CS210-Theoretical Assignment 1

Shubham Agarwal 13674
Vikas Jain 13788
Dept of Computer Science and Engineering
IIT Kanpur

August 24, 2014

Solution 1

In the given local minima in a grid problem can be through the divide and conquer algorithm paradigm. First, samllest element is found from the middle vertical row and based on the values of its neighbour it can be decided which portion(either left or right of vertical row) to be conquer to search local minima. Then the divided portion can obtained from first can be further divided into 2 parts by a horizontal row in middle. Now, smallest element from middle horizontal row is obtained. But likewise in first case, the direct choice cannot be made of which portion to be consider or which to be left(**SUBTELITY**). Therfore, hte portion to be search for local-minima has to be decided based on both previous and new smallest element obtained in rows.

psuedo code:

```
Int Local_minima_in_grid(M) { //Retutn local minima of grid
L←0; R←n-1;
l←0; r←n-1;
k←1;
while(NOT found) {

if(k mod 2) {

        row_flag← 0; mid← (L+R)/2;
        if(M[l to r,mid] has a local minima)
        {found← TRUE;
        Return M[i,mid];
        else { Let M[i,mid] be smallest element in M[l to r,mid] }

        if(l=0 and r=n-1){ smallest_element = M[i,mid]; index=i; }
        if(smallest_element > M[i,mid]){
```

```
                    if(index > mid)
                    { L← mid+1; row_flag←1; if(col_flag) index←l;}
                    else
                    { R← mid-1; index←r;}
          }
          else {

                    if(M[i,mid+1] < M[i,mid]){ L← mid+1; row_flag←1;}
                    else { R← mid-1;}
                    smallest_element←M[i,mid]; index←i;
          }
          k++;
}
else{
          col_flag← 0; mid← (l+r)/2;
          if(M[mid,L to R] has a local minima)
          { found← TRUE;
          Return M[mid,i];} else {Let M[mid,i] be smallest element in M[mid,L
          to R]}

          if(smallest_element > M[mid,i]){

                    if(index > mid)
                    { l← mid+1; col_flag←1; if(row_flag) index←l;}
                    else
                    { r← mid-1; index←R;}
          }
          else {

                    if(M[mid+1,i] < M[mid,i]){ l← mid+1; col_flag←1;}
                    else { r← mid-1;}
                    smallest_element←M[mid,i]; index←i;
          }
          k++;
} }
}
```

Proof of correctness

It will be sufficient to show that
• On termination while loop will indeed return local minima of grid.
• The while loop terminate.

Let L_i, R_i, l_i, r_i be the respective value of L, R, l, r after i^{th} iteration of while loop.

And smallest element is s.

Assertion: P(i):= After i_{th} iteration there is a local minima in M[l_i,L_i] to M[r_i,R_i].

The assertion can be proved using mathematical induction.

Proof: Assume P(i) holds.

mid = (l+r)/2; (when going from row division to column division)

Take smallest element in M[mid, L to R] equal to s'.

Now, two cases arrived.

1. if s'<s local minima can be surely found in the side of smaller neighbour of s' because we can apply **Explore()** from s' to the samller neighbour and since s'<s it will not cross the walls of divided half region and local minima is in the region obtained in $(i+1)^{th}$ iteration.

2. if s'>s the i^{th} region iteration cannot be divide on the basis of s'. $(i+1)^{th}$ iteraton region should have the smallest element s in the region so it is obtained on this basis. **Explore()** can be applied from s and local minima is obtained in the divided region on the bais of i as s'>s it will not cross the walls of divided region.

Similar argument can be given while going from column division to row division.

Hence after every i^{th} iteration the divided region contains a local minima.

Termination of While loop

Since on every iteration grid is reduced by half, Suppose the worst case that local minima is not obtained on any iteration while checking the whole middle array(obtained by dividing the region in two) then at last iteration, the reduced grid will be only a single array having some elements then the minima of the array has to be local minima because it cannot jump the side walls array since their values are greater than it and the function will return local minima of grid terminating the while loop.

Time Compexity

Since after every iteration the grid is reduced by half hence the total number of steps is equal to log$n$.

Now each time the minimum is found in the middle array and its order depend on the number of elements in the array. Number of elements in array are also reduced by half each time. And the rest of computation in the while loop is done in $O(1)$ time.

Hence, time taken on a particular iteration is proprtional to the no. of elements in the middle array.

