Assignment 4

Algorithms & Complexity (CIS 522-01)

Javier Arechalde

March 2, 2018

Part A: Read the solved exercises and Practice

Solved excercise #1 in Chapter 5

In this problem, we are given an array containing n entries. Inside this array, we haev a peak entry p in a position j of the array, so that the array positions prior to p go in increasing order until we reach p, and the values after p go in straight decreasing order.

Our goal is to find that peak entry p without having to read the entire array, and only by reading as few values as possible.

In our algorithm, we will first check the element in the middle of the given array, and we will check if:

- $array[n/2-1] < array[n/2] < array[n/2]+1$
- $array[n/2-1] > array[n/2] > array[n/2]+1$
- $array[n/2-1] < array[n/2] > array[n/2]+1$

In the first case, we are in a positive slope, which means that we still didn't reach the p value, in this case, we will get rid of the left half of the array and continue. In the second case, we are in a negative slope, which means that we already passed the p value, in this case, we will get rid of the right half of the array and continue. In the last case, the value is greater than the value that goes before it, and greater than the value that goes after it, which means that we hit the p value, in this case, we will stop running our algorithm and return p.

We will follow this structure recursively, checking the value in the middle and getting rid of one of the halves of the array until we find the p value.

Algorithm Pseudocode

Algorithm 1 Finding maximum pseudocode

```
1: function FINDMAXIMUM(pos_start, pos_end, array)
2:     n = (pos_start + pos_end)/(2)
3:     if array(n/2 − 1) < array(n/2) < array(n/2 + 1) then
4:         We have a positive slope, so we havent reached the maximum yet
5:         FINDMAXIMUM((pos_start + pos_end)/2, pos_end, array)
6:     else if array(n/2 − 1) > array(n/2) > array(n/2 + 1) then
7:         We have a negative slope, we already passed the maximum
8:         FINDMAXIMUM(pos_start, (pos_start + pos_end)/2)
9:     else if array(n/2 − 1) < array(n/2) > array(n/2 + 1) then
10:        We have found the maximum point
11:        return value(n/2)
12:    end if
13: end function
```

Solution for problem instance of size 10

Now we are going to prove that our algorithm works, by manually running it over a sample set of size 10. This will be our problem's sample set:

$$S = [1, 2, 4, 12, 14, 21, 6, 4, 3, 1]$$

When we start running the algorithm, we will first check the number in the 5^{th} position.

In this case we will have that $S[4] = 12 < S[5] = 14 < S[6] = 21$, which means that we are in a positive slope, and we still haven't reached the peak value p. Then we will call the function again, so in the next iteration of our algorithm, we will work with the set that is on the right side of the value that we checked.

$$S' = [14, 21, 6, 4, 3, 1]$$

Now we will check the 3^{rd} position. In this case we have that $S[2] = 21 > S[3] = 6 > S[4] = 4$, which means that we are in a negative slope, so we already passed the peak value p. Thus, we will call the function again, and in the next iteration of our algorithm we will work with the set at the left of the value we just checked.

$$S'' = [14, 21, 6]$$

Now we will check the middle position, in this case the 2^{nd} position. We get the result that $S[1] = 14 < S[2] = 21 > S[3] = 6$, which means that the number we are checking is indeed the peak entry p. Now we will return that value, and stop running our algorithm.

$$max = 21$$

Time Complexity

In this problem, with each one of the recursive calls, we reduce the problem to one of at most half the size of the initial problem. Then,

$$T(n) \leq T(n/2) + c$$

when $n > 2$, and

$$T(2) \leq c$$

Then, we can confirm that the running time of our algorithm will be $O(\log n)$.

Solved exercise #2 in Chapter 5

In this problem, we have an investment company that stores during n consecutive days the values of a given stock. For each of these days, the stock has a price $p(i)$ per share for the stock on that day. We assume that the stock prize was fixed on that day.

