Theoretical Assignment 1 (ESO207)

Dhruv Gupta (220361)

September 2023

Question 1. Ideal profits **(40 points)**

In the X world, companies have a hierarchical structure to form a large binary tree network (can be assumed to be a perfect binary tree). Thus every company has two sub companies as their children with the root as company X. The total number of companies in the structure is N. The wealth of each company follow the same general trend and doubles after every month. Also after every year, half of the wealth is distributed to the two child companies (i.e. one fourth to each) if they exist (i.e. the leaf node companies do not distribute their wealth). You can assume that at the end of the year, the month (doubling) operation happens before the year (distribution) operation. Also the sharing operation at the end of the year happens simultaneously for all the companies. Given the initial wealth of each of the N companies, you want to determine the final wealth of each company after m months. (A perfect binary tree is a special tree such that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps. Detailed explanation here)

(a) (20 points) Design an algorithm in $O(n^3 \log(m))$ complexity to find the final wealth of each company after m months.

(b) (10 points) Analyze the time complexity of your algorithm and briefly argue about the correctness of your solution.

(c) (10 points) Consider the case of a single company (i.e. only root) in the tree. Give a constant time solution to find the final wealth after m months.

Solution 1:

(a) We will first implement level order traversal algorithm to convert given binary tree to array using a queue. We will assume we have following functions defined for a queue as they were defined in lecture slides – **isEmpty(Q), Front(Q), CreateEmptyQueue(Q), Enqueue(x,Q), Dequeue(Q).**

```
PBT_to_Array(T,n){
  int A[n]; // n = total number of nodes.
  CreateEmptyQueue(Q);
  p ← T; // p becomes a pointer to the root of the tree.
  Enqueue(p);
  i←0;
  while( not isEmpty(Q) ){
    if( p->left <> NULL) { Enqueue(p->left, Q) }
```

```
    if( p->right <> NULL) { Enqueue(p->right, Q) }
    A[i] ←Dequeue(Q)
    i++;
  }
  return A;
}
```

Consider the case when number of companies n>1. Let $N_k(y)$ denote the wealth of company k at the end of y years (after distribution).
And $P_k(y)$ denotes the wealth of the parent company of company k at the end of y years.
Now, note that
$N_k(y) = 2^{11}N_k(y-1)$, if k is the root node.
$N_k(y) = 2^{11}N_k(y-1) + 2^{10}P_k(y-1)$, if k is neither the root node, neither a leaf node.
$N_k(y) = 2^{12}N_k(y-1) + 2^{10}P_k(y-1)$, if k is a leaf node.

For m months, we first calculate for y = $\left\lfloor \frac{m}{12} \right\rfloor$ years, and then for remaining m-12$\left\lfloor \frac{m}{12} \right\rfloor$ months we just have to multiply by $2^{(m-12\left\lfloor \frac{m}{12} \right\rfloor)}$.

Initially A = **PBT_to_Array(T,n)** would store the initial wealth of each company. Suppose A[k] stores the value of node k. Note that parent node of node k will be $\left\lfloor \frac{k-1}{2} \right\rfloor$.
Above equations can be captured using matrix multiplication. We use A as n x 1 matrix. And we create a matrix M, with values as follows –
M[i][j] = $\frac{1}{2}$ if i = j ≤ (n-1)/2
M[i][j] = 1 if i = j > (n-1)/2
M[i][j] = $\frac{1}{4}$ if j = $\left\lfloor \frac{i-1}{2} \right\rfloor$
M[i][j] = 0 otherwise.

$$M = \begin{bmatrix} 1/2 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 1/4 & 1/2 & 0 & \ddots & & & & \vdots \\ 1/4 & 0 & 1/2 & 0 & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & 0 & 1 & 0 & 0 \\ \vdots & & & & \ddots & 0 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{bmatrix}$$

Let $A^{(m)}$ denote the values after m months. Then $A^{(m)} = 2^m M^{\left\lfloor \frac{m}{12} \right\rfloor} A$.
To calculate 2^m we can use bitshift operator to compute it in constant time. (Explained in part (c))
To calculate M^k we can use matrix exponentiation. (Pow_matrix function)
And to multiply M with A we can use M_V_multiply function defined below.

