NAME: NIKHIL SATHYANARAYANA SHASTRY
USC ID: 8464475401
Email: nsathyan@usc.edu

CSCI570 - Analysis of Algorithms (HW2)

---

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)

Solution:

a. Algorithm to determine the minimum number of lecture halls required:

1. Sort the input list intervals based on the class end times in ascending order.
2. Now, let us initialize an empty min-heap to track the scheduled classes' end times.
3. Iterate through the sorted intervals one by one:
    a. If the min-heap is empty, allocate a new lecture hall for this class and **insert** its end time into the min-heap.
    b. If the min-heap is not empty and the smallest end time in the heap (i.e. the top-most element in the min-heap) is lesser than or equal to the start time of the current class in the list, it means that the current class can now reuse that particular lecture hall without any overlap. Hence, we can **deleteMin** the smallest end time from the heap and **insert** the end time of the current class into the heap.
    c. If the min-heap is not empty and the smallest end time in the heap is greater than the start time of the current class, it means that there are no lecture halls available for the current class. Hence, in this case, we allocate a new lecture hall for the current class and **insert** its end time into the min-heap.
4. The size of the min-heap at the end of the iteration represents the minimum number of lecture halls needed to accommodate all classes without any overlap.

b. The time complexity analysis is as follows:

- The time complexity for sorting the input interval array will be $(O(nlogn))$.
- The time complexity for executing the push and pop operations in the loop n times will take $(O(nlogn))$.

- So the overall time complexity the combination of the 2 time complexities $O(n\ log\ n) + \ O(n\ log\ n) \approx O(n\ log\ n)$.

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)

Solution:

a. Algorithm to efficiently merge the given sorted lists by taking into account the total number of research papers (m) and the number of departments (n) using binary heaps is as follows:

1. Create an empty min-heap H that will store tuples of (paper publication date, department index) and the min ordering is with respect to the publication date. (Earlier the date, closer it is to the root)
2. For each department i = 1 to n:
    - Insert the first paper's (*publication date, i*) tuple into H
3. Initialize a result array Merged[] of size m
4. For i = 1 to m:
    - Extract the min tuple (*date*, *dept*) from H
    - Add the paper with date to the next position in Merged[]
    - Insert the next paper from that particular department *dept* into H
5. Return the result array Merged[] containing all m papers sorted by publication date

b. The time complexity analysis is as follows:

- Inserting the first paper from each of n departments into the heap takes $O(n\ log\ n)$ time
- There will be m **deleteMin** extractions and insertions on the heap, each taking $O(log\ n)$ time so the total time complexity at this stage will be O($m\ log\ n$)
- So the overall worst case time complexity is $O(n\ log\ n\ +\ m\ log\ n) \approx\ O(m\ log\ n\ )$ because $m >> n$.

3. 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)

Solution:

a) Algorithm to efficiently calculate the ***refuelStops*** needed for a successful voyage to the target star using binary heaps is as follows:

1. **Initializing variables:**

    A. Initialize ***currentPosition*** to **0** (the celestial origin) and the ***refuelStops*** to **0** (for counting the number of refuelling stops used)

    B. Create a max-heap called ***maxFuelHeap*** to keep track of the available fuel capacities at each fuel station.

2. **Iterate Through Space Stations:**

    A. Iterate through the space stations in ascending order of ***distanceToStationFromOrigin***.

    B. For each space station:

    1. Check if the spaceship can reach the station without refuelling:

        - While ***currentPosition*** + ***currentFuel*** is less than ***distanceToStationFromOrigin***, it means the spaceship can't reach the station without refueling.
        - In this case, we need to pop stations from ***maxFuelHeap*** (if available) and refuel the spaceship until it can reach the current station.
        - Increment ***refuelStops*** for each refueling operation.

    2. If ***maxFuelHeap*** is empty (i.e., there are no stations with enough fuel to reach the next station), return -1 because it's impossible to reach the target star.

3. Push the current station's fuel capacity into ***maxFuelHeap*** since we can refuel at this station if needed.
4. Update ***currentPosition*** to the current station's ***distanceToStationFromOrigin***.

