HW 3 – Algorithms

Sanjay Roberts

February 12, 2019

1

1.1 a - Super Mode Divide and Conquer

Psuedocode:

```
Input: List $A$ of size $n$
Output: Super-mode for the List

Supermode_divide(A):
if $A.size == 1$ then
    return A.element
end if
minimum_length = A.size // 2
A_left = [1 through minimum_length - 1]
A_right = [minimum_length through A.size]
Recursive_left = Supermode_divide(A_left)
Recursive_right = Supermode_divide(A_right)
if Recursive_left == Recursive_right then
    return Recursive_right
end if
if A.count(Recursive_right) > minimum_length then
    return Recursive_right
end if
if A.count(Recursive_left) > minimum_length then
    return Recursive_left
end if
return "No supermode" =0
```

Explanation:
The main idea is that if there is a supermode in our list, then that value must be the supermode in either the left split list *A_left* or the right split list *A_right*. And if there is a supermode for either of the two sublists, then that value must be the supermode for the sublists' sublists. And on and on.

Using divide and conquer to find the supermode, first we define the minimum length to be $\frac{n}{2}$. Then we split the inputted list into half, creating two sub-lists *left* and *right*. From there, we call the function recursively to split the sublists again, first acting on the right list. The recursive calls goes back through the algorithm, defining the minimum length to be $\frac{n}{4}$. We go through the code and again recursively call the function to get sub-lists of the first right sublist. This continues, recalculating the mininimum length and dividing sublists down the furthest right sublist until we end up with a minimum length of 1, and a sublist size of 1.

At this point, due to the first 2 lines of code in the method, we get a return value for Recursive_left and Recursive_right. We drop into the *if* statements and compare the two values. If they are the same, then for that level of the recursion, we have a supermode for the combined sublists and return that value. If the Recursive_left and Recursive_right values are not the same, then we count the number of times the Recursive_right value shows up in the combined sublists, and if it is greater than the minimum length for that level of the recursion, we return that as the supermode for the combined sublist of that level. If the Recursive_right value count is not larger than the minimum length at the level, we try the Recursive_left value count. If neither count is large enough, then we return "No supermode" for the combined sublists for that level of the recursion.

Next we go back up a recursion level to the previous two sublists and redo this scenario. Then we should have a supermode or "No supermode" defined for the next level up of recursion. We repeat this until we have entirely found and compared supermodes/"no supermodes" for the entire right sublist we started with. Then we turn around and do the same thing on the entire left sublist. At this point we should have a supermode/"no supermode" for the first level of recursion's right sublist and left sublist. Again we drop into the *if* statements for a final time. If Recursive_left and Recursive_right have the same value for their supermodes, then that is the supermode for the list. If one side has a supermode and the other side has "no supermode", then we count the supermode value in both sublists and if it is greater than the minimum length, then it is the supermode for the entire list. If both sides end up with "no supermode", then there is no supermode for the list. Else if they both have a supermode value that are not the same, we count the number of times each supermode shows up in the combined sublists, and if one supermode is greater than the minimum length ($\frac{n}{2}$), then it is the supermode for the list.

The runtime is $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) \in \theta(nlogn)$, according to Master's theorem. This is the same as mergesort, for pretty much the same reason as mergesort. Each recursive level, we split the problem into 2 smaller chunks, so each step of the recursion is $log(n)$. On each recursive level, we have to do at most cn operations if we need to count how many times a supermode value shows up in the given sublist for that level, so $\mathcal{O}(n)$.

1.2 b - Super Mode Decrease and Conquer

Psuedocode:

Input: List A of size n

Output: Super-mode for the List

```
Supermode_decrease(A, extra=None):
A_length = A.size
if A_length == 0 then
   return extra
end if
$tuples\_list = []$
if $A\_length$ modulo 2 == 1 then
   $extra = A[-1]$
end if
for $i \leftarrow 0$ to $A\_length - 1$, by 2 do
   if $A[i] == A[i+1]$ then
      $tuples\_list.append(A[i])$
   end if
end for
Supermode = Supermode_decrease(tuples_list, extra)
if Supermode is None then
   return "No supermode"
end if
$Supermode\_count = A.count(Supermode)$
if Supermode_count > $\frac{A\_length}{2}$ then
   return Supermode
end if
if Supermode_count == $\frac{A\_length}{2}$ and Supermode == extra then
   return Supermode
end if
return "No supermode" =0
```

Explanation:

Similar to Divide and Conquer, the main idea for Decrease and conquer is that if a supermode is present in the input list, then it should be present in its sublists. In this case, we make tuples of the list values i.e. (element 1, element 2), (element 3, element 4), (element 5, element 6), and so on, and if the tuple values match, then we add that value to a new list. For each pair that do not match, even if the supermode for the list is present, we throw out the pair (i.e. decrease). Once we go through the original list, we recursively call the function on the new list.