M_V_multiply(M,V,n){

```
//Returns the ans array after multiplying M and V, where M is n x n matrix and V is an
array of size n.
//Can be seen as multiplying two matrices of dimensions n x n and n x 1.
// Time complexity = O(n^2)
 for(i=0 to n-1){
    Ans[i] ←0;
    for(k = 0 to n-1){
      Ans[i] ← Ans[i] + M[i][k]*A[k];
    }
  return Ans;
}

Pow_matrix(M,k,n){
  If(n=1) return M;
  if(n is even) return Pow_matrix(M, k/2,n) * Pow_matrix(M, k/2,n);
  if(n is odd) return Pow_matrix(M, (k-1)/2,n) * Pow_matrix(M, (k-1)/2,n) * M;
} // Here by * we mean matrix multiplication, which can be done as per following
code.(Matrix_multiply function)
// n is the order of the Matrix M.

Matrix_multiply(A,B,n){
//Returns the matrix multiplication of A and B which are n x n matrices in O(n^3)
  for(i=0 to n-1){
    for(j=0 to n-1){
      C[i][j] ← 0;
      for(k=0 to n-1){
        C[i][j] ← C[i][j] + A[i][k]*B[k][j];
      }
    }
  }
  return C;
}
```

(b) **Time complexity:**

We first convert the given tree to array using PBT_to_Array(T,n) function.
Time taken by one functional call of PBT_to_Array
T(n) = a + F(n) , where F(n) is time taken by the while loop and a is some constant.
Clearly, we Enqueue and dequeue each node exactly once until the while loop ends. And for each node we perform some constant number of comparisons or assignment operations. Hence total run time for the while loop must be b*n, where b is a constant and n is the total number of nodes. Hence, F(n) = b*n
So, T(n) = a+b*n = O(n).

For creating the matrix M, we need to visit each element exactly once to assign it its value, so this will take $O(n^2)$ time.

Then for calculating $A^{(m)} = 2^m M^{\lfloor \frac{m}{12} \rfloor} A$
Time for calculating $2^m = O(1)$
Time for calculating $M^{\lfloor \frac{m}{12} \rfloor}$ = constant *number of times Pow_matrix is called * Time taken for matrix multiplication = $O(1) * O(\log\lfloor \frac{m}{12} \rfloor) * O(n^3) = O(n^3 \log(m))$
Time for Multiplying $M^{\lfloor \frac{m}{12} \rfloor}$ by A = $O(n^2)$ [M_V_multiply]
Finally multiplication by 2^m will take O(n) time, since each element of $M^{\lfloor \frac{m}{12} \rfloor} A$ will be multiplied once.
Hence total time = $O(1) + O(n^3 \log(m)) + O(n^2) + O(n) = O(n^3 \log(m))$

Proof of Correctness:
For proof of correctness of the algorithm we need to prove the correctness of the recurrence relations we have made and the way we have used them to form our matrix.
Assume that at the end of y years N node has value N(y).
Then after 12 months, each nodes value becomes 2^{12} times (before distribution).
After distribution, For root node, $N(y+1) = 2^{12}N(y) * (1/2) = 2^{11}N(y)$.
Each leaf node receives one fourth of their parents have at that moment.
For leaf nodes, $N(y+1) = 2^{12}N(y) + (1/4) * (2^{12}P(y)) = 2^{12}N(y) + 2^{10}P(y)$
Every other node will give as well as get, so
For every other node, $N(y+1) = 2^{12}N(y) *(1/2) + + (1/4) * (2^{12}P(y)) = 2^{11}N(y) + 2^{10}P(y)$

Note that elements in M are so placed that only the values which are multiplied by the node itself (k) or its parent ($\lfloor \frac{k-1}{2} \rfloor$) while multiplication with a vector are non zero.
All other terms are zero. Also $2^{12}M$ when multiplied by the vector $[N_0(y)…N_k(y)…N_{n-1}(y)]$ will yield the same equations as above. N_0 is the root node, Since we know there are total (n+1)/2 leaf nodes, N_k is a leaf node when k > (n-1)/2. Rest all are intermediate nodes.
So for y years multiplying $(2^{12}M)^y$ to Initial vector A would work.
For m months we have $\lfloor \frac{m}{12} \rfloor$ years, and $m-12\lfloor \frac{m}{12} \rfloor$ months. So after m months final vector will be $2^{(m-12\lfloor \frac{m}{12} \rfloor)} * 2^{12\lfloor \frac{m}{12} \rfloor} M^{\lfloor \frac{m}{12} \rfloor} A = 2^m M^{\lfloor \frac{m}{12} \rfloor} A$.
This proves the validity of our algorithm.