The goal is to find, without having to check each possible combination of days, which will take $O(n^2)$, in which day they should have bought the shares, and in which day they must have sold them to make as much money as possible.

In our algorithm, we will recursively divide our problem while our array is longer than 2, until we are left with two sets of S and S'. Then we will substract the first value to the second value of the array, this way we calculate the potential benefit that could be earned if buying in the first day and selling on the second. We will also calculate the optimal solution of substracting the minimum value on S to the maximum value of S'. In the end we will compare the three options, to see which one yields more benefit, and return that option.

- The optimal solution on S
- The optimal solution on S'
- the optimal solution of $p(j) - p(i)$, over $i \in S$ and $j \in S'$.

Algorithm Pseudocode

Algorithm 2 Stocks Divide-and-Conquer pseudocode

```
1: function MAX(list)
2:     max = list[0]
3:     for element in list do
4:         if element > max then
5:             max = element
6:         else
7:             continue
8:         end if
9:     end for
10:    return max
11: end function
12: function MIN(list)
13:    min = list[0]
14:    for element in list do
15:        if element < min then
16:            min = element
17:        else
18:            continue
19:        end if
20:    end for
21:    return min
22: end function
23: function FINDOPT(array)
24:    if len(array) > 2 then
25:        We divide the array in two halves Larray and Rarray
26:        LSide = FINDOPT(Larray)
27:        RSide = FINDOPT(Rarray)
28:        LOpt = LSide[1] - LSide[0]
29:        ROpt = RSide[1] - RSide[0]
30:        MOpt = MAX(RSide) - MIN(LSide)
31:        Marray = [MIN(LSide),MAX(RSide)]
32:        maxvalue = MAX(LOpt,ROpt,MOpt)
33:        if maxvalue == LOpt then
34:            return LSide
35:        else if maxvalue == ROpt then
36:            return RSide
37:        else if maxvalue == MOpt then
38:            return Marray
39:        end if
40:    else if len(array) <= 2 then
41:        return array
42:    end if
43: end function
```

Solution for problem instance of size 10

Now we are going to prove that our algorithm works, by manually running it over a sample set of size 10. This will be our problem's sample set:

$$S = [7, 4, 3, 2, 5, 7, 10, 20, 14, 7]$$

First, as the array is longer than 2, we will divide it into two sets recursively until we have arrays of length smaller or equal to 2.

$$[7, 4, 3, 2, 5] \quad [7, 10, 20, 14, 7]$$

$$[7, 4] \quad [3, 2, 5] \quad [7, 10] \quad [20, 14, 7]$$

$$[7, 4] \quad [3, 2] \quad [5] \quad [7, 10] \quad [20, 14] \quad [7]$$

Now we are going to compare set by set, if the optimal solution between one set, compare it with the other set, and compare it also with the maximum of the "right" set and the minimum of the "left" set, and see which one gives us a better solution.

Now we are going to compare the sets, to see which is the best solution, between S, S' or $p(j) - p(i)$, over $i \in S$ and $j \in S'$. We will return the best solution in each iteration.

$$[7, 4] \quad \& \quad [3, 2]$$

$$OptL = 4 - 7 = -3 \quad OptR = 2 - 3 = -1 \quad OptM = 3 - 4 = -1$$

$$[5] \quad \& \quad [7, 10]$$

$$OptM = 10 - 5 = 5 \quad OptR = 3$$

$$[20, 14] \quad \& \quad [7]$$

$$OptL = -6 \quad OptM = 7 - 14 = -7$$

We return $[3, 2] \quad [5, 10] \quad [20, 14]$.

Now we have:

$$[3, 2] \quad [5, 10] \quad [20, 14]$$

Lets start comparing.

$$[3, 2] \quad \& \quad [5, 10]$$

$$OptL = -1 \quad OptR = 5 \quad OptM = 10 - 2 = 8$$

We return $[2, 10]$.

