays such that:

$max - min < x - y$
where max is the maximum element from all arrays and min is the minimum element from all arrays

The above inequality will hold true in 3 cases:

Case 1: $x > max$ and $y = min$
However, this is not possible since there cannot be any element greater than the max element

Case 2: $y < min$ and $x = max$
However, this is not possible since there cannot be any element lesser than the min element

Case 3: $x > max$ and $y < min$
However, this is not possible since there cannot be any element greater than the max element and any element lesser than the min element

Hence, we have proven that $max - min \geq x - y$

Ans 10

Instead of going from a→b using subtraction with 1 and multiplication with 2, we will look at the number of flights required to go from b→a using the inverse operations of addition with 1 and division with 2. In other words we shall use greedy algorithm to go from b→a

Algorithm:

1. Initialize a variable **CurrentCity** and initialize it with the destination city i.e. b
2. Initialize a counter **Flights** and initialize it to 0
3. While CurrentCity ≠ source city i.e. a perform the following:
   a. If CurrentCity < a
      CurrentCity = CurrentCity + 1

      Else If CurrentCity is odd (CurrentCity modulo 2 ≠ 0):
      CurrentCity = CurrentCity + 1

      Else if CurrentCity is even (CurrentCity modulo 2 == 0)
      CurrentCity = CurrentCity/2
   b. Increment Flights counter by 1
4. Return the Flights counter as the total number of flights required to go from $a \to b$

Example 1: a=3, b=9
Iterations of CurrentCity = 9, 10, 5, 6, 3
Hence the sequence of flights is: 3→6→5→10→9 with Flights = 4

Example 2: a=6, b=3
Iterations of CurrentCity = 3, 4, 5, 6
Hence the sequence of flights is: 6→5→4→3 with Flights = 3