(c) For a single node tree, i.e. the root node, we have Value after m months = 2^m * Initial Wealth. Since there is no child, no distribution takes place and wealth gets doubles each month. For calculating this in constant time, we need a way to find 2^m in constant time. This can be done using the **left bit shift operator** (denoted by <<). This operator just shifts the binary notation of a number to the left by given number of

bits (say k) adding k zeros at the end. Overall this has an effect of multiplying the number by 2^k.

a<<b is same as operating a*2^b and this can be done in constant time. For calculating 2^m, we can just do p ← 1<<m. Now p will have the value 2^m, and p*Initial wealth will be our answer.

Dhruv Gupta (220361)

Question 2. Moody Friends **(40 points)**

P friends arrive at a hotel after a long journey and want rooms for a night. This hotel has n rooms linearly arranged in form of an array from left to right where array values depict the capacities of the rooms. As these are very close friends they will only consider consecutive rooms for staying. As you are the manager of the hotel you are required to find cheapest room allocation possible for them (sum of the capacities of selected rooms should be greater than or equal to P). Cost of booking every room is same and is equal to C.

(a) (15 points) Design an algorithm in O(n) time complexity for determining the minimum cost room allocation. The allocated rooms should be consecutive in the array and their capacities should sum to at least P. You will get 5 points if you design an O(n log n) time algorithm.

(b) (15 points) Now suppose they don't care about the cost and total capacity anymore. But they came up with a beauty criteria for an allocation. According to them, an allocation is beautiful if GCD (Greatest Common Divisor) of capacities of all rooms in the allocation is at least equal to or greater than a constant K. And they want to take maximum number of contiguous rooms possible. Your task is to design an algorithm in O(nlog(n)) time complexity for determining the maximum number of contiguous rooms they can get which satisfy the beauty constraints. You can assume access to a blackbox GCD algorithm which can give you GCD of two numbers in constant O(1) time.

(c) (10 points) Give proof of correctness and time complexity analysis of your approach for part (a).

Solution 2:

(a) We are assuming that such a room allocation is possible, i.e., Sum of all the elements of the array is greater that equal to P, otherwise no room allocation is possible. (This can be checked in O(n) time).

We keep 2 pointers **i** and **j**, both pointing initially at the start of the array. Let Sum[i,j] denote the sum of capacities from **i** to **j**. We maintain a variable **count** to store the minimum number of rooms required. We will also maintain two variables **start** and **end** to store the corresponding starting and ending location of the subarray. We keep incrementing j, until Sum[i,j] becomes greater than **P.** We store the length of this subarray, **j-i+1** in **count**. After this we we will make sure that Sum[i,j] remains greater than P.

We increment i if it can be done, otherwise increment j. Then if j+i-1 < count, set count to j+i-1. Do nothing if j+i-1>count.

Repeat the last step until j reaches the end of the array.

The value stored in count will be our answer.

Pseudo Code:

i ← 0;

```
j ← 0;
start ←0;
end ← 0;
Sum ← 0; // At any point of time, Sum stores the value of Sum[i,j].
While(j <> n)  // n = number of elements in the array A.
{ if (Sum < P) {
        Sum ← Sum + A[j];
        j++;
        continue;
  }
count ←  j+i-1;
start ← i;
end ← j;
  if (Sum – A[i] < P){
        Sum ← Sum + A[j];
        j++;
        if(j-i+1 < count){
        count ← j-i+1;
        start ← i;
        end ← j;
        }
        continue;
  }
 else{
        Sum ← Sum – A[i];
        i++;
        if(j-i+1 < count){
        count ← j-i+1;
        start ← i;
```

end ← j;

}

continue;

}

}

After this loop, **count** stores the minimum value of rooms required and minimum cost then equals **count*C.**

(b) We need to find the maximum length possible for a subarray of A, such that GCD of all elements of A ≥ K.

