CS 600 A - Advanced Algorithms – Homework 4

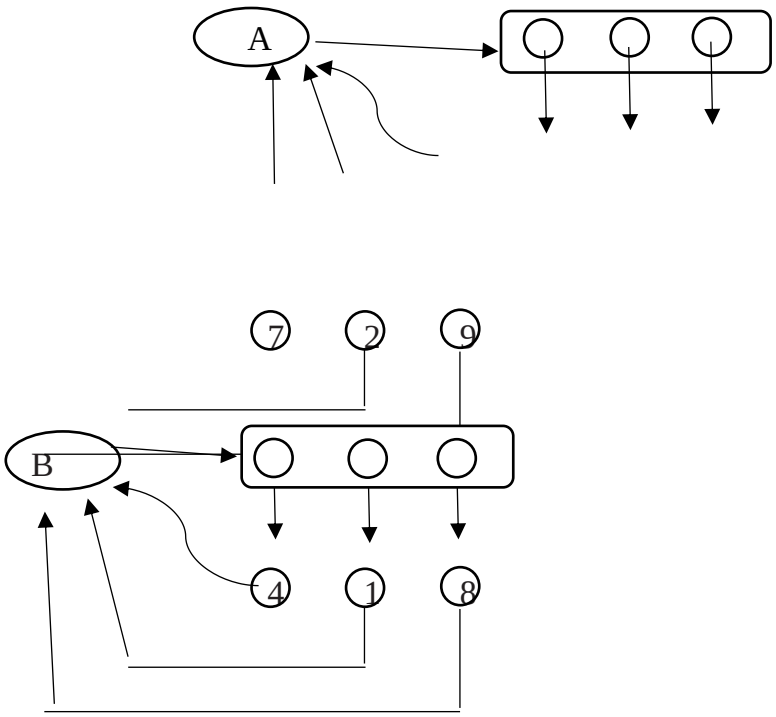
Name: Deepali Nagwade | 20013393

Question 7.5.5: One additional feature of the list-based implementation of a union-find structure is that it allows for the contents of any set in a partition to be listed in time proportional to the size of the set. Describe how this can be done.

Answer: Suppose this is a list-based implementation of the union find structure

$$A = \{7,2,9\} \text{ and } B = \{4,1,8\}$$

Here the set has object a with a head node and tail node. The head points to the first node and the tail points to the last node. Each set can be traversed starting from the header pointer.



Algorithm
Input is the object set, and the output would be all the nodes.

```
printSet(node)
while node.next != null
        print node
        node = node.next
print node.next
```

The loop will run as per the size of the set to list all the nodes of the set so, if the size is n then the loop will run n times. So, here time complexity is O(n)

Question 7.5.9 Describe how to implement a union-find structure using extendable arrays, which each contains the elements in a single set, instead of linked lists. Show how this solution can be used to process a sequence of m union-find operations on an initial collection of n singleton sets in O (n log n + m) time.

Answer: To implement a union-find (disjoint-set) data structure using extendable arrays where each array represents a single set of elements instead of linked lists, we can follow "union by rank" approach with path compression and has a time complexity of O (n log n + m) for processing a sequence of m union-find operations on an initial collection of n singleton sets.

Algorithm:

Step 1: Initialization

- Create an array parent of size n.
- Create an array rank of size n.
- Initialize each element i in parent to itself, i.e., parent[i] = i for 0 <= i < n.
- Initialize each element i in rank to 0, i.e., rank[i] = 0 for 0 <= i < n.

Step 2: Find Operation

- Define a function find(x) to find the representative of set containing element x.
- If parent[x] equals x, return x.
- Otherwise, recursively call find(parent[x]).
- Perform path compression by updating parent[x] to the result of find(parent[x]).
- Return parent[x].

Step 3: Union Operation

- Define a function union(x, y) to union the sets containing elements x and y.
- Find the representatives (roots) of the sets containing x and y using the find operation:
  - root_x = find(x)
  - root_y = find(y)
- If root_x equals root_y, no further action is needed (x and y are already in the same set).
- Otherwise, compare the ranks of root_x and root_y:
  - If rank[root_x] is greater than rank[root_y], set parent[root_y] = root_x.
  - If rank[root_x] is less than or equal to rank[root_y], set parent[root_x] = root_y.
  - If rank[root_x] equals rank[root_y], increment rank[root_y] by 1.

