A2-S20190010034

C Bhavesh Kumar

September 2020

1 Proof of Correctness

(a) The pseudo code is for sorting algorithm called selection sort.
We prove this by using mathematical induction.
To prove this by using mathematical induction we use a loop invariant.
Loop Invariant:A[0:i] is sorted
Inductive hypothesis:Loop variant is true at the end of i-th iteration of the outer loop.
Base case: i=0, the array A[:0] contains only one element which is sorted and hence induction hypothesis is true for base case.
Induction part: Let the hypothesis is true for any k where $k < n$ and where n is size of array.
That is, after the k-th iteration A[:k] is sorted.
$=> a[0] < a[1] < ..... < a[k-1]$
$=> a[x] > a[i]\ \forall i < k$ and $x >= k$
Now for k+1 iteration,
We find the minimum element in the array A[k:]
And we swap it with a[k] and therefore a[k] has the minimum element of the array A[k:]
Now from the k-th iteration we have $=> a[0] < a[1] < ..... < a[k-1]$ and we have that $a[k] >= a[x] \forall x < k$ and $a[k] <= a[x]\ \forall x > k$
Hence the array element is at the right position after the end of iteration.
$=> a[0] < a[1] < a[2] < .... < a[k-1] < a[k]$
Hence the array is sorted and by mathematical induction the given algorithm works fine.

(b) Written code in c and submitted in code file.
For multiple inputs the array is getting sorted and the code is working without any problem.

2 New friends

(a) (a) The algorithm for $O(n^2)$ is given below:
It is a simple brute force approach where you check all the distinct pairs possible.

```
entry=[1,2,7,9,6]
exit=[4,5,8,10,10]
n=len(entry)
for i=1 to n:
   for j=i+1 to n:
      if((entry[j]<=entry[i] and exit[j]>=entry[i]) or (
    entry[j]>=entry[i] and entry[j]<=exit[i])) then
                cnt+=1;

```

This algorithm checks all possiblities of pairs and count the necessary and valid ones.
Since nested loop is invloved the time complexity is $O(n^2)$, That is,
For every i the inner loop will execute $n-i$ times
And i goes for 1 to n
Therefore, the sum $(n-1)+(n-2)+(n-3)+.....+3+2+1$ is the time complexity
Which is $O(n*(n+1)/2)$
Therefore the time complexity is $O(n^2)$

(b) The code for the above algorithm is written and submitted in code file.

(b) (a) The algorithm for O(nlogn) approach is discussed below.

```
entry=[1,2,7,9,6]
exit=[4,5,8,10,10]
n=len(entry)
entry.sort()
exit.sort()
guest=i=1
j=0
cnt=0
while i<n and j<n:
    if entry[i]<=exit[i] then
        guest+=1
        cnt+=1
    else:
        guest-=1
        cnt+=guest
        j+=1
while j<n:
    guest-=1
    cnt+=guest
    j+=1

```

The above code is the $O(nlogn)$ version of the previous question.
In the above code what we actually do is that we calculate the distinct pairs that a exiting guest can make, suppose at a time t there are three guests a,b,c and guest c leaves, then guest c can make 2 distinct pairs with a and b which are (a,c) and (b,c).
So the above code sorts the guest intime and exittime and calculates number of guests at a particular time, and when a guest exits we

calculate the distinct pairs he can make with the guests he met.
The Time complexity of above code is O(nlogn).
This is because we need nlogn time for sorting and an addition of n time for parsing the arrays.In the worst case we parse the arrays separately,that is,all the guests enter before anyone exit,In this case we parse the two arrays completely and we do it in n+n time.
Which is nothing but O(nlogn+n+n)
$=> O(nlogn)$

(b) The code for the above algorithm is written and submitted in code file.

3 Needlessly complicating the issue

(a) Linear time algorithm for finding minimum.

```
a=[.....]
n=len(a)
m=INF
for i=1 to n:
    m=min(m,a[i])

```

(b) It is indeed true that any algorithm must do atleast n operations when finding a minimum of n items.
Since we do not know whether the array is sorted, if we have to find out the minimum element we need to visit all the elements of array and check.
Visiting all elements in array would take O(n) time.
Hence no algorithm can find the minimum in less than n operations.

(c) The code for this program have been written in code file and is submitted.

(d) The blank in the pseudo code is filled as return A[0].

```
Input: List A = [a1, . . . , an] of n items
Output: mini{a1, . . . , an}
if n=1 then
    return A[0]
A1 = A[0 : n/2]
A2 = A[n/2 : n]
return min(findMinimum(A1), findMinimum(A2) )

```

This algorithm is correct with the choice i made for the blank because when the length of array is 1, then the element itself is the minimum and hence it is returned as the answer.

(e) Analysis of running time: O(n)
This algorithm has the running time as O(n).
This can be obtained from the recurrence relation created.
$T(n) = 2T(n/2) + 1$

This equation is formed because as each part is divided into 2 subparts and each part does only one operation which is the comparison operation. Using master theorem with $a = 2$, $b = 2$ and $d = 0$,we have $a > b^d$ $[, i.e, 2 > 2^0]$ and therefore the time complexity is
$T(n) = O(n^{log_2(2)})$
$T(n) = O(n)$
This algorithm is similar to the algorithm in part(a), this is because ultimately we are visiting all the array elements and finding minimum.
The only difference is that in this algorithm we use the divide and conquer approach where we find minimum for two sub problems and hence finding the minimum for the problem just by comparing the minimum of the two. This is a recursive approach while the part(a) is iterative approach.But even so they share same running time complexity which is O(n).