Now we have:

$$[2, 10] \quad \& \quad [20, 14]$$

$$OptL = 8 \quad OptR = -6 \quad OptM = 18$$

We finally return $[2, 20]$, and that will be our final result.

Time Complexity

As we said, in this problem, we will have to recursively subdivide our array, and take the best possible solution out of this three possible solutions:

- The optimal solution on S
- The optimal solution on S'
- the optimal solution of $p(j) - p(i)$, over $i \in S$ and $j \in S'$.

The first two items are computed, in time $T(n/2)$ by recursion, and the third item is computed by finding the maximum in S' and the minimum in S, which can be done in $O(n)$ time. Then our running time $T(n)$ satisfies

$$T(n) \leq 2T(\frac{n}{2}) + O(n)$$

,

so the time complexity of our implementation will be $O(n \log n)$.

Part B: Problem Solving

Significant inversion

Problem Model

In this problem, we are given a sequence of n numbers $a_1, ..., a_n$, which we will we assume that are all distinct, and we define inversion to be a pair $i < j$ such that $a_i > a_j$. We call a pair *significant inversion* if $i < j$ and $a_i < 2a_j$.

Our goal is to count the number of *significant inversions* between two orderings, using an algorithm that has $O(n \log n)$ time complexity.

Our implementation will be based on the *merge-sort* algorithm, as we will recursively sort and merge the array, but we will also count the number of significant inversions while at it.

We also added some improvements to the pseudocode. If for example, while we are merging the two sorted arrays, we finish adding elements from one array while there are still elements available in the other array, we will add this remaining elements and exit the while loop.

Another important thing to note is that when we are merging two arrays and checking if there exists a significant inversion, if there is indeed one, there will be as many significant inversions as elements are left in the array on the "left", as the array is sorted in increasing order, then the rest of the values that come after that one will also be greater than the value we are comparing with.

Pseudocode

Algorithm 3 Significant inversion pseudocode

```
1: function CountInversions(array)
2:     if length(array) > 2 then
3:         Divide the array in two halves array_left and array_right and
4:         recursively call this function
5:         array_left = CountInversions(array_left)
6:         array_right = CountInversions(array_right)
7:         while We havent sorted array_left and array_right do
8:             value_left = array_left[i] and value_right = array_right[k]
9:             if We added all the elements in array_left then
10:                array_sorted = array_sorted + array_right[j : end]
11:                Break loop
12:            else if We added all the elements in array_right then
13:                array_sorted = array_sorted + array_left[i : end]
14:                Break loop
15:            end if
16:            if value_left < value_right then
17:                Add value_left to the end of array_sorted
18:                i++
19:            else
20:                Add value_right to the end of array_sorted
21:                j++
22:                if value_left > 2 * value_right then
23:                    All the elements from position i to the end will be greater
24:                    too as array_left is sorted in increasing order
25:                    N_inv = N_inv + length(array_left) − i
26:                end if
27:            end if
28:        end while
29:        return array_sorted
30:    end if
31:    if length(array) == 1 then
32:        return array
33:    else if
34:        if  thenthenarray[0] < array[1]
35:            return array[0], array[1]
36:        else
37:            if array[0] > 2 * array[1] then
38:                N_inv + +
39:            end if
40:            return array[1], array[0]
41:        end if
42:    end if
43: end function
```