Step 4: Processing a Sequence of m Union-Find Operations

- 4.1. Initialize a Union-Find data structure with n singleton sets using Step 1.
- 4.2. Process the sequence of m union-find operations:
    - For each operation (x, y) where x & y are to be united call union(x, y).
- 4.3. The total time complexity for processing m union-find operations on n singleton sets is O(n log n + m).

Question 7.5.21: Consider the game of Hex, as in the previous exercise, but now with a twist. Suppose some number, k, of the cells in the game board are colored gold and if the set of stones that connect the two sides of a winning player's board are also connected to $k' \leq k$ of the gold cells, then that player gets k' bonus points. Describe an efficient way to detect when a player wins and, at that same moment, determine how many bonuses points they get. What is the running time of your method over the course of a game consisting of n moves?

Answer: To efficiently detect a winning player in Hex and calculate bonus points, you can use DFS and Union-Find:

Algorithm:

- Initialize the Hex board and Union-Find for tracking player components.
- Maintain arrays for each player's components reaching opposite sides and bonus points for gold cell connections.
- For each move, update player components using DFS and merge adjacent stones.
- After a move, check if a player's component connects both sides and if it's linked to gold cells.
- Continue until a player wins or the board is full.
- Determine the bonus points for the winning player based on gold cell connections.
- The worst-case time complexity for this approach is O(n * N), where n is the number of moves and N is the board size, but optimizations make it efficient for practical Hex boards.

Pseudocode:

Initialize the Hex board and data structures

- Initialize HexBoard and UnionFind data structures
- Initialize connectedToA and connectedToB arrays as False
- Initialize bonusPointsA and bonusPointsB arrays as 0

```
# Main game loop
while the game is ongoing:
    # Player makes a move by placing a stone
        MakeMove(player, position)
    # Update connected components using DFS and Union-Find
        UpdateComponents(player, position)
   # Check if player's component reaches opposite sides
        if player is A and DFSReachOppositeSide(player's components):
                Set connectedToA[player] to True
       if player is B and DFSReachOppositeSide(player's components):
                Set connectedToB[player] to True
   # Check if connected components are linked to gold cells
       UpdateBonusPoints(player, goldCells)
  # Check for a winning player
       if connectedToA[player]:
            Declare Player A as the winner with bonusPointsA
       End the game

      if connectedToB[player]:
           Declare Player B as the winner with bonusPointsB
       End the game

# End of the game
```

This pseudocode outlines the main steps of the algorithm for detecting a winning player in Hex and calculating bonus points. You would need to implement the specific functions like MakeMove, UpdateComponents, DFSReachOppositeSide, and UpdateBonusPoints according to your programming language and data structures.

Question 8.5.12 Suppose we are given a sequence S of n elements, each of which is colored red or blue. Assuming S is represented as an array, give an in-place method for ordering S so that all the blue elements are listed before all the red elements. Can you extend your approach to three colors?

Answer: To reorder an array of elements (colored red or blue) in-place so that all blue elements appear before all red elements, you can use a two-pointer approach. Here's a method to achieve this for two colors:

This code uses two pointers, left and right, to scan the array from both ends. The left pointer moves from left to right, looking for red elements, while the right pointer moves from right to left, looking for blue elements. When a red and blue element are found at the respective positions of left and right, they are swapped to place blue elements before red elements. This process continues until the pointers meet in the middle.

