

ASSIGNMENT 4

ALGORITHMS & COMPLEXITY (CIS 522-01)

Javier Arechalde

March 2, 2018

Part A: Read the solved exercises and Practice

Solved exercise #1 in Chapter 5

In this problem, we have an array with n entries, and inside this array, we have a peak entry p in a position j of the array, so that the values in the array prior to j go in increasing order, and after the peak value, they go in 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.

Algorithm Pseudocode

Algorithm 1 Finding maximum pseudocode

```
1: function FINDMAXIMUM( $pos_{start}, pos_{end}, array$ )
2:    $n = (pos_{start} + pos_{end}) / (2)$ 
3:   if  $array(\frac{n}{2} - 1) < array(\frac{n}{2}) < array(\frac{n}{2} + 1)$  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  $array(\frac{n}{2} - 1) > array(\frac{n}{2}) > array(\frac{n}{2} + 1)$  then
7:     We have a negative slope, we already passed the maximum
8:     FINDMAXIMUM( $pos_{start}, (pos_{start} + pos_{end}) / 2$ )
9:   else if  $array(\frac{n}{2} - 1) > array(n/2) < array(\frac{n}{2} + 1)$  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

In this case, we will run our algorithm, when we have an instance of size 10.

In this problem's scope, we will have an set of increasing numbers that grow until a maximum, followed by another set of numbers that go in decreasing order.

This will be our problem's working set:

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

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

In this case we will have that $S[4] < S[5] < S[6]$, which means that we are in a positive slope, and we still haven't reached the maximum. Then we will call the function again, so in the next iteration of our algorithm, we will work with this set.

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

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

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

Now we will check the middle position, in this case the 2^{nd} position. We get the result that $S[1] < S[2] > S[3]$, which means that the number we are checking is indeed the maximum. 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, 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 looks at n consecutive days 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.

Algorithm Pseudocode

Algorithm 2 Stocks Divide-and-Conquer pseudocode

```
1: function MAX(list)
2:   max
3:   for element in list do
4:     if element > max or max == NA then
5:       max = element
6:     else
7:       continue
8:     end if
9:   end for
10:  return max
11: end function
12:
13: function MIN(list)
14:   min
15:   for element in list do
16:     if element < min or min == NA then
17:       min = element
18:     else
19:       continue
20:     end if
21:   end for
22:  return min
23: end function
24:
25: function FINDOPT(array)
26:   if len(array) > 2 then
27:     Larray = array[0 : len(array)/2]
28:     Rarray = array[len(array)/2 : len(array)]
29:
30:     Lside = FINDOPT(Larray)
31:     Rside = FINDOPT(Rarray)
32:
33:     LOpt = Lside[1] - Lside[0]
34:     ROpt = Rside[1] - Rside[0]
35:     MOpt = MAX(Rside) - MIN(Lside)
36:
37:     Marray = [min(Lside),max(Rside)]
38:
39:     maxvalue = MAX(LOpt,ROpt,MOpt)
40:     if maxvalue == LOpt then
41:       return Lside
42:     else if maxvalue == ROpt then
43:       return Rside
44:     else if maxvalue == MOpt then
45:       return Marray
46:     end if
47:
48:   else if len(array) <= 2 then
49:     return array
50:   end if
51: end function
```

Solution for problem instance of size 10

Now we will compute a solution using our algorithm for a problem instance of size 10.

This will be the set we are going to work with.

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

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.

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

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

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

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

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

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

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

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

,

Then 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, \dots, a_n , which we will 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.

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.

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:      $array_{left} = \text{COUNTINVERSIONS}(array_{left})$ 
6:      $array_{right} = \text{COUNTINVERSIONS}(array_{right})$ 
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  $array[0] < array[1]$  then
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 given 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 children, 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 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 $O(\log n)$ probes to the nodes of T .

In our proposal, we will recursively go deeper into the tree until we reach the leaf nodes, once we reach the leaf 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.

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 havent reached the leaves of T then
3:     We go down one level in the tree
4:      $left_{val} = \text{FINDMIN}(Tree, level + 1, pos_{array}.append(0))$ 
5:      $right_{val} = \text{FINDMIN}(Tree, level + 1, pos_{array}.append(1))$ 
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  $x_v$ 
16:  end if
17: end function
```

Running time