3. Check if We Can Reach the Target Star:

A. After looping through all the stations, check if the spaceship can reach the target star (***currentPosition + currentFuel >= targetDistance***).
B. If it can, return ***refuelStops*** as the minimum number of refueling stops required for a successful voyage.

4. Return -1 If Not Possible:

A. If the spaceship couldn't reach the target star with the available fuel, return -1 to indicate that the journey is not possible.

b) The time complexity analysis is as follows:

- The time complexity iterating through the list of n stations is ***O(nlog(n))***.
- The time complexity for performing insert and delete operations on a binary max-heap in the worst case is ***O(log(n))***.
- Hence, the overall time complexity becomes ***O(nlog(n))*** since we perform these heap operations within the loop that iterates through all ***n*** stations.

4. You are tasked with performing operations on binomial min-heaps using a sequence of numbers. Follow the steps below:

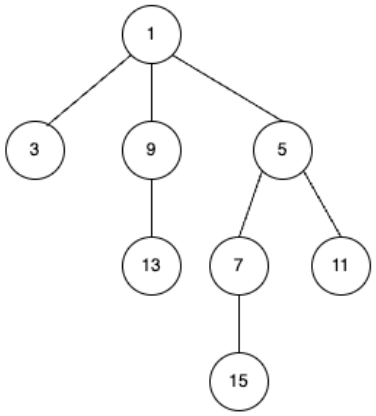
   a. Create a binomial min-heap H1 by inserting the following numbers from left to right: 3, 1, 13, 9, 11, 5, 7, 15. (2 points)
   b. Perform one deleteMin() operation on H1 to obtain H2. (2 points)
   c. Create another binomial min-heap H3 by inserting the following numbers from left to right: 8, 12, 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.

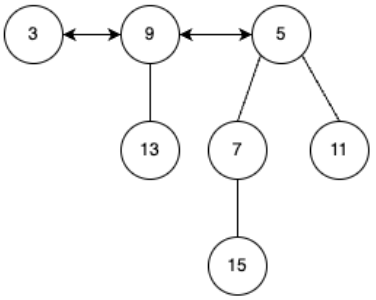
Solution:

PLEASE REFER THE NEXT PAGE

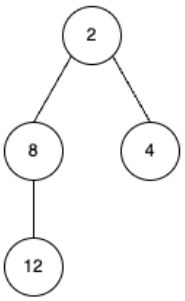
H1


1
3
9
5
13
7
11
15

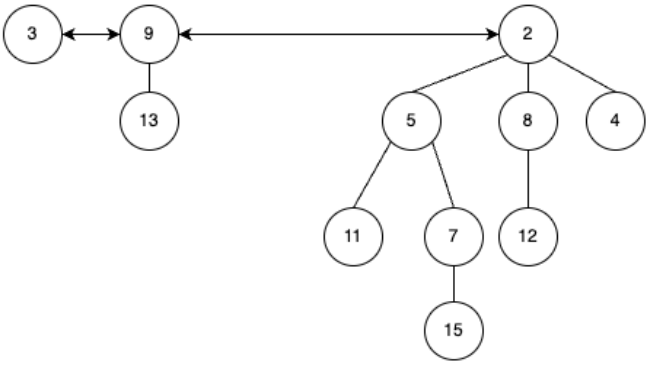
H2


3
9
5
13
7
11
15

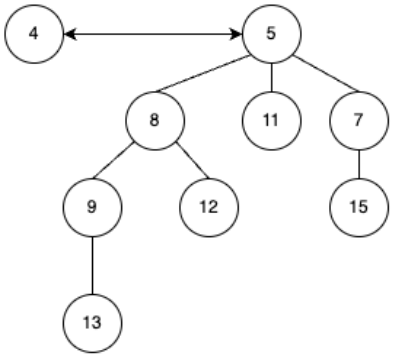
H3


2
8
4
12

H4


3
9
2
13
5
8
4
11
7
12
15

H5


4
5
8
11
7
9
12
15
13

5. If we have a k-th order binomial tree (Bk), which is formed by joining two Bk−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)

Solution:

By making use of mathematical induction, we need to prove that removing the root of a k-th order binomial tree results in k binomial trees of smaller orders.