First we have two observations

1. $GCD(a_1,a_2,a_3) = GCD(GCD(a_1,a_2) , a_3)$
2. $GCD(a_1, a_2,\ldots a_n) \geq GCD(a_1, a_2,\ldots a_i)$ for all $i \leq n$.

Our idea is to binary search type algorithm for the length of the array.

Suppose we have a method to find if there exists a subarray of length m, such that GCD of all its elememts ≥ K. And assume that this can be done in O(n) time.

Check_gcd(A,m) : returns 1 if a subarray with GCD ≥ K is found, and 0 if no such array exists.

//

L←1; R←n;

while(L ≤ R){

mid ← (L+R)/2;

// If a subarray of length mid is not found, then any any length greater than mid is not possible. So we restrict our search from lengths L to mid-1,

// If such subarray is found, then we just have to check for lengths mid+1 to R.

// We will keep largest length possible stored in temp, whenever it is found.

if (Check_gcd(A,mid) = 0) { R ← mid-1;}

else { L ← mid+1; temp← mid}

}

Since this loop is same as that we use in binary search, loop will iterate for log(n) times.

Therefore if we can make such a function Check_gcd, which takes O(n) time, total time complexity will be O(n*log(n)).

For this purpose we will use following algorithm:

For a given length k.

First we will divide the array A into subarrays of lengths n/k.

Then we maintain two Arrays B and C.

We are denoting GCD of subarray from i to j by GCD(i,j).

Array B will store the values of GCD(0,0), GCD(0,1),…GCD(0,k-1), GCD(k,k), GCD(k,k+1),…GCD(k,2*k-1),…GCD(j*k, j*k), GCD(j*k,j*k+1),…GCD(j*k, (j+1)*k-1)… and so on.

Array C will store the values of GCD(k-1,k-1), GCD(k-2,k-1),…GCD(0,k-1),GCD(2*k-1, 2*k-1),GCD(2*k-2,2*k-1),…GCD(k,2*k-1),…GCD((j+1)*k-1, (j+1)*k-1), GCD((j+1)*k-2, (j+1)*k-1),… GCD((j*k, (j+1)*k-1),… and so on.

We can find the GCDs of all subarrays starting at index 0 of an array of m elements easily in O(m) time.

Therefore for each array of length n/k, we will need O(n/k) time. And since there are k such arrays total time to create B = O(n/k) * k = O(n).

Similarly, We can find the GCDs of all subarrays ending at index m-1 of an array of m elements easily in O(m) time.

Therefore for each array of length n/k, we will need O(n/k) time. And since there are k such arrays total time to create C = O(n/k) * k = O(n).

Now consider any i.

Now GCD(i,i+k-1) (subarray of length k) can be calculated in constant time using B and C.

GCD(i,j) = GCD(B[i+k-1], C[i]).

Now if we traverse A for all such i, it will take O(n) time and at any point if we find such i, for which GCD(i,i+k-1) ≥ K then we return 1, and if no such i is found, we return 0.

So we have designed an algorithm for Check_gcd function which works in O(n) time. Hence by above method we can find required k in O(n*log(n)) time.

(c) **Proof of correctness of (a)**

Initially, the the first if statement is evaluated, since Sum is 0 initially. In every iteration, this block keeps on evaluating, unless Sum becomes ≥ P. Suppose this happens at j_0.

Clearly, count evaluated just after j_0th iteration stores the minimum number of rooms required for Sum≥ P, if we allow only $j \leq j_0$. Also the values of start and end at this point correspond to the starting and ending indices of room allocation.

Also note that after j_0 th iteration, Sum ≥ P always. This is because, Sum is decreased only when Sum – A[i] ≥ P, and this becomes the new value of Sum, which is still greater than or equal to P.

P(k) : After j is incremented from k-1 to k (j_0<k≤n), count stores the minimum number of rooms if we only consider the array from index 0 to k.

Assume P(k) is true.

Base case: P(j_0) holds.