If there is a supermode for an even length list, then it should at least show up once next to itself, i.e. there is a pair of supermodes next to each other. If the list length is odd, we keep track of the last element of the list (*extra* in the code above) and use it to determine the supermode if we end up with two potential supermodes.

For example, if list length = 8 or 9, then the supermode needs to occur 5 times in either case. If the value 4 is the supermode for both, then a 4-4 pair will have to occur in the list length = 8 $\rightarrow$ [4,3,4,2,4,6,4,4]. It's not the case for list length = 9 $\rightarrow$ [4,3,4,3,4,3,4,3,4], but in that case, 4 would be the last element of the list = *extra*. By the method laid out, such as [3,3,3,3,4,4,4,4,4], we might end up with 2 supermode options for sublists, 3 and 4, in which case *extra* = 4, which would break the tie and determine 4 as the supermode.

Step by step: First we get the length of the list. If length is odd, we keep the last number stored in *extra*. Then we pair off the leftover values in the list. If the paired values match, we keep that value in *tuples_list*. Then we call the function recursively on *tuples_list* and *extra*. We repeat until we end up with an empty *tuples_list* i.e. none of the values in the previous *tuples_list* matched. At this point, we return *extra* for the *Supermode* variable. We drop into the *if* statements. If *extra* is None, then we return "No supermode" for that level of the recursion. Else, we go to the next statement and count the supermode value in the above recursion. If the supermode's count is greater than the length of the list for that recursion, divided by 2, then it is the supermode for that level of the recursion. Else, if it is equal to the length of the list for that recursion, divided by 2, then we check the value of *extra*, and if it matches the supermode, then it breaks the tie for that level, and it is the supermode for that level of the recursion. Else if *extra* doesn't match, then we return "No supermode".

We go up a level and use the list length, *extra* value for that level of recusion, and the supermode, and go through the *if* statements again. At any point that we don't return a supermode, then there is no supermode for the list. Else we keep moving up until we get to the orginal list and check the *if* statements to determine if the supermode from all the previous decreased lists translates to a supermode for the entire list.

The runtime is $T(n) = T(\frac{n}{2}) + \mathcal{O}(n) \in \theta(n)$ according to Master's theorem. By pairing the elements in the list and throwing out non-matching pairs, we decrease the problem by at least half each time $\rightarrow [2, 2, 2, 2, 2]$, after one recursion, the list will be [2,2] and *extra* = 2. Next recursion is [2] and *extra* = 2. Last recursion is [] and *extra* = 2. We add n since we have a *for* statement that must go through the list to create pairs, and once we have a potential supermode, we also count at most n times in the list to see how many times it occurs.

2

2.1 a

Wikipedia defines a metric space as a set together with a metric on a set, where the metric function defines 'distance' between two members of the set. We are given two rankings, r and s taken from the same set of n movies. Each ranking is a permutation of the same movie ids. We define distance as $d(r, s)$ where we count inconsistencies in s with respect to r.

2.1.1 Identity

Using the above, we can say that the distance from a point to itself is $0 \rightarrow d(s, s) = 0$ since the set of rankings s will contain the same set of movie ids in the exact same position, which means there will not be any inconsistencies between the sets. This fulfills the identity aspect of metric spaces

2.1.2 Symmetry

Given that the set of rankings in r and s have the same permutation of movies, but do not contain the same movie id positions, or that the two sets rank the movies differently, we can say that $d(r, s) = d(s, r)$ since the way in which we determine an inconsistency is the same regardless if we take s with respect to r, or r with respect to s. This fulfills the symmetry aspect of metric spaces.

2.1.3 Triangle inequality

Let $d(r, s) = 0$ and $d(s, t) = 0$. Then r, s, t all have the same set of movie ids with rankings in the same positions. Therefore, $0 = d(r, t) \leq d(r, s) + d(s, t)$.