Base case:

- For k = 1, Removing the root of the 1st order binomial tree gives rise to 1 binomial tree of zero order.

Induction Hypothesis:

- Now, let us assume that the given statement is true for all positive integers k <= n-1. Therefore, removing the root of an (n-1)-th order binomial tree leads to the formation of (n-1) binomial trees of smaller orders.

Induction Step:

Now, we need to prove that if our hypothesis holds for k = n - 1, it also holds true for k = n.

- Let us consider a (n)-th order binomial tree, which can be denoted by B_n, and it is formed by joining two (n-1)-th order binomial trees, B_{n-1} and B_{n-1}. Now let us rename the two B_{n-1} trees as B and B' and assume the root belongs to tree B. Now, when we remove the root of B_n, We obtain (n-1) binomial trees of smaller order from tree B alongside B', as supported by our induction hypothesis.
    - These (n-1) binomial trees have smaller order than B_{n-1}, hence will surely have a lesser order than B_n. The order of B' is also (n-1) which is less than that of B_n. In total, we have (n-1) + 1 = n trees of lower order.

- Hence, removing the root from an n-th order binomial tree resulted in n binomial trees of lower order.

Hence, we have shown that removing the root of a (k)-th order binomial tree results in (k) binomial trees of smaller orders. This completes our proof by mathematical induction.

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)

Solution:

➔ Algorithm to determine if a particular edge e is contained in the minimum spanning tree of the graph in O(V + E) is as follows:

- **Remove Edge 'e' and edges with Greater Weight than 'e':**
  - Start by removing the edge 'e' from the original graph.
  - Remove any other edges in the graph that have a greater weight than edge 'e'. We can do this by iterating through the edges of the graph and removing those that don't meet this condition.
- **Run Depth-First Search (DFS):**
  - Now, we choose one of the endpoints of edge *'e'* (in our example let's call it *u*) and run a Depth-First Search (DFS) from *u* in the modified graph. DFS explores all reachable vertices from *u*.
- **Check Reachability to the Other Endpoint:**
  - After running DFS, check if the other endpoint of edge *'e'* (in our example let's call it *v*) is reachable from *u*. If you can reach *v* from *u* in the modified graph, it means there is a path between *u* and *v* that doesn't include edge *'e'*.
- **Result:**
  - If you can reach *v* from *u*, then the edge *'e'* is not part of the MST because there exists an alternative path in the modified graph to connect *u* and *v*.
  - If you cannot reach *v* from *u*, then edge *'e'* is part of the MST because there is no other path between *u* and *v* in the modified graph, and removing *'e'* would disconnect them.
- This algorithm would work because here we are testing if the edge *'e'* can be safely removed without disconnecting the graph. If removing *'e'* doesn't disconnect the endpoints *u* and *v*, it implies that *'e'* is not part of the MST. Conversely, if removing *'e'* disconnects *u* and v, it implies that *'e'* is part of the MST.
- This method can efficiently determine whether *'e'* is part of the MST with a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph. This is because the DFS traversal takes O(V + E) time.

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)

Solution:

➔ Our strategy involves applying Depth-First Search (DFS) to the graph to identify any cycles and then storing the edge with the highest weight within that cycle so that we can proceed to remove this particular edge from that cycle. By employing this method iteratively, we aim to eliminate edges until we have precisely V-1 edges remaining in the graph.

