CSCI803 Assignment

Yao Xiao
SID 2019180015

October 11, 2020

1 Problem 1

1.1 Question A

We can choose counting sort which runs in O(n) time and it is a stable sorting algorithm.

1.2 Question B

We can execute Quicksort-Partition one pass around around the pivot (x=0). If it is a Lomuto partition, it will place all 0 elements on the left and all 1 elements on the right. This will also sort the array. After that, it is in place and has a O(n) running time.

Here is the pseudocode:

```
1 i = 0
2 for j = 1 to n
3 if A[j] <= 0
4 then i = i + 1
5 swap(A[i], A[j])
```

1.3 Question C

We can choose insertion sort which is an in place sorting algorithm and is also stable. We can consider the situation ($A[i] = A[j]$ and $i < j$). Since i < j, the priority $A[i]$ will be added to the sorted array $A[1 \cdots i-1]$ by moving in the correct position. This will lead the sorted array $A[1 \cdots i]$ containing the original $A[i]$ at some position $k \leq i$. Now $A[i]$ is $A[k]$, when considering $A[j]$, $A[j]$ must be moved down to $A[1 \cdots j-1]$, where $A[1 \cdots i]$ is a sub-array containing $A[k]$. $A[j]$ cannot bypass $A[k]$ in the shifting process because $A[k] = A[j]$. Therefore, the original $A[i]$ and the original $A[j]$ will preserve their relative order.

1.4 Question D

Question A - Counting sort. Counting sort runs in O(n) times, and for b-bit keys with each bit value varies from 0 to 1, the sort time can be O(b(n + 2)) = O(bn).

1.5 Question E

Here is the modified counting sort:

```
initial C[0],···,C[k] a new array.
for i = 1 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
for i = 2 to k
    C[i] = C[i] + C[i − 1]
insert sentinel element at the start of A
B = C[0...k − 1]
insert number 1 at the start of B

for i = 2 to A.length
while C[A[i]] != B[A[i]]
    key = A[i]
    swap(A[C[A[i]]], A[i])
    while A[C[key]] == key
        C[key] = C[key] − 1
remove the sentinel element
return A
```

The storage space is O(k), but the algorithm is not stable.

2 Problem 2

2.1 Question A

Suppose x_k is the median of $x_1, \cdots, x_n$, x_k should be larger than $\lfloor \frac{1+n}{2} \rfloor - 1$ elements. According to the definition, the sum of the weights of elements less than x_k is:

$$\begin{aligned} \sum_{x_i < x_k} w_i &= \frac{1}{n} \cdot (\lfloor \frac{1+n}{2} \rfloor - 1) \\ &= \frac{1}{n} \cdot \lfloor \frac{n-1}{2} \rfloor \\ &\leq \frac{n-1}{2n} \\ &< \frac{n}{2n} \\ &< \frac{1}{2} \end{aligned} \tag{1}$$

And x_k is smaller than exactly $n - \lfloor \frac{1+n}{2} \rfloor$ elements:

$$\begin{aligned}\sum_{x_i > x_k} w_i &= \frac{1}{n} \cdot (n - \lfloor \frac{1+n}{2} \rfloor) \\ &= 1 - \frac{1}{n} \cdot \lfloor \frac{n+1}{2} \rfloor \\ &\leq 1 - (\frac{1}{n})(\frac{n}{2}) \\ &\leq 1 - \frac{1}{2} \\ &\leq \frac{1}{2}\end{aligned} \tag{2}$$

Based on the above derivation, x_k is also the weighted median.

2.2 Question B

Here is the pseudocode **weight-median**:

```
k = 1
# total weight of all x_i < x_k
s = 0
while s + w_k < 1 / 2
      do s = s + w_k
         k = k + 1
return x_k
```

The loop invariant of this algorithm is that s is the sum of the weights of all elements less than x_k:

$$s = \sum_{x_i < x_k} w_i \tag{3}$$

And because the list is sorted, for all $i < k$, $x_i < x_k$. By induction, s is correct because in every iteration through the loop s increases by the weight of the next element.

The loop is guaranteed to terminate because the sum of the weights of all elements is 1. We can show that when the loop terminates x_k is the weighted median.

Suppose s' is the value of s at the start of the next to last iteration of the loop , and since the next to last iteration did not match the termination condition:

$$\begin{aligned}s &= s' + w_{k-1} \\ s' + w_{k-1} &< \frac{1}{2} \\ \sum_{x_i < x_k} w_i = s &< \frac{1}{2}\end{aligned} \tag{4}$$

The sum of the weights of elements greater than x_k is:

$$\sum_{x_i > x_k} w_i = 1 - (\sum_{x_i < x_k} w_i) - w_k = 1 - s - w_k \tag{5}$$

By the loop termination condition:

$$\begin{aligned} \sum_{x_i<x_k} w_i = s &\geq \frac{1}{2} - w_k \\ -s &\leq -\frac{1}{2} + w_k \\ 1 - s - w_i &\leq \frac{1}{2} \\ \sum_{x_i>x_k} w_i &\leq \frac{1}{2} \end{aligned} \tag{6}$$

From equation 4 and 6, we can conduct that x_k is the weighted median.

By analyzing the running time of the algorithm, the time required to sort the array plus the time required to find the median.

Sorting the array can be $O(nlgn)$ time (using heapsort or others) and the loop in **weight-median** takes $O(n)$ time. The total running time in the worst case, therefore, is $O(nlgn)$.

2.3 Question C

The basic strategy of this algorithm is similar to a binary search, A is an array containing the median of the initial input, and l is the total weight of the initial input elements less than A.

Here is the pseudocode **linear-weight-median(A,l)**:

```
n = len(A)
m = median(A)
# B = {A[i] < m}
B = NULL
# C = {A[i] >= m}
C = NULL
# total weight of B
w_B = 0
if len(A) == 1
    return A[1]

for i in range(1,n)
    if A[i] < m
        w_B = w_B + w_i
        B.append(A[i])
    else
        C.append(A[i])

if l + w_B > 1/2
    linear-weight-median(B,l)
else
    linear-weight-median(C,w_B)
```

In order to prove that the algorithm is correct, we prove that for each recursive call, the weighted median y of the initial A always exists in the recursive call of A, and l is the element whose total weight of all elements x_i is less than all A.

First let us consider the case where $l + w_B > \frac{1}{2}$. Since y must be in A, y must be in B or C in line 19. Since the total weight of all elements is less than any element in C, by definition, the weighted median cannot be in C, so it must be in B. In addition, we did not discard any elements less than any element in B, so l is correct and the precondition is met.

And if there is a situation where $l + w_B \leq \frac{1}{2}$, then y must be in C. All elements of C are greater than all elements of B. Therefore, the total weight of elements less than C is $l + w_B$, and the condition is that the recursive call is also satisfied. Therefore, by induction, the premise is always correct.

Each time the recursive call is made, the size of A will decrease, and the algorithm will eventually terminate. Since the weighted median is always in A, when there is only one element left, it must be the weighted median.

It takes $O(n)$ time to calculate the median and divide A into B and C. Each recursive call **linear-weight-median** makes the size of the array from n to $\lceil \frac{n}{2} \rceil$, so the recursion is $T(n) = T(\frac{n}{2}) + O(n) = O(n)$.