Let $d(r, s) = 0$ and $d(s, t) = x$. Then r, s have the same set of movie ids with rankings in the same positions. Therefore, $x = d(r, t) = d(s, t) \leq d(r, s) + d(s, t)$

Let $d(r, s) = x$ and $d(s, t) = x$. Then r, t have the same set of movie ids with rankings in the same positions. Therefore, $0 = d(r, t) \leq d(r, s) + d(s, t)$

Thus triangle inequality holds for all cases for this metric space.

2.1.4 Positive definitiveness

For rankings r, s, t, where $r = t$ meaning that they have the same rankings in the positions, from the triangle inequality, we see that

$$d(r, t) \leq d(r, s) + d(s, t)$$

$$d(r, t) - d(s, t) \leq d(r, s)$$

Substitute t for r, use symmetry for second term

$$d(r, r) - d(r, s) \leq d(r, s)$$

and since $d(r,r) = 0$ according to the identity,

$$-d(r,s) \leq d(r,s)$$

which can only happen if $d(r,s) = 0$. So therefore we can say

$$0 = d(r,r) \leq d(r,t) + (t,r) = 2d(r,t)$$

So distance must be positive, and this fulfills the positive definiteness apsect of metric spaces.

2.2 b

```
def d(r,s):
    #check if lists are same length
    if len(r) != len(s):
        return "Lists are not the same length"
    # check if lists have the same elements
    if sorted(r) != sorted(s):
        return 'lists do not have the same elements in them'

    n = len(r)
    position = {}
    # dict of truth. Already sorted
    for i in range(n):
        position[s[i]] = i
    # run inconsistencies on this list and count the diffs
    k = []
    for i in r:
        k.append(position[i])

    global count
    count = 0
    return inconsistencies(k)


def inconsistencies(list):
    global count

    if len(list) > 1:
        middle = len(list) // 2
        left = list[:middle]
        right = list[middle:]

        #recursive calls on left and right lists
        inconsistencies(left)
        inconsistencies(right)

        left_inc = 0
        right_inc = 0
        full_inc = 0

        while left_inc < len(left) and right_inc < len(right):
            if left[left_inc] < right[right_inc]:
                list[full_inc] = left[left_inc]
                left_inc += 1
            else:
                list[full_inc] = right[right_inc]
                right_inc += 1
                count = count + len(left) - left_inc
            full_inc += 1

        while left_inc < len(left):
            list[full_inc] = left[left_inc]
            left_inc += 1
            full_inc += 1

        while right_inc < len(right):
            list[full_inc] = right[right_inc]
            right_inc += 1
            full_inc += 1
        return count
```

2.3 c

Using the two lists given: $[23, 45, 71, 18, 57]$ and $[71, 23, 18, 45, 57]$, I am using the merge sort algorithm but enhancing it by counting the number of inconsistencies between the two lists during the sort step and merge step. I map the first list to a dictionary with the entries as keys and the indexes as values $\rightarrow 23 : 0, 45 : 1, 71 : 2, 18 : 3, 57 : 4$. Then I get the second list in terms of the indexes of the first list $\rightarrow [2, 0, 3, 1, 4]$.

I pass this new list through the *inconsistencies* method, which divides the list in two, and counts the inconsistencies on the right side and left side recursively (conquer). Then during the combine step, I count the number of inconsistencies between the left and right side. Each time an element from the right list is smaller than the elements in the left list, then that element must be smaller than all the elements in the left list, meaning that that element is 'inconsistent' with each element in the left list, so therefore we add the current length of the step list to our count ($count = count + len(left) - left\_inc$). Finally, I return the sum of all three counts (left, right, and merged) to get the total number of inconsistencies.

The largest number of inconsistencies in two n element rankings is $\binom{n}{2}$, and happens when one ranking is reverse of the other ranking $\rightarrow r1 = [1, 2, 3, 4, 5]$, and $r2 = [5, 4, 3, 2, 1]$, so there are 10 inconsistencies between the two rankings.

The runtime is basically the same as merge sort for the same reasons, $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) \in \theta(nlogn)$, according to Master's theorem. This is the same for worst and best case. Setting up the problem, looping through the dictionary, creating the list to run the inconsistencies method takes $\mathcal{O}(n)$ time The divide step takes constant time. The conquer step takes $\mathcal{O}(log(n))$ time. The combine step takes at most $\mathcal{O}(n)$ time

2.4 d

Located in adj_list_NN.csv

2.5 e

Located in adj_list_RNN.csv