➔ Algorithm to Find Minimum Spanning Tree (MST) in a Weighted Undirected Graph with E = V + 10 in O(V) Time is as follows:

- Repeat the following steps 11 times:
    - Maintain a record of the cycle's maximum weight edge as ***maximumWeightEdgeInCycle*** and its corresponding maximum weight as ***maxWeight***, initializing both to negative infinity (-∞).
    - Perform a Depth-First Search (DFS) on the provided graph to identify any cycles. If a cycle is detected during the DFS traversal, iterate through the cycle once more, updating the values of ***maxWeight*** and ***maximumWeightEdgeInCycle*** as necessary.
    - Delete ***maximumWeightEdgeInCycle*** from the cycle.
- The time complexity analysis is as follows:
    - Initializing the variables in the first step takes O(1)
    - In Step 2, we employ a Depth-First Search (DFS) operation, which takes O(V+E) time. The process of identifying the ***maximumWeightEdgeInCycle*** requires O(E) time. Given that E= V+10, this step's time complexity becomes O(V+E) + O(E), which simplifies to O(2V+10) + O(V+10), ultimately resulting in O(V) as the time complexity.
    - Deleting the ***maximumWeightEdgeInCycle*** takes O(1)
    - Since all these steps take place 11 time the final complexity is 11*[O(1)+O(V)+O(1)] = **O(V)**

8. There are N people with the i-th person's weight being wi. A boat can carry at most two people under the max weight limit of M ≥ maxi wi. 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)

Solution:

➔ The greedy algorithm to find minimum number of boats to carry all N people given that the weights are sorted is as follows:

- **Step 1:** Initialize a variable *count* to count the number of boats. Initially it's value is set to **zero**.
- **Step 2:** We need to maintain 2 pointers, one at the left end of the weights array (*w*) and another at the right end. These *left* and *right* pointers are initalized to 0 and *n-1* respectively (Here *n* is the length of the weights array)
- **Step 3:**

  While left ≤ right:

  A. $w_{left} + w_{right} \leq$ M, it means both the people can be accommodated in a single boat. So, increment the *left* and decrement the *right* pointers.

  B. $w_{left} + w_{right} >$ M, it means the person with the heaviest weight (w_{right}) cannot share a boat with anyone. In this case, only the person at the *right* pointer takes a boat. Then we need to decrement the *right* pointer.

  C. Increment the *count* for each boat taken.

- **Step 4:** Return the *count*

➔ **Proof of correctness using Mathematical Induction:**

Base case:

- Let us consider N = 1. We have to carry only one person with weight w_0 in the boat. The weight of w_0 can either be greater than the weight of the boat M or lesser.

- ***For N = 1:*** *left* = 0, *right* = (N-1) = 0. Since, *left* ≤ *right*, the algorithm enters the while loop.

  1. If $2w_o \leq$ M, *count* becomes 1 and loop terminates as left is 1 now and right is -1.
  2. If $2w_o >$ M, *count* still becomes 1 and loop terminates because left is 0 and right is -1.

- The algorithm returns 1 in both cases as expected.

Induction Hypothesis:

- Assume that the algorithm works correctly for all $N \leq (k-1)$ where k is a set of all positive integers. Hence assume that the algorithm holds good for N = k-1 and N = k-2.

Induction Step: Now we need to prove the algorithm works correctly for N = k:

- **For N= k:** *left* = 0, *right* = (k-1). Now *left* ≤ *right* holds true, hence the algorithm enters the while loop.

    1. If $w_o + w_{k-1}$ <= M, the lightest and heaviest person are sent in the same boat. *count* will be 1. Next, *left* is incremented 1 and *right* is decremented to k-2. Now, the problem has been reduced to the size of k-2. The induction hypothesis shows that the algorithm works correctly for N = k-2. Hence, the algorithm works as expected in this case.

    2. If $w_o + w_{k-1}$ > M, then the heaviest person will be individually sent in one boat. *count* becomes 1. *left* stays 0, but *right* is decremented to k-2. The problem is now reduced to the size of k-1. The induction hypothesis shows that the algorithm works correctly for N = k-2 Hence, the algorithm will work as expected in this case as well.

Hence, this shows the proof of correctness of our algorithm.