(f) The code for this algorithm has been written in c and submitted in code file.

4 Recursive local minimum finding

(a) Recursive local minimum finding for 1-d array.

i Recursive algorithm for local minimum in 1d array in $O(logn)$ time.

```
localminimum(a,l,r):
    if l<r then
        n=len(a)
        mid=(l+r)/2
        if mid-1>=0 a[mid]>a[mid-1] then
            return localminimum(a,l,mid-1)
        elif mid+1<n a[mid]>a[mid+1] then
            return localminimum(a,mid+1,r)
        else
            return mid

```

ii Proof of correctness
This algorithm is indeed correct.We prove it by analysing its three cases and proving them right,
We start from the middle element of the array assuming array consists of only distinct elements.
Case 1: if $a[mid] > a[mid - 1]$
In this case the local minimum is in the left side of the array divided at mid.we prove this by contradiction.
Let a[i] is not local minimum for $0 \leq i < mid$
Then a[mid-1] is not local minimum, therefore $a[mid-2] < a[mid-1]$
Similarily with a[mid-2], $a[mid - 3] < a[mid - 2]$
Continuing this we get $a[0] < a[1]$ which implies that a[0] is local minimum, which contradicts $0 \leq i < mid$ as i is 0 in this case.
Therefore our assumption is wrong and local minimum exists in left

side in this case.
Case 2:if $a[mid] > a[mid+1]$
This is similar to case 1 , the difference is that we proceed towards right rather than left.
This is symmetrical to case 1 and we can prove this too with contradiction.
Case 3: if $a[mid] < a[mid-1]$ and $a[mid] < a[mid-1]$
In this case a[mid] is the local minimum and we return it.
Hence the given algorithm is correct.

iii Runtime analysis
The given algorithm follows the equation $T(n) = T(n/2) + c$
According to master theorem with $a = 1, b = 2$ and $d = 0$, we get $a = b^d$ and hence the time complexity is $O(logn)$

iv The c program for this algorithm has been written in code file and submitted.

(a) Recursive local minimum for 2d array.

i Recursive algorithm for local minimum in 2d array in O(n) time

```
local_min_grid(a,l,r):
    if l<=r:
        n=len(a)
        mid=(l+r)/2
        min_column=find_min_col_index(a) //This returns
    the column number of the minimum number is row number=
    mid.
        min_row=find_min_row_index(a,min_column,mid)//This
     returns the minimum index represented by column
    number min_column in the rows, mid,mid+1,mid-1
        if(min_row==mid) return a[min_row][min_column];
        else if(min_row<mid) return local_min_grid(n,a,l,
    mid-1);
        else if(min_row>mid) return local_min_grid(n,a,mid
    +1,r);

```

ii Correctness of algorithm
We will start by taking the middle row as mid
Then we will find the minimum element index in this row and assign it to mincolumn.Thus it gives the the 1d array local minimum of the row mid.
Then we will find the minimum element index of the rows, mid,mid+1,mid-1 with column index as mincolumn.
We will assign this value to minrow.
Case 1:if $minrow == mid$ then this is our local minimum of 2d array which we return.
Case 2:if $minrow < mid$ then it means that our local minimum is in the upper half of the matrix and so we recursively traverse upper half of the grid.

We will prove this by contradiction.
Assume that there is no local minimum in upperhalf, i.e, a[i][mincolumn] is not local minimum for $0 \leq i < mid$.
Then a[mid-1][mincolumn] is not local minimum, this implies $a[mid-2][mincolumn] < a[mid-1][mincolumn]$.
a[mid-2] is also not local minimum, this implies $a[mid-3][mincolumn] < a[mid-2][mincolumn]$.
Continuing this we obtains $a[0][mincolumn] < a[1][mincolumn]$, but this is local minimum.
This contradicts our assumption a[i][mincolumn] is not local minimum for $0 \leq i < mid$.
Hence our assumption is wrong and local minimum exists in the upper half.
Case 3:if $minrow > mid$ then it means that our local minimum is in the lower half of the matrix and so we recursively traverse lower half of the grid.
This case is also similar to case 2.Infact the matrix is symmetrical and can be proven right as the same way in case 2.With the difference being traversing the lower part of the matrix instead of traversing upper part.
Hence this case can also be proved right using contradiction.
Hence this algorithm is correct.

iii Runtime analysis of the algorithm
From the algorithm we can obtain the recurrence relation $T(n^2) = T(n^2/2) + O(n)$
Here O(n) is for the linear search of the column with minimum element in the row.
And at each step the problem size is reduced by half so $T(n^2/2)$ substitute $N = n^2$
We get $T(N) = T(N/2) + \sqrt{N}$
From masters theorem with a=1,b=2 and d=1/2 we get $a < b^d$
$T(N) = O(N^{1/2})$
$=> T(N) = O((n^2)^{1/2})$
$=> T(N) = O(n)$
Therefore the running time complexity of the algorithm is O(n)

iv The c program for this algorithm has been written in code file and submitted.