In any iteration of while loop, if the **else** block is executed, value of i increases by 1, thereby decreasing count and the Sum. Such iterations continue until count reaches a certain minimum value after which Sum-A[i] becomes less than P, i.e., sum cannot be further decreased, i.e., count can not be decreased.

After this happens, block corresponding to **if(Sum-A[i] <P)** is executed. Now j is increased by 1, hence the total number of rooms increase, but requirement is still met by the earlier value of count, which remains unchanged, Thus after this iteration j points at k+1, count stores the minimum number of rooms required if we take only the elements from index 0 to k+1.

Hence by induction P(k) is true for all k (j_0<k≤n). The while loop ends when k = n, i.e., j points to the end of the array. So P(n) is true implies the correctness of our algorithm.

Also since we update the start and end values accordingly every time count changes, A[start,end] gives us the corresponding room allocation.

Time Complexity:

All the comparisons and assignment operations take constant time. Therefore, each iteration of while loop takes constant *c* or O(1) time.

Loop ends when j =n. Also at any point of time, 0 ≤ i ≤ j ≤ n. In each iteration, either i is incremented by 1 or j is incremented by 1. Since their maximum value can be n,

Maximum number of iterations possible = 2*n.

So, total time taken ≤ *c* * 2*n = O(n).

Hence, time complexity is **O(n)**.

Question 3. BST universe **(30 points)**

You live in a BST world where people are crazy about collecting BSTs and trading them for high values. You also love Binary Search Trees and possess a BST. The number of nodes in your BST is n.

(a) (10 points) The Rival group broke into your lab to steal your BST but you were able to stop them. But still they managed to swap exactly two of the vertices in your BST. Design an O(n) algorithm to find which nodes are swapped and the list of their common ancestors.

(b) (20 points) Seeing you were able to easily revert the damage to your tree, they attacked again and this time managed to rearrange exactly k of your nodes in such a way that none of the k nodes remain at the same position after the rearrangement. Also all the values inside this BST are positive integers and upper bounded by a constant G. Your task is to determine the value of k and which nodes were rearranged. Design an algorithm of complexity O(min(G + n, nlog(n))) for the same. (Hint : Consider two cases for G < nlog(n) and G > nlog(n))

Solution 3:

(a) We can use the **Traversal Algorithm** (as discussed in Lecture slides) to traverse the whole BST in O(n) time. But instead of printing the numbers, we will store them in an array A of size n. If the BST was not tampered, we would have received a sorted array with elements in ascending order.

```
Traversal(T){
  p←T;
  if(p=NULL) return;
  else{ if(left(p <> NULL) Traversal(left(p));
        A[i] ← value(p);
        if(right(p)<>NULL) Traversal(right(p))0;
       }
}
```

But since two of its nodes were swapped, there would be exactly two elements in the array A which are swapped.

Let array A = {a_0, a_1, a_2, ……a_{n-1}}. Then we will have two elements a_p and a_q (p<q)such that a_p is larger than its neighbours and a_q is smaller than its neighbours. We need to find these a_p and a_q.

This can be done in following way:

```
if(A[0] > A[1]) { p←A[0] }
else{
  for(i=1 to n-2){
    if(A[i] > A[i-1] & A[i] > A[i+1]) { p←A[i]; break; }
  }
}
```

```
if(A[n-1] < A[n-2]) { q←A[0] }
else{
  for(i=1 to n-2){
    if(A[i] < A[i-1] & A[i] < A[i+1]) { q←A[i]; break; }
  }
}
```

//Now p and q have the two values which were swapped.

// for common ancestors we will maintain an array parents[] which will store the value of the common ancestors.
// count will have the number of common ancestors.

```
count ← 0, k ←0;
parents[n]; // Array declaration

Find_parents(T, p, q){
  t←T;
  if(t<>NULL){
    if( p = val(t) or q = val(t)) { return count; }
    else if( p<val(t) & q<val(t) ) {
      count++;
      parents[k] ← val(t); k++;
      Find_parents(left(t), p, q);
     }
     else if( p>val(t) & q>val(t) ) {
       count++;
       parents[k] ← val(t); k++;
       Find_parents(right(t), p, q);
     }
     else{ count++; return count; }
  }
}
```

