COMP 3804 Assignment 2

February 28, 2021

Question 1.

Name: Braeden Hall
Student Number: 101143403

Question 2.

1. A majority element is defined as an element that occurs $> \frac{n}{2}$ times in some sequence S. Since the size of S is equal to n there cannot be more than 1 element of S that occurs more than $\frac{n}{2}$ times. Therefore, the number of majority elements must be ≤ 1 and ≥ 0.

Algorithm 1 majorElemSlow(S, n):

2.
```
if n = 1 then
    return S[0]
end if

A ← MergeSort(S, n)
count ← 0
curNum ← A[0]
t ← n/2

for i = 0 to n do
    if A[i] = curNum then
        count ← count + 1
        if count > t then
            return curNum
        end if
    else
        curNum ← A[i]
        count ← 1
    end if
end for

return there is no majority element
```

Correctness:
The algorithm starts by sorting the elements from least to greatest. The algorithm then traverses the sorted list one element at a time. Since the list is sorted all elements of equal value will be grouped together. The algorithm keeps track of how many times its sees each element.

If the current element is equal to the previous element it increments a counter, otherwise it resets the counter to 1. If the counter variable ever reaches a value greater than $\frac{n}{2}$ then the algorithm returns the current element that it was counting. The counter reaching a value greater than $\frac{n}{2}$ means that the current element appeared more than $\frac{n}{2}$ times in the list and it is therefore the majority element.

If the for loop terminates then the algorithm returns that there was no majority element. This must be true because if the value of the counter had ever reached more than $\frac{n}{2}$ the algorithm would have returned before the loop finished.

Run Time:
The MergeSort step at the beginning of the algorithm takes $O(n \log n)$ time, as we have seen in class. The for loop then looks at each element in the sorted list at most 1 time; this is a linear iteration, so it takes $O(n)time$. The rest of the statements, both in and outside of the for loop take constant time. Therefore, the running time of the algorithm is $O(n \log n)$.

Algorithm 2 majorElemFast(S, n):

3.
```
t ← n/2
mid ← Select(S, t)
count ← 0

for i = 0 to n do
    if S[i] = mid then
        count ← count + 1
        if count > t then
            return mid
        end if
    end if
end for

return there is no majority element
```

Correctness:
The algorithm starts by running 'Select' to find the $\frac{n}{2}^{th}$ smallest element in the sequence. A majority must appear at least $\frac{n}{2}$ times in the sequence. So even if the majority element is the smallest or the largest element in the sequence, it must still appear in the middle of the sorted list (or as the middle 2 elements if n is even). This means that if a sequence has a majority element it will be the $\frac{n}{2}^{th}$ smallest element.

Once the middle element is found, the algorithm traverses the sequence and counts each time it sees that number. If the number appears more than $\frac{n}{2}$ times then the algorithm returns the middle element, otherwise it returns that there is no majority element.

Run Time:
As we have seen in class, there is a version of the Select algorithm that runs in $O(n)$ time. The for loop in the algorithm iterates over each element at most once which also takes $O(n)$ time. The rest of the statements, both in and outside of the for loop take constant time. Therefore, the running time of the algorithm is $O(n)$.

Question 3.

Big-O notation is used to state the upper bound of some algorithm's run time, in other words, the algorithm runs in time less than or equal to some constant times its Big-O. Therefore, stating that the running time of an algorithm is at least its Big-O is equivalent to stating that the running time is greater than or equal to its worst case run time. This does not makes since Big-O notation implies that the run time is less than or equal to what is inside the brackets. The statement "at least" would be correct when referring to an algorithm's Big-Omega notation as this represents the lower bound of its run time.

Question 4.

Assumptions:

1. Array A contains the k lists in its indices
2. Each list is stored as a linked list
3. Each node in a linked list stores the node's data and a pointer to the next element in that list
4. The heap orders elements and does operations based on the data stored in the node
5. None of the k lists are empty to start with
6. Getting the first element of a linked list takes $O(1)$ time
7. Appending to the end of an array takes $O(1)$ time

General Idea:
Each of the k lists in sorted least to greatest, so the smallest of all the elements is the first element of one of these lists. I use a min-heap to find this smallest element. Once the algorithm finds and extracts the minimum element in the heap it inserts back the next element from its respective list (if it has one). This process is repeated until we get a fully sorted sequence of n numbers.