Let say the time is c×(no. of elements in mid array)

Total time to run algorithm is

$$\underbrace{cn + \frac{cn}{2} + \frac{cn}{4} + \cdots}_{logn\ times}$$

$= 2cn$

Hence the order of time compexity of the above algortihm is $O(n)$

SOLUTION 2 . a part We'll be going to each element a[i] of array (order n time)and then searching for x-a[i] in that array through BINARY SEARCH (order $\log_2 n$ time)

PSEUDO CODE

```
found← 0
i←0
while(not found and i<n){

        if (binary-search(x-a[i])){

                        found ←1;
                        break;

        }
        else

                        i++;
}
return found ;
}
```

Time complexity analysis
Total number of instructions = 2 + n*($\log_2 n$+ 3) +1
$\Longrightarrow$ **O(n*** $\log_2 n$)
Space complexity analysis
Array of size n + some variables
$\Longrightarrow O(n)$
Proof of correctness
If the program returns true , indeed it found pair (a , b) s.t. a+b=x exist in array .
It will return false only when it has gone to each element of array , checked that it's pair not existed in array .

SOLUTION 2 b PART

We'll set left and right pointers of array . Then check a[l] + a[r] .If sum >key We'll reduce sum by shifting r pointer to left . If sum < key , we'll increase sum by shifting l pointer to left. We'll keep iterating till found or pair not exist in array

Pseudo Code :

```
find-duplet(int key) { l←0
r←n-1
found←0
```

```
while(not found and l≤r)do
{
        if (a[l]+a[r] == key )
        {
                found← 1
                break
        }
        else if (sum>key )
                r← r-1
        else
                l←l+1
}
return found;
}
```

Time Complexity

Number of instructions executed = 3+n×4 + 1 $\implies$ Order n

Space Complexity

Array of size n + few variables

$\implies$ Order n

Proof of correctness

In this proof , we'll be asserting that the elements discarded from array can't pair up with any of elements of array to give sum equals to key .

First of all , If return true , indeed true.

We have to prove that if it returns false , no pair exist in array to give sum equals to key.

Original array is

{ a[0] , a[1] ,a[2] ,a[3] , . . . , a[l] , . . . ,a[r] , a[n-2] ,a[n-1] }

Suppose after some iterations l points to a[l] and r points to a[r] , then

Define Excluded-array as

{ a[0] , a[1] ,a[2] ,a[3] , . . . , a[l-1] , . . . ,a[r+1] , a[n-2] ,a[n-1] }

Also define Remaining-array as

{ a[l] , a[l+1] , . . . , a[r] }

Proof by induction :

Assertion : Any element of excluded-array can't pair up with any element of original-array to give sum equals to key i.e.

$a_m + a_n \neq$ key

$\forall\ a_m \in$ excluded-array

$\forall\ a_n \in$ original-array

Base-Case : At starting , Excluded array is empty .

Induction-Hypothesis : After some iterations assume left poiner at a[l] and right pointer at a[r]. Assume that assertion is true here .

Proof : In next step , either a[l] will enter excluded-array or a[r].

Case-1 : a[l] enters

Claim : $a_l + a_p \neq$ key

$\forall a_p \in original - array$

Proof : $a_l + a_r < key$:implies from code

$\implies a_l + a_i < key \forall i <= r \ldots (1)$

(Because array is sorted)

Now we have to show for $i > r$
Consider , a_{r+1} which is in excluded-array because
$a_{r+1} + a_i >$ key for some $i < l$
$\implies a_{r+1} + a_l > key$
$\implies a_l + ai > key \forall i > r \cdots (2)$
$\implies a_i + a_l \neq key \ \forall i \in [0, n-1]$
$\implies H.P.$
Case-2 : a[r] enters excluded array
Claim : $a_r + a_p \neq key \ \forall a_p \in original - array$
Proof : $a_r + a_l > key$:Clear from code
$\implies a_r + a_p > key \ \forall p \in [l, n-1]$
(Because array is sorted)
Now , we have to show it true for p < l
Observe that $a_{l-1}$is in excluded array because
$a_{l-1} + a_i < key$ for some $i >= r$
$\implies a_{l-1} + a_r < key$
$\implies a_i + a_r < key \forall i < l$
$\implies a_i + a_r \neq key \forall a_p \in original - array$
$H.P.$

SOLUTION 2 . c PART

In this part , take c equals to a[p] . p varies from 0 to n-1 (order n time) . Now , treat c as key and check find-duplet(c) (order n time). In this way ,one could find a triplet a,b ,c such that $a + b = c$ in order n^2 time .