(b) Again, we first use **Traversal** to create the array A. If k nodes are swapped, their would be exactly k nodes in the array A formed after traversal which are not in there correct position. By correct position we are referring to the position where the number will land after sorting the array. So if we create an array B which is a copy of A, and then sort B, then we can just compare the two arrays and if for any i, if A[i] ≠ B[i], then A[i] was moved, and the number of such i will be the value of k.
For creating the array B we will discuss two algorithms.
Algo 1: Create a copy of A, and sort it. This will be our array B.

```
for(i=0 to n-1) {B[i] ← A[i]}
sort(B);
```

Algo 2: Since G is an upper bound on all numbers in A, if we create an array of C of size G+1, and index it from 0 to G, we are sure that each number in A is equal to an index in C.

for(i=0 to G) {C[i] ← -1} //We initialize all values of C to -1.

for(i=0 to n-1) {C[A[i]] ← A[i]} // We place A[i] at A[i]-th position in C

//Now C[x] = x, if x is an element of A, and -1 if x is not an element of A.

//Also, note that all the non-negative elements of C are elements of A and all of them are in ascending order.

//So now we copy all the non-negative elements from C to B in order. B will be our required array.

```
j←0;
for(i=0 to G){
  if(C[i] ≥ 0) { B[j] = C[i]; j++}
}
```

//Now compare A and B and store the 'incorrectly' positioned numbers in an array D.

```
j←0;
count←0;
for(i=0 to n-1){
  if(A[i] = B[i]) { D[j] ← A[i]; j++; count++;
}
```

//All the required numbers are listed in array D, and their number k = count.

Time Complexity Analysis:

Traversal step takes a*n +b (=O(n)) time since each node is visited exactly once.

Algo 1 will take c*n + d*n*log(n) time.

Algo 2 will take c*G+ c*n+ f*G = c*n + p*G time.

(Here we have used the same constant c since the loops are performing exactly the same operations in one iteration)

Compare step will take q*n time.

Total time with Algo 1 = c_1n + d *n*log(n) = O(n*log(n) + n)

Total time with Algo 2 = c_2n + p*G = O(G+n)

Now if G<n*log(n), Algo 2 will work faster, and if G>n*log(n), Algo 1 will work faster. So we can do this comparison in the start and then decide which algorithm to use. Hence overall time complexity = O(min(G+n, n*logn)).

Question 4. Helping Joker **(20 points)**

Joker was challenged by his master to solve a puzzle. His master showed him a deck of n cards. Each card has value written on it. Master announced that all the cards are indexed from 1 to n from top to bottom such that (a1 < a2 < ... < an−1 < an) .Then his master performed an operation on this deck invisible to Joker (Joker was not able to see what he did), he picked a random number k between 0 and n and shifted the top k cards to the bottom of the deck. So after the operation arrangement of cards from top to bottom looks like (a_{k+1}, a_{k+2} . . . a_n, a_1, a_2 . . . a_k) where (k + 1, k + 2 . . . n, 1, 2 . . . k) are original indices in the sorted deck. Joker's task is to determine the value of k. Joker can make a query to his master. In a query, joker can ask to look at the value of any card in the deck. Joker asked you for help because he knew you were taking an algorithms course this semester.

(a) (15 points) Design an algorithm of complexity O(log(n)) for Joker to find the value of k.

(b) (5 points) Provide time complexity analysis for your strategy

Solution 4:

(a) Let the final array be B[n] = (a_{k+1}, a_{k+2} . . . a_n, a_1, a_2 . . . a_k) = (b_0, b_1 . . . b_{n-1}).
Note that in the final array, index/position of a_k is n-1. Let index of a_1 in final array be m. From a_1 to a_k (both included) we have total k elements which we can write in terms of m and n as k = (n-1)-m+1 = n-m. We know the value of n. So if we can find m, we can find k. So our approach will be to find the index of a_1 in the final array.