Algorithm 3 Merge(A, k):

```
H ← empty min-heap
sorted ← []

for i = 0 to k do
    H.insert(A[i].getFirst())
end for

while n > 0 do
    r ← H.extractmin()
    if r.next! = NULL then
        H.insert(r.next)
    end if
    sorted.append(r.data)
    n ← n − 1
end while

return sorted
```

Run Time:
The first for loop iterates k times and each time it inserts into with $O(k)$ elements. As seen in class, the insert operation in a heap takes time proportional to the logarithm of the size of the heap, therefore, the for loop runs in $O(k \log k)$ time.

The while loop runs for n iterations and each time it extracts something from the heap and then inserts something into the heap. Both of these operations take time proportional to the logarithm of the size of the heap; the heap always has $O(k)$ elements. The remaining operations in the while loop take constant time. Therefore, the while loop runs it $O(n \log k)$. Since the longest operation done in this algorithm takes $O(n \log k)$, therefore, the running time of the algorithm is $O(n \log k)$.

Question 5.

Assumptions:

1. The heap H stores tuples
2. The first index of the tuple is the data to use for operations in H
3. The second index of the tuple is the index of the data in the heap A
4. Appending to the end of an array takes $O(1)$ time

Algorithm 4 smallestK(A, k):

```
H ← empty min-heap
sorted ← []
m ← 0
H.insert((A[1], 1))

while m < k do
    r ← H.extractmin()
    sorted.append(r[0])
    i1 ← 2 · r[1]
    i2 ← (2 · r[1]) + 1

    if i1 ≤ n then
        H.insert((A[i1], i1))
        if i2 ≤ n then
            H.insert((A[i2], i2))
        end if
    end if
    m ← m + 1
end while

return sorted
```

Correctness:
As stated in the question, the root of a subtree in a min-heap contains the minimum element in that subtree. Therefore, the root of A is the smallest element in the heap. The second smallest element is one of the children of the root. The third smallest is either the roots other child or one of the children of the second smallest element, and so on and so forth.

This algorithm maintains both a list of the smallest elements encountered so far (i.e. the m smallest elements in the heap A where $0 < m <= k$) and a helper min-heap that contains the children of these m elements. As I previously described, one of these children must be the $(m + 1)^{th}$ smallest element, this child will be located at the root of the helper heap.

While $m < k$ the algorithm extracts the smallest element currently stored in the helper heap and adds it to the sorted list, then it adds that elements children in the heap A into the helper heap (if they exist). This ensures that all elements that could possibly be the next smallest element are considered in the search. Once $m = k$ the algorithm will have found the k smallest elements in the heap A and it will terminate by returning the sorted list.

Run Time:
The first heap operation inserts an element into an empty heap, this should take constant time. The while loop will then iterate a total of k times. Each time it iterates it removes 1 element from the heap and inserts at most 2 elements. Both of these operations take time proportional to the size of the heap. On iteration i of the loop, the helper heap will have $O(i)$ elements since the maximum number of elements it can have is $i+1$. Therefore, on the k^{th} iteration the helper heap will have $O(k)$ elements, so the while loop completes in $O(k \log k)$ time. Therefore, the running time for the algorithm is $O(k \log k)$.

Question 6.

Assumptions:

1. All vertices have a distinct label in the set $\{0, 1, \ldots, |V|-1\}$ that identifies which vertex it is
2. The label of an arbitrary vertex u is stored as a property of the vertex and can be accessed by: $u.label$ in constant time
3. The array A is a list of pointers to adjacency lists, where $A[i]$ points to vertex v_i's adjacency list
4. Appending to the end of an array takes $O(1)$ time

Algorithm 5 twodegree(A):

```
1:  degrees ← []
2:  k ← 0

3:  for each element of A do
4:      degrees.append(0)
5:      for each vertex in A[k] do
6:          degrees[k] ← degrees[k] + 1
7:      end for
8:      k ← k + 1
9:  end for

10: twodeg ← []
11: for i = 0 to k do
12:     twodeg.append(0)
13:     for each vertex u in A[i] do
14:         twodeg[i] ← twodeg[i] + degrees[u.label]
15:     end for
16: end for

17: return twodeg
```

Note: For each $i \in \{0, 1, \ldots |V|-1\}$: the twodeg array that is returned at the end of the algorithm contains the twodegree of vertex v_i at index i in the array.