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)

Solution:

➔ Algorithm to find the maximum difference between the two selected numbers among all possible choices in O(NM) is as follows:

- Firstly, we need to initialize 2 arrays which store the maximum and minimum values along with the indices of the array to which it belongs. Let's name them ***maxArray*** and ***minArray.*** ⇒ ***O(1)***
- Now, let us iterate through all the N arrays. For each array in Arrays[i]: ⇒ **O(NM)**
    - Create a tuple - (max(Arrays[i]),i) and add it to the maxArray at index i.
    - Create a tuple - (min(Arrays[i]),i) and add it to the maxArray at index i.
- After our ***maxArray*** and ***minArray*** are filled we can run a heapify function on it which creates a max-heap and a min-heap respectively in **O(N)** time (As taught in the lecture).
- Once the heapify is done we need to get the root from both the max-heap and the min-heap and check if both of them are from the same array. We can achieve this by checking whether the 2nd value of their tuples are the same.
    - If the indices of the roots of the max and min heap are not equal, it means that we have got the global max and min value. Subtracting the two roots will give us the maximum difference between the two selected numbers among all possible choices and this can be returned as the answer.
    - If the indices of the two root elements are the equal, we need to do a pop on both the arrays to extract the second smallest and second largest values from both min and max heaps respectively. Now- we can get the maximum difference by doing *max (largest - second smallest, second largest - smallest)* and return the answer. (Popping elements in the heaps take **O(logn)** complexity)
- The overall time complexity is O(1) + O(NM) + O(N) + O(log(N)) = **<u>O(NM)</u>**.

➔ Next, let us prove by contradiction the correctness of our algorithm:

- Let's assume the algorithm does **NOT** return the maximum difference. Then, there exists 2 other values, lets call them *a* and **b** in the arrays such that |a-b|> the value returned by our algorithm.
- Our algorithm uses both a min-heap and a max-heap to maintain lists of minimum and maximum values from each N array in the Arrays list. We know that these heaps store the smallest and largest values of the list passed to them at the root, which gets updated each time we delete or pop the root from the heap.
- We are additionally verifying that the elements we extract from either of the heaps do not originate from the same array by comparing their indices (2nd tuple value) to ensure they are from distinct arrays.
- Hence, our algorithm ultimately provides us with the maximum difference between the root values of the heaps, and these heaps ensure that we exclusively utilize the minimum and maximum values contained within them.
- Therefore, our initial assumption that there exists a and b such that a>root(max-heap) & b<root(min-heap) is **wrong.**
- By contradiction, we can prove that our algorithm always returns the max difference between the elements in a list of arrays such that the min and max values are from distinct arrays.

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 3→6→5→10→9. You are not required to prove the correctness of your algorithm. (10 points)

Solution:

- ➔ Algorithm to find the minimum number of flights required to fly from city a to city b is as follows:
  - ◆ Let us traverse from ***b*** to ***a*** backwards to identify the sequence of minimum number of flights required to fly from ***a*** to ***b***.
    - ○ First, we need to check if both a and b are equal. If yes, then we will need to return 0 as the output. ====> O(1)
    - ○ If *b* and N are equal and *b* is odd then there is no path between *a* and *b*. Therefore, we return -1 as the output. In the rest of the cases a sequence of flights from city *a* to *b* exist. ====> O(1)
    - ○ Now, let us initialize the variable *curr* to the value of *b*. This variable is used to track the current city at which the flight is in. ====> O(1)
    - ○ Let us initialize a variable *count* to 0. This is used to keep track of the number of flights required to reach the destination. ====> O(1)
    - ○ Do the following while(*curr*>*a*): ====> O(logN)
      - ■ If *curr* is even, *curr* = *curr*/2 ====> O(1)
      - ■ If *curr* is odd, *curr* = *curr*+1 ====> O(1)
      - ■ *count* = *count* + 1 ====> O(1)
    - ○ If the value of *curr* < *a*, we need to find a path from *a* to *curr*. Since there are direct flights from city i to city i-1, we will require further (*a* - *curr*) flights. Hence, our answer will be *count* + (*a* - *curr*).
    - ○ If the value of *curr* = *a*, we will return *count* as the final answer.
    - ○ Since (*a* - *curr*) is zero in our previous step we can generalize that the total number of flights required to reach from point a to point b will be given by *count* + (*a* - *curr*).

      So, the time complexity of this algorithm is **O(logN)**.