Pseudo Code (with comments) :

```
// Find_a1 will return the index of a1.
Find_a1(B,n){
  // We will keep two indices i and j and our search will be limited to elements between i and j.
   i←0; //initialize i to 0
   j ←n-1; // initialize j to n-1.
// Since a(k+1) > ak for every k=1,2…n-1, hence B[0]>B[n-1] for k = 1,2…n-1.
//Only in case when k = 0 or k = n, B will be same as initial array, i.e., no change. In that case a1 is at index 0. This will happen when B[0]<B[n-1].
   if(B[0]<B[n-1]) return 0;
flag←0;

// We set the flag to 0 initially, and when we will find a1 we will set it to 1. The loop must go on until we have not found a1, i.e., until flag is 0.
   while(flag = 0){
      mid ← (i+j)/2;
//Check if the middle element of B[i,j] is a1. We use the fact that a1 is the only element which is smaller than both its neighbours.
      if(B[mid] < B[mid+1] & B[mid] < B[mid-1]){
         flag←1;
```

```
        return mid;
      }
```

//Compare the value of middle element with the last element. Since all values before a_1 are larger than last element, and all values after it are smaller than last element, if value of middle element is larger, a_1 lies to the right of it and if it is smaller a_1 lies to the left of it. Therefore we update the values of i and j accordingly and restrict the searching length. Note that the searching length is actually halved.
//The loop will end when B[mid] becomes a_1

```
      else if(B[mid] > B[n-1]){
        i ← mid+1;
      }
      else{
        j ← mid-1;
      }
    }
  }
```

Our final answer k = n – Find(B,n).

(b) Let the time taken for array of length n be T(n). After every iteration, we are halving the length of the array. All the comparisons and assignment operations take some constant time (O(1) time). We can therefore write the time taken T(n) recursively as T(n) = T(n/2) + c (Total time taken = some constant + time taken to search half the array), and T(1) = c' (constant).
T(n) = c + T(n/2) = c + c + T(n/4) = c + c + c + T(n/8) = . . . = c + c + c +….($\log_2 n$ times) + T(1) = c*$\log_2 n$ + c'
Therefore, T(n) = c*$\log_2 n$ + c' = **O(log(n))**.

Question 5. One Piece Treasure **(30 points)**

Strawhat Luffy and his crew got lost while searching for One piece (worlds largest known treasure). It turns out he is trapped by his rival Blackbeard. In order to get out he just needs to solve a simple problem. You being the smartest on his crew are summoned to help. On the gate out, you get to know about a hidden string of lowercase english alphabets of length n. Also, an oracle is provided which accepts an input query of format (i, j), and returns true if the substring(i, j) of the hidden string is a palindrome and false otherwise in O(1) time. But there is a catch, the place will collapse killing all the crew members, if you ask any more than $n * \log^2(n)$ queries to the oracle. The string is hidden and you can't access it. (Assume the string is very big i.e. n is a large number)

(a) (30 points) You need to design a strategy to find number of palindromic substrings in the hidden string so your crew can safely escape from this region. Please state your algorithm clearly with pseudocode. (A contiguous portion of the string is called a substring)

Solution 5:

We will assume a function named **isPalindrome(i,j)** which will return **1** if substring(i,j) is a palindrome and return **0** is it is not a palindrome.

First we will count the number of palindromes of odd length.

We will use the fact that if a_k,a_(k+1),…..a_(m-1),a_m, is a palindrome, then obviously, a_(k+1),…..a_(m-1) is also a palindrome, and so on. Similarly if a_k,….a_m is not a palindrome, then a_(k-1), a_k,…….a_m,a_(m+1) is also not a palindrome.

Pseudo Code with Algorithm (in comments):

// Count_at_index function returns the number of palindromes centred at index i.

Count_at_index_odd(i,n){

// L and R correspond to minimum and maximum 'half-length' of the substring.

// half-length means the number of elements on one side of the centre.

// We will be finding maximum possible value of half-length (**ans**) for which a palindrome exist at centre i. And total number of palindromes with centre I will then be = half-length.

