CS 5800 Summer 2022 Problem Set #2
Farbod Alinezhad (Thursday 26th May, 2022)

Contents

1 Dasgupta Question 2.14

To find duplicates in $O(n \log n)$ time, we can do a merge-sort, which take $O(n \log n)$ and then look through the sorted array one more time, which takes $O(n)$ time, to see if any of the neighbors are equal. This would finish in $n \log n + n$ time which is $O(n \log n)$. A pseudocode of this would be:

```
b=sort(b)
i=1
while (i < (length(b)-1)) {
    if (b[i]==b[(i+1)]){
        b=(b[-(i+1)])
    }
    else{
        i=i+1
    }
}

```

It should be noted that the sort should use merge-sort algorithm. The pseudocode uses R language syntax.
If we want the whole process to happen as a divide and conquer algorithm, we should simply integrate the duplicate removal into the merge-sort algorithm. Based on the book's pseudocode, the "function mergesort" stays the same, but the "function merge" should be changed as follows:

```
function merge(x[1...k],y[1...l])
if k=0: return y[1...l]
if l=0: return x[1...k]
if x[1]<y[1]:
    return x[1] o merge (x[2...k],y[1...l])
if x[1]==y[1]:
    return merge (x[2...k],y[1...l])
else return y[1] o merge (x[1...k],y[2...l])

```

Basically, the idea is to skip when we find equals, instead of concatenating with the x[1] or y[1].

2 Dasgupta Question 2.15

Here, we first find a number v randomly, to separate the data into three sections (smaller, equal, and bigger than v). The book's does this by creating three new lists and allocating new memory. However, we can go through the data two times, and create a new data that has the sublists within and know where they start and finish. To do this:

```
        #First we find the v
        v=sample(s,1),

        count_equal=0
        count_higher=0

    then we find every s that is equal to v and move it to the end of the
   s and also count such occurences,
        for (i in 1:length(s)){
            if (s[i]==v){
                remove (s[i])
                append (s,s[i])
                count_equal=count_equal+1
            }
    then we find every s that is more than v and move it to the end of the
    s and also count such occurences,
        for (i in 1:length(s)){
            if (s[i]>v)
                append (s,s[i])
                remove (s[i])
                count_higher=count_higher+1
            }
        }

    now, we know the counts of each segment
    (I start indexing in 1 like in R)

        lower=1:length(s)-count_equal-count_higher
        equal=lower+1 : lower+1+count_equal
        higher=equal+1 : length(data)

    now we can remove the parts that do not contain the nth largest number
    (like the original algorithm) and reiterate.


```

This algorithms will only move values within the data around and never create new data in the memory, as in each iteration, it will only find the places of the start and end of three groups, and remove them.

3 Goddard A3 Question 3

Here, we can first sort using merge-sort, which is a divide and conquer algorithm. Then, a loop like this easily gives us the majority number in $O(n)$

```
1
2       a=sort(a)
3
4       majority=length(a)/2
5       count=1
6
7       for(i in 2:(length(a))){
8           if (a[i-1]==a[i]){
9               count=count+1
10              if (count>majority){
11                  print(a[i])
12                  break
13              }
14          }
15          else {
16              count=1
17          }
18      }
19
```

Assuming the sort is done using the merge-sort algorithm, this for loop only needs to go through the data once, hence will finish in $O(n)$. So, the whole algorithm will take $O(n \log n)$ to finish.