1 Pseudo Code:

```
find-triplet {
i← 0
found← 0
for( i=0 ; i<n ; i++) {
        c←a[i];
        if(find-duplet(c)){
                found ← 1;
                break ; }
        }
return found; }
```

Time Complexity
Number of instructions = 2+ n ×(n + 2)
$\implies$ order n^2 complexity
Space Complexity
Order n
Proof of correctness
Again we are going to each element and treating it as key ,finding the other 2 using find-duplet

function .
If return true , indeed program found a triplet . If returned false ,no triplets existed in array because we have looped through entire array and treated each element as x . Then tried to find a and b using previous algorithm .

Solution 3

The given Range minima in a array problem, the data structure should have size of $O(n)$ and each query time to be $O(logn)$.
The given problem can be solved by dividing the given array appropriately. The array is divided into **blocks of elements** each having **log$n$ elements**. Then the minimum element of each block is stored in separate array. This new array will have size of $\frac{n}{logn}$ elements. Similarly this new array can also be divided into **blocks of log(size of aaray)** elements and the minimum of these blocks are again store in new array.
In the same way the minimum values of the blocks of the array can be store recursively and during query time these values can be exploited to obtain the result in required computation time.
The data strucure can be made using dynamicly allocated 2-D array(say A[ ][ ]).
The first row will have n elements (given array),second row will have $\frac{n}{logn}$ elements, third row $\frac{n/logn}{log(n/logn)}$ elements and so on.

Now for query inputs i and j, the given range is divided into **[i,c_0]**, **[c_0,c_1]**, **[c_1,c_2]**, ..., **[c_{k-1},c_k]**, **[c_k,j]** where c_0, c_1, c_2, ..., c_k are the first elements of the blocks obtained by dividing array. Now the minima of Range(i,j) will be the minima of minima of all these blocks and the minima of center blocks are stored in data structure and can be found out recursively.

pseudo code:

Function will be used: **Log**[ ](defined in class).

```
Int min(i, j, n, A[ ][ ], index){
sm_initial← A[index][i];
sm_last← A[index][j];

while(1){
        if(i mod Log[n] = 0) { c_0=i; break; }
        i++;
        if(sm_initial < A[index][i])
        sm_initial=A[index][i];

} while(1){
        if(j mod Log[n] = 0) { c_k=j; break; }
        j–;
        if(sm_last < A[index][j])
        sm_last=A[index][j];

}
p← i/Log[n];
```

q← j/Log[n];
if(n mod Log[n] = 0)
t← n/Log[n];
else
t← n/Log[n]+1;

if(j-i<=3) return minimum of A[index][i to j];
else
return minimum of sm_initial, min(p, q, t, A, index+1) and sm_last;
} }

Time compexity

Time compexity analysis can be divided into two parts. First, number of recursion or stack level maintained for query and second, the time analysis of while loop in for each function call.

Each time the number of elements in the array are get divided by log(no. of elements) itself. if the number is divided by 2, the total number of steps are logn. Since log(no. of elements in array) is greater than 2 for n>= 4. Hence it is confirmed that no. of steps will be always less than logn.

Now, at each step while loop is getting evaluated. First **it will evaluate logn elements then log(n/logn) elments and so on**. The argument used to analysetime comlexity of first question can be used here.

$$\underbrace{cn + \frac{cn}{2} + \frac{cn}{4} + \cdots}_{logn\, times}$$
$$= 2cn$$

Since each time n is divided by logn(and in above case it is divided by 2) therefore summation will decrease only.

$$\underbrace{clogn + c\frac{n}{logn} + \frac{n/logn}{log(n/logn)} + \cdots}_{logn\, times}$$
$$< 2clogn$$

The rest of the function except while loop can be executed in $O(1)$ time. Hence the total time compexity of each query time is $O(logn)$.

Data structure space compexity: Since each time array is divided by log(no. of elements in array) and the no. of such array will always be less than log(size of given array,n). The first array obtained will be of n/logn alements and the rest will be smaller than it.

So,it is confirmed that size will always be less than
$\log n \times \frac{n}{logn} = n$.
Hence space compexity of data structure is $O(n)$.