CS 600 Advanced Algorithms

Sri Dhanush Reddy Kondapalli

CWID: 10476150

Homework 4

1 R-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.

A union-find structure's list-based implementation is essentially a collection of linked lists. Each set is given its own list. Each of these lists has a head node, and every other node in the linked list keeps a pointer back to it. This means that finding a node in the list is incredibly efficient, with the process taking O(1) time. For a set of size n, repeat this n times, and the total duration is directly proportional to the size of the set. This makes sense because the entire point of list-based union-finds is that they are incredibly efficient at all operations other than union.

2 C-7.5.9 Implement a union-find structure using extendable arrays, which each contains the elements in a single set, instead of linked lists.

Algorithm:

```
makeSet():
    Extendable Array S:
    For each singleton element in S, i do:
        Add i to S
Find(i):
    Return S which contains i
Union(p, q):
```

```
If set p < q, then
    For each element i in the set p, do:
        Remove i from p and add to q
Else, then
    For each element i in the set q, do:
        Remove i from q and add to p
```

We know that in an extensible array, all add and delete operations require O(n) time.
The search procedure takes O(n) time. The union find procedure takes O(nlog n + m) time.
We don't need a head node as a pointer here; we can traverse the array by indexing.

3 A-7.5.21 Describe an efficient way to detect when a player wins and also, at that same moment, determine how many bonus points they get. What is the running time of your method over the course of a game consisting of n moves?

In this version of the game, the gold cells are represented by a set called S_G. At the start of the match, four sets are established, each of which is constant and comprises cells from a given boundary of the board that indicate win criteria (e.g., S_{Right}).
When a piece is put at a cell during a move, a new set at that cell is created that initially just contains the piece used in that move. If that cell happens to be gold, it is additionally added to that player's gold set. Following that, other cells around the active one are inspected to determine if they contain fragments of the same hue. If this is the case, the newly produced individual set is united with the existing set of the same-color union. Use **findSet()** after each union to see if the union set has a member from **both** of that player's victory sets (e.g., S_{Right} and S_{Left} for white). If this is the case, that player has won, and the winning set has been found. **findSet()** is also used to retrieve the amount of gold cells in the winning set in order to compute the bonus points.
In terms of execution time, for n movements, there are a total of n **union()** and 2n **findSet()** operations (1 and 2 per move, respectively). The total number of bonus points a player has at the conclusion of the round is equal to the number of times **find()** must be called to add these points to their score. Because of the distribution of function calls, which favors **find()** over **union()**, I chose a list-based implementation of union-find. The run speeds of this implementation, according to Section 7.3 of the textbook, favor find operations. With this in mind, the total cost for n movements is O(nlogn), which can be calculated by doing 1 **union()** (of time O(log n)) and 2 **find()** (of time O(1)) n times. The

gold bonus computation is O(1) since it is simply a series of n **find()** operations. As a result, this method runs in $O(n \log n + n)$ time.

4 C-8.5.12 Suppose we are given a sequence S of 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?

This issue is made easier to solve by modifying the in-place quick-sort technique described in Section 8.3 of the textbook. The two colors in this case, blue and red, will be assigned numerical values so that val(blue) < val(red). This may be performed using a variety of approaches, such as static variables or helper methods, as long as the underlying property of blue's evaluation being smaller than red's holds. The process will then proceed as typical rapid sort does. Instead of explicitly examining the content of each sequence index, the result of that sequence's valuation function (i.e., val(S[i]) is employed. This allows the less valuable blue pieces to be placed ahead of their more valuable red counterparts.
Using the same approach, this technique may be modified to work with three colors. The value calculation algorithm will be changed to reflect the new color's inclusion. Blue will continue to be the smallest, with red and color x being valued in the order specified. The updated quick sort method will proceed as described above, with the series eventually being arranged so that all blue items are first, followed by all second color elements, and finally all third color elements.