Pseudocode:

```
reorderRedAndBlue(arr):
    left <- 0  # Initialize the left pointer
    right <- length(arr) - 1  # Initialize the right pointer
    current <- 0  # Initialize the current po b inter


    while current <= right:
        if arr[current] == 'blue':
            # Swap blue element with the leftmost position (left pointer)
            swap(arr[current], arr[left])
            left <- left + 1
            current <- current + 1
        elif arr[current] == 'red':
            # Swap red element with the rightmost position (right pointer)
            swap(arr[current], arr[right])
            right <- right - 1
        else:
            # Elements of other colors remain in place
            current <- current + 1
```

To extend this approach to three colors, we can use a similar strategy with three pointers and additional conditionals:

```
Procedure reorderThreeColors(arr):
    leftBlue <- 0  # Initialize the leftmost position for blue elements
    rightRed <- length(arr) - 1  # Initialize the rightmost position for red elements
    current <- 0  # Initialize the current pointer

    while current <= rightRed:
        if arr[current] == 'blue':
            # Swap blue element with the leftmost position for blue elements
            swap(arr[current], arr[leftBlue])
            leftBlue <- leftBlue + 1
            current <- current + 1
        elif arr[current] == 'red':
            # Swap red element with the rightmost position for red elements
            swap(arr[current], arr[rightRed])
            rightRed <- rightRed - 1
        else:
            # Elements of the third color (e.g., white) remain in place
            current <- current + 1
```

Question 8.5.22: Suppose we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an O(n log n)-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.

Answer: We can determine the winner of the election represented by the sequence S in O(n log n) time using a divide-and-conquer approach combined with sorting. Here's an algorithm to achieve that:

- Divide and Conquer: Divide the sequence S into two equal halves.
- Recursion: Recursively find the winner of each half.
- Count Votes: Count the number of occurrences of the winners in both halves.
- Determine Global Winner: Compare the vote counts of the two winners from the halves and choose the candidate with the most votes as the global winner.
- Return Winner: Return the global winner as the winner of the entire election.

Pseudocode:

```
Function findWinner(S):
    # Base case: If there is only one vote, that candidate wins
    If Length(S) == 1:
        Return S[0]
    # Divide the sequence into two equal halves
    mid = Length(S) // 2
    leftHalf = S[0:mid]
    rightHalf = S[mid:]
    # Recursively find the winners of the two halves
    winnerLeft = findWinner(leftHalf)
    winnerRight = findWinner(rightHalf)
    # Count the votes for the winners in both halves
    countLeft = CountOccurrences(leftHalf, winnerLeft)
    countRight = CountOccurrences(rightHalf, winnerRight)

    # Determine the global winner
    If countLeft > countRight:
        Return winnerLeft
    Else:
        Return winnerRight
```

In the pseudocode above:

- findWinner(S) is the main function to find the election winner.
- The base case checks if there is only one vote, and if so, returns that candidate as the winner.
- The sequence S is divided into two equal halves, leftHalf and rightHalf.
- The algorithm recursively finds the winners of the two halves.
- CountOccurrences(sequence, candidate) is a function that counts the number of occurrences of a candidate in a given sequence.
- Finally, the algorithm compares the vote counts of the two winners from the halves and returns the global winner.

Question 8.5.12: Consider the voting problem from the previous exercise, but now suppose that we know the number $k < n$ of candidates running. Describe an $O(n \log k)$-time algorithm for determining who wins the election.

Answer: If we know that there are k candidates running in the election, we can determine the winner in $O(n \log k)$ time using a modified approach. Instead of a divide-and-conquer strategy, we can use a priority queue (min-heap) to efficiently keep track of the k candidates with the highest vote counts. Here's the algorithm:

- Initialize a dictionary or hash map to store the vote counts for each candidate.
- Initialize a min-heap (priority queue) with the k candidates with the highest vote counts. Initially, the heap is empty.
- Iterate through the sequence S and update the vote counts for each candidate in the dictionary.
- As you update the vote counts, maintain the min-heap with the k candidates who have the highest vote counts. If the heap size is less than k, insert the candidate into the heap. If the heap size becomes $k + 1$, remove the candidate with the lowest vote count from the heap.
- Continue iterating through sequence S until you have counted all the votes.
- After counting all the votes, the candidates in the min-heap are the k candidates with the highest vote counts. Determine the candidate with the most votes among these k candidates, and that candidate is the winner of the election.

```
Function findWinnerWithKCandidates(S, k):
    Initialize a dictionary voteCounts
    Initialize a min-heap (priority queue) topK with a maximum size of k

    For each vote in S:
        If vote is in voteCounts:
            Increment voteCounts[vote] by 1
        Else:
            Add vote to voteCounts with a count of 1

        If size of topK < k:
            Insert vote into topK with voteCounts[vote] as the priority
        Else if voteCounts[vote] > smallest priority in topK:
            Remove the candidate with the smallest priority from topK
            Insert vote into topK with voteCounts[vote] as the priority

    Determine the winner from the candidates in topK with the highest vote counts

    Return the winner
```