Run Time:
The for loop beginning at line 3 iterates for each element inside the array A, this is equal $|V|$ (the size of the graph's vertex set). Inside of that there is another for loop that iterates for every vertex that appears in vertex v_i's adjacency list ($A[i]$). Since a vertex in an adjacency list indicates and edge in the graph, in total this for loop will iterate $2|E|$ times (an edge will appear in the adjacency list for both vertices). Therefore, the total running time of the for loop beginning on line 3 is $O(|V| + |E|)$.

Similarly, the for loop that begins on line 11 iterates k times, k is calculated to be equal to $|V|$. The for loop inside of it iterates for every vertex that appears in the adjacency list stored at $A[i]$, so a total of $2|E|$ times. This means that the for loop on line 11 finishes in $O(|V| + |E|)$ time. And since the other operations in the algorithm all take constant time: the algorithm runs in $O(|V| + |E|)$ time.

Question 7.

Note: to conserve space the algorithm in this question appears after the description.

Assumptions:

1. Since the graph is undirected, an edge (u, v) will be represented as vertex v appearing in u's adjacency list and vertex u will not appear in v's
2. All vertices have a distinct label in the set $\{0, 1, \ldots, |V| - 1\}$ that identifies which vertex it is
3. The label of an arbitrary vertex u is stored as a property of the vertex and can be accessed by: $u.label$ in constant time
4. The array A is a list of pointers to adjacency lists, where $A[i]$ points to vertex v_i's adjacency list
5. Getting and removing the first element of a linked list both take $O(1)$ time
6. A linked list is doubly linked and stores a pointer the the tail element, so adding to the back takes $O(1)$ time
7. Appending to the end of an array takes $O(1)$ time

Run Time:
The for loop on line 3 iterates once for each element of A, so $|V|$ times. It also sets the value $n = |V|$.

The for loop on 7 iterates $|V|$ times, the loop inside of it iterates for every vertex in the adjacency list for vertex $A[i]$. Since vertices in an adjacency list represent edges in the graph and the outer for loop iterates for every vertex in the graph, the inner for loop iterates a total of $|E|$ times. This means that the outer for loop takes $O(|V| + |E|)$ time to finish.

After this for loop has finished the indegree of every vertex in the graph has been calculated. The indegree of vertex v_i will be stored at index i of the *indeg* array.

The for loop on line 14 does a linear search through the *indeg* array, which is the same size as the array of vertices; so this takes $O(|V|)$ time. This loop fills the original $noIn$ list will all the vertices that have an indegree of 0.

The while loop on line 19 iterates $|V|$ times, this is where a vertex v of indegree 0 is chosen and its number is set. The inner for loop goes through every vertex in the v's adjacency list, decrements its indegree by 1 (since we will be removing v from the graph and it will no longer point to anything) and decides whether or not it should be added to the list of vertices with indegree 0. In total, the inner for loop will iterate $|E|$ times, once for each edge. This means that the while loop finishes in $O(|V| + |E|)$ time.

Since both initially constructing the indegree array and assigning a number to each vertex takes $O(|V| + |E|)$ time, therefore the running time of the algorithm is $O(|V| + |E|)$.

Algorithm 6 topSort(A):

```
1: indeg ← []
2: n ← 0

3: for each element of A do
4:     indeg.append(0)
5:     n ← n + 1
6: end for

7: for i = 0 to n do
8:     for each vertex u in A[i] do
9:         indeg[u.label] ← indeg[u.label] + 1
10:    end for
11: end for

12: k ← 1
13: noIn ← empty linked list
14: for i = 0 to n do
15:    if indeg[i] = 0 then
16:        noIn.addLast((vertex v_i, i))
17:    end if
18: end for

19: while n>0 do
20:    v ← noIn.getFirst()
21:    v[0].number ← k
22:    k ← k + 1

23:    for each vertex u in A[v[1]] do
24:        indeg[u.label] ← indeg[u.label] − 1
25:        if indeg[u.label] = 0 then
26:            noIn.addLast((vertex v_{u.label}, u.label))
27:        end if
28:    end for

29:    remove vertex v[0] from graph          ▷ Please see note below algorithm
30:    n ← n − 1
31:    noIn.removeFirst()
32: end while
```

Note: Lines 29 of the algorithm is there to conform to the "Remove v from the graph" step in the algorithm given in the question. This is a general step and I'm not exactly sure what it implies. If it causes confusion please ignore it as it is not necessary for my algorithm to produce the correct output.