L ← 1;

R ← min(i, n-i-1); // We are assuming a function **min(a,b)** which returns minimum of a and b in O(1) time.

if(not isPalindrome(i-L, i+L)) {return 0;}

// half-length=1, is the minimum length of palindrome possible, if that is not present, no palindrome exist with centre i. so **ans** = 0;

else if(isPalindrome(i-R,i+R) {return R;}

// half-length=R, is the largest possible palindrome, if that palindrome is present all smaller palindromes also exist at centre i. So **ans** = R;

else{

//In conditions when **L ≤ ans < R**, we use a binary search type technique, and keep reducing our search area into half. Initially search area is **(L, R)**.

ans ← L;

while(L ≤ R){ // loop ends when L becomes greater than R, i.e., we have exhausted the search area and search is complete.

mid ← (L+R)/2;

ans ← mid; // After the while loop ends the last value of **mid** will be our **ans**.

// If palindrome is not found at half-length = mid, We restrict our search to **(mid+1, R)**

if(isPalindrome(i-mid,i+mid)) { L← mid+1;}

// If palindrome is not found at half-length = mid, We restrict our search to **(L, mid-1)**

else { R←mid-1;}

}

return ans;

}

}

//Now we will iterate i over the whole string and for each i count the number of palindromes with centre at i

count_odd ← 0;

for(i=1 to n-2){ // Clearly no palindrome can have i=0 or i=n-1 as centre.

count_odd ← count_odd + Count_at_index_odd(i,n);

}

//**count_odd** now stores the total number of palindromes with odd length.

We will use similar approach for even length. Only difference will be that now there will be two numbers for centre. We can take any one of them as centre. Let us take the smaller index to be the centre. Also we will now define half-length as number of elements to the left of centre. Number of elements to the right of centre will be half-length +1. Also, since 2

length palindromes are also possible, half-length =0 is also allowed, so in this case our answer would be half-length+1, instead of half-length.

Count_at_index_even(i,n){

// Complete algorithm works the same as **Count_at_index_odd(i,n)**, only difference is in half- length, initial values of L,R, and **ans** will be half-length +1.

L ← 0;

R ← min(i, n-i-2);

if(not isPalindrome(i-L, i+L+1)) {return 0;} // No palindrome possible

else if(isPalindrome(i-R,i+R+1) {return R+1;}//Every substring with i as centre is palindrome.

else{

//In conditions when **L ≤ ans < R**, we use a binary search type technique, and keep reducing our search area into half. Initially search area is **(L, R)**.

ans ← L+1;

while(L ≤ R){ // loop ends when L becomes greater than R, i.e., we have exhausted the search area and search is complete.

mid ← (L+R)/2;

ans ← mid+1; // After the while loop ends the last value of **mid+1** will be our **ans**.

// If palindrome is not found at half-length = mid, We restrict our search to **(mid+1, R)**

if(isPalindrome(i-mid,i+mid+1)) { L← mid+1;}

// If palindrome is not found at half-length = mid, We restrict our search to **(L, mid-1)**

else { R←mid-1;}

}

return ans;

}

}

//Now we will iterate i over the whole string and for each i count the number of palindromes with centre at i

count_even ← 0;

for(i=0 to n-2){ // This time palindrome with i=0 as centre is possible.

count_even ← count_even + Count_at_index_even(i,n);

}

//**count_even** now stores the total number of palindromes with even length.

final_ans ← count_odd + count_even

Time Complexity Analysis:

Let each call of Count_at_index_odd(i,n) takes T(n) time.

Clearly $T(n) \leq a + F(n)$ (= Time taken for while loop), where a is some constant.

No since after each iteration, our search length is reduced by half,

$F(n) \leq b + F(n/2)$ i.e., some constant time b for one iteration and F(n/2) after that.

Using method of unfolding,

$F(n) \leq b + b + F(n/4) \leq b+b+b+\ldots(\log_2(n) \text{ times}) + F(1) \leq \text{some constant} + b*\log_2(n) = O(\log(n))$

Therefore, $T(n) \leq$ (some constant) $c1 + b*\log_2(n) = O(\log(n))$

Hence each functional call will take O(log(n)) time. Same will be for Count_at_index_even(i,n).

Now in first for loop, the function is called n-2 times, therefore, time taken = O(n*log(n))

Similarly in second loop, function is called n-1 times, therefore, time taken = O(n*log(n))

So, total time = $O(n*\log(n)) + O(n*\log(n)) = O(n*\log(n)) \leq O(n*\log^2(n))$.