This algorithm maintains a heap of the k candidates with the highest vote counts throughout the iteration, ensuring that it only needs to consider the top candidates and not the entire list of candidates. As a result, the time complexity is O(n log k), where n is the number of votes and k is the number of candidates running.

Question 9.5.17: Suppose you are given two sorted lists, A and B , of n elements each, all of which are distinct. Describe a method that runs in O (log n) time for finding the median in the set defined by the union of A and B .

Answer: We can find the median of the union of two sorted lists, A and B, of distinct elements in O(log n) time using a binary search approach. Here's a step-by-step explanation of the algorithm:

- Find the smaller of the two lists, let's say A, and the larger one, B.
- Initialize two pointers, low and high, for the smaller list A. Set low to 0 and high to the length of A.
- While low is less than or equal to high, perform the following steps:
  - Calculate the middle index of A as midA = (low + high) // 2.
  - Calculate the corresponding element in list B, midB = (len(A) + len(B) + 1) // 2 - midA.
  - Check if A[midA] <= B[midB] and B[midB - 1] <= A[midA]. If these conditions are met, it means we have found the correct partition of elements between A and B.
  - If the conditions are not met, adjust the low and high pointers accordingly:
  - If A[midA] > B[midB], set high to midA - 1.
  - If B[midB - 1] > A[midA], set low to midA + 1.
- Once the correct partition is found, calculate the median:
  - If the total number of elements in the union of A and B is odd, the median is the maximum of A[midA - 1] and B[midB - 1].
  - If the total number of elements is even, the median is the average of max(A[midA - 1], B[midB - 1]) and min(A[midA], B[midB]).

This algorithm efficiently partitions the elements between the two lists using binary search and calculates the median in O(log n) time complexity.

Question 9.5.24: Suppose University High School (UHS) is electing its student-body president. Suppose further that everyone at UHS is a candidate and voters write down the student number of the person they are voting for, rather than checking a box. Let A be an array containing n such votes, that is, student numbers for candidates receiving votes, listed in no order. Your job is to determine if one of the candidates got a majority of the votes, that is, more than n/2 votes. Describe an O(n)-time algorithm for determining if there is a student number that appears more than n/2 times in A. Hint: Use quick select to find median.

Answer: We can use the Quick Select algorithm to find the median element of the array A in O(n) time and then check if that element appears more than n/2 times to determine if there is a majority candidate. Here's the algorithm:

- Implement the Quick Select algorithm to find the median element of the array A. The Quick Select algorithm is similar to QuickSort but stops once the pivot element is in its sorted position.
- After applying Quick Select, you'll have the median element (let's call it "median_candidate").
- Initialize a counter variable "count" to 0.
- Iterate through the entire array A. For each element:
- If it matches the "median_candidate," increment the "count" by 1.
- If "count" becomes greater than n/2, return "median_candidate" as the majority candidate.
- If you reach the end of the array and "count" is less than or equal to n/2, there is no majority candidate, and you can return "No majority."

Pseudocode:

```
Function findMajorityCandidate(A):
  median_candidate <- QuickSelect(A, 0, n-1, n/2)  # Find the median using Quick Select
  count <- 0

  For each vote in A:
    If vote == median_candidate:
      count <- count + 1

  If count > n/2:
    Return median_candidate  # The majority candidate
  Else:
    Return "No majority"   # No majority candidate found
```

- QuickSelect is a function that returns the kth smallest element in the array A. In this case, we find the median (k = n/2) efficiently.
- We then count the occurrences of the median_candidate in the array A.
- If the count is greater than n/2, we return the median_candidate as the majority candidate. Otherwise, we report that there is no majority candidate.
- The key insight is that the majority candidate (if it exists) will always be the median in the sorted array, and Quick Select helps us find it in linear time.