5 A-8.5.22 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.

- To begin, split the sequence S into two sections, Left and Right. Compare their two candidates to see if they have a majority. Because they are integers, this is simple.
  - If x is majority from both Left and Right, then it is the majority of Left + Right

  - If x is majority from Left, whereas y is majority from Right, we calculate the votes of x and y. It is a tie if they both have the same number of votes.

- Algorithm:

```
Input: sequence S, index value i, j
Ouput: Winner of our subsequence S[i:j]
winnerOfMajority(S, i, j):
    If i == j, then:
        Return S[i]
    X = winnerOfMajority(S, i, (i + j)/ 2)
    y = winnerOfMajority(S, (i + j)/ 2 + 1, j)
    if x == y, then:
        return x
    else if x.voteCount > y.voteCount, then:
        return x
    else if x.voteCount < y.voteCount, then:
        return y
    else, then:
        return 'tied'
```

The winner is determined by this method in time O(n log n).
This will be executed first, assuming the sequence indexing is from 1-n:
winnerOfMajority(S, i, n)

6 A-8.5.23 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.

2 approaches:
Algorithm:

```
winnerOfMajority(S, i, j):
    If i == j, then:
        Return S[i]
    X = winnerOfMajority(S, i, (i + j)/ 2)
    y = winnerOfMajority(S, (i + j)/ 2 + 1, j)
    if x == y, then:
        return x
    else if x.voteCount > y.voteCount, then:
        return x
    else if x.voteCount < y.voteCount, then:
        return y
    else, then:
        return 'tied'
```

This algorithm takes time O(n log k) to decide the winner. This will execute first: winnerOfMajority(S, i, k)
2. Algorithm:

- Put our candidates' IDs in a hashset - this is for n unique candidates, and the hashing will be with no to little collision.
- Assign each candidate a number of votes and enter them into a lookup database.
  - Candidate.put(c, c.votes) – to insert the candidate/ candidate votes in the hashset
  - Candidate.get(c) – to get the number of votes for our candidate c
- It takes O(1) time to count votes for each contender. This will take O(n) time for n candidates in a row.

7 C-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.

To begin, we determine the median of our two sorted arrays by continually finding the means of the two arrays and comparing and finding the median again by dividing them into subarrays.
Algorithm:

```
1.
FindingMedian(A, B, n):
    Input: 2 sorted arrays, A and B, consisting of n elements
    Output: Median of union of A and B
    If n == 1, then
        Return (A[0] + B[0])/2
    Else if n == 2, then
        Return {max(A[0], B[0]) + max(A[1], B[1])} / 2
    Else:
        Median1 = median(A, n)
        Median2 = median(B, n)
        If Median1 > Median2, then:
            Return FindingMedian(A[0: n/2], B[n/2:n], n/2)
        Else
            Return FindingMedian(A[ n/2:n], B[0:n/2], n/2)
2.
Input: Above Sorted array A
Output: Median of A
Median(A, n):
    If n is even, then:
        Return (A[n/2] + A[n/2 -1])/2
    Else:
        Return (A[n/2])
```

The time cost of the above algorithm is O(log n):
T(n) = T(n/2) + O(1)
n/2 → because we are halving the size of our n array every time
1 → because we solve one problem at time

8 A-9.5.24 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.

This, like the previous election problem, can be addressed effectively using a map. Because everyone at UHS is a candidate, there might be a maximum of n candidates. Because A contains the student numbers of those voting, it is reasonable to suppose that no two students have the same number, hence eliminating the potential of collisions. Based on this data, the map may be the size of the maximum student ID number (in which case key indexing will utilize each student's ID number to access their map value) or use a hash function to translate the student's number to their map index. The ultimate outcome is the same regardless of how it is done. The key scheme of the map is the student ID numbers (or hashed equivalents), and the value is the number of votes cast for that number. The array A is iterated over, and at each index, the key is either placed into the map with a value of 1 or, if it already exists, a value of 1 is added to its value. After each insertion/update, the value of that key is checked to determine if it is more than n/2. If this occurs, the process is terminated and the winner is proclaimed. This approach takes O(n) time in total since the map operations employed here run in O(1) time and the array is iterated through in O(n) time.