Implementation

Here is the code for the implementation of the *pseudocode* shown above.

```
#The array we are going to work with
S = [1,5,4,8,10,2,6,9,12,11,3,7]

inv = 0

def countsinv(array,inv):

 #If the length of the array is greater than two, we divide it in two
 if len(array)>2:

  mid = len(array)/2

  arrayL = array[0:mid]
  arrayR = array[mid:len(array)]

  #We recursively divide our problem
  L_arr,linv = countsinv(arrayL,inv)
  R_arr,rinv = countsinv(arrayR,inv)

  #We add up the two returned inversions
  inv = linv + rinv

  #Initializing counters
  i = 0
  j = 0

  #This is the array in which we are going to merge sort the two given arrays
  Sort_arr = []

  for k in range(0, len(R_arr)+len(L_arr)):

   #If we already completed adding the L_arr, or the R_arr, we add the opposite
   if i == len(L_arr):
    Sort_arr.append(R_arr[j])
    j = j + 1
    continue

   if j == len(R_arr):
    Sort_arr.append(L_arr[i])
    i = i + 1
    continue
```

```
    if L_arr[i]<R_arr[j]:
     Sort_arr.append(L_arr[i])
     i = i + 1
     continue

    else:
     if L_arr[i]>2*R_arr[j]:
      inv = inv + len(L_arr)-(i) #If the position j of the array in the left is g
     Sort_arr.append(R_arr[j])
     j = j + 1

  #We return the sorted list and the number of inversions
  return Sort_arr,inv

 #If the length of the array is one, we just return it
 elif len(array)==1:
  return array,inv

 #Otherwise, if the length of the array is 2, we check if the array is sorted or
 else:
  if array[0]<array[1]:
   return [array[0],array[1]],inv
  else:
   if(array[0]>2*array[1]):
    inv = inv + 1
   return [array[1],array[0]],inv


Sort_arr,n_inv = countsinv(S,inv)
print("Number of significant inversions: %i" %n_inv)
```

Running time

Our algorithm uses mergesort to sort the array and count the number of significant inversions at the same time, so the running time of our algorithm will be $O(n \log n)$

Local minimum

Problem Model

In this problem, we are give a complete binary tree T. They also state that the number of nodes is $n = 2^d - 1$ for some d, which means that the tree is not only complete, but is also perfect, which means that all nodes that have childre, have two children, and all leaves are at the same level. This is extremely important, as if the tree was only complete, instead of perfect, we wont have all the leaves at the same level, so we would have to come up with a method to avoid trying to go deeper into the tree when we were reaching the last level, which is not completely filled and all the leaves are as left as possible.

Each node v of T is labeled with a real number x_v. For each node in the tree, we can only determine its value x_v by probing the node v.

Our goal is to find a local minimum, that is if the label x_v is less than the label x_w for all the nodes w that are joined to v by an edge. We also have to find this local minimum of T using only $O(logn)$ probes to the nodes of T.

In our proposal, we will recursively go deeper into the tree until we reach the leave nodes, once we reach the leave nodes, we will compare the one on the left and the one on the right and return the smaller one, while we go up until the tree root node. This can be seen as a "tournament", as only the smaller value will remain after every round, and in the end we will have the smallest value in the tree.

To explore the tree, we will use a position array called pos_{array} that we will initialize as $\{0\}$ and when we go deeper in the tree, we will append in each level, a 0 if we are exploring the left branch and 1 if we are exploring the right branch. Then when we want to probe a node, we will use this array to reference it. For example, if the tree has $d = 3$ and we want to reference the leftmost leaf, we will do $x_v = T(\{0, 0, 0\})$

Pseudocode

Algorithm 4 Local minimum pseudocode

```
1: function FindMin(T, level, pos_array)
2:     if We haven't reached the leaves of T then
3:         We go down one level in the tree
4:         left_val = FindMin(T, level + 1, pos_array.append(0))
5:         right_val = FindMin(T, level + 1, pos_array.append(1))
6:
7:         We find the minimum value and return it
8:         if left_val < right_val then
9:             return left_val
10:        else
11:            return right_val
12:        end if
13:    end if
14:    if We are on a leaf then
15:        return x_v = T(pos_array)
16:    end if
17: end function
```

Running time

In this problem, with each one of the recursive calls, we reduce the problem to one of at most half the size of the initial problem. Then,

$$T(n) \leq T(n/2) + c$$

when $n > 2$, and

$$T(2) \leq c$$

Then, we can confirm that the running time of our algorithm will be $O(\log n)$.