CS2800 - Programming Assignment № 2

CS18B006, CS18B057, CS18B062 09-02-20

Question 1

Binary GCD algorithm: Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the binary gcd algorithm, which avoids the remainder computations used in Euclid's algorithm.
(a) Prove that if a and b are both even, then gcd(a, b) = 2.gcd$(a/2, b/2)$.
(b) Prove that if a is odd and b is even, then gcd(a, b) = gcd$(a, b/2)$.
(c) Prove that if a and b are both odd, then gcd(a, b) = gcd$((a - b)/2, b)$
(d) Design an efficient binary gcd algorithm for input integers a and b, where a $\geq$ b, that runs in $O(lga)$ time. Assume that each subtraction, parity test, and halving takes unit time. [lg a is the same as $\log_2 a$]

Solution - 1

(a) If a and b are both even, then we can write them as a = 2(a/2) and b = 2(b/2) where both factors in each are integers. This means that gcd(a,b) = 2 gcd(a/2,b/2).
(b) If a is odd, and b is even, then we can write b = 2(b/2), where b/2 is an integer, so, since we know that 2 does not divide a, the factor of two that is in b cannot be part of the gcd of the two numbers. This means that we have gcd(a,b) = gcd(a,b/2).
(c) If a and b are both odd, then, first, we show that gcd(a,b) = gcd(a-b,b). Let d and d be the gcd's on the left and right respectively. Then, we have that there exists n_1, n_2 so that $n_1a + n_2b = d$,but then, we can rewrite to get $n_1(a - b) + (n_1 + n_2)b = d$. This gets us d≥d. To see the reverse, let n'_1, n'_2 so that $n'_1(a - b) + n'_2b = d$.We rewrite to get $n'_1a + (n'_2 - n'_1)b = d'$, so we have d'≥d. This means that gcd(a,b) = gcd(a-b,b) = gcd(b,a-b). From there, we fall into the case of part b. This is because the first argument is odd, and the second is the difference of two odd numbers, hence is even. This means we can halve the second argument without changing the quantity. So, gcd(a,b) =gcd(b,(a-b)/2) = gcd((a-b)/2,b).
(d) Consider the pseudcode for the binary gcd algorithm algorithm:

```
BinaryGCD(int a, int b){
    if(a%2==1){
        if(b%2==1)
            return BinaryGCD((a-b)/2, b);
        else
```

```
            return binaryGCD(a, b/2);
    }
    else{
        if(b%2==1)
            return BinaryGCD(a/2, b)
        else
            return BinaryGCD(a/2, b/2);
    }
}
```

In each iteration, we are either reducing the number of bits of a or b by atleast 1. So the maximum number of iterations is the sum of number of bits in a and b i.e. $\lceil lga \rceil + \lceil lgb \rceil$. As a $>$ b, we get the complexity of the algorithm as $O(lga)$.

Question 2

Analysis of bit operations in Euclid's algorithm.

(a) Consider the ordinary "paper and pencil" algorithm for long division: dividing a by b, which yields a quotient q and remainder r. Show that this method requires $O((1+\lg q)\lg b)$ bit operations.

(b) Define $\mu(a, b)$ = (1 + lg a)(1 + lg b). Show that the number of bit operations performed by Euclid(a, b) in reducing the problem of computing gcd(a, b) to that of computing gcd(b, a mod b) is at most c($\mu(a, b)$ - $\mu(b, a \bmod b)$) for some sufficiently large constant c > 0.

(c) Show that Euclid(a, b) requires O($\mu(a, b)$) bit operations in general and O(β^2) bit operations when applied to two β-bit inputs.

(d) If a > b ≥ 0, show that the call Euclid(a, b) makes at most $1 + \log_\phi b$ recursive calls. Improve this bound to $1 + \log_\phi(b/gcd(a, b))$.

Solution - 2

(a) We are given two binary numbers a and b. If we perform the ordinary "paper and pencil" algorithm, then the number of times we do comparisons for subtraction is the same as the number of digits in q, i.e $\lceil \lg q \rceil$. The subtraction in itself will take as many comparisons as the number of digits in b, that is $\lceil \lg b \rceil$. At the last step in the long division, we check if the remainder is less than the divisor b. This takes another $\lceil \lg b \rceil$ comparisons.
Therefore, the total number of bit operations are $\lceil \lg b \rceil + \lceil \lg b \rceil \times \lceil \lg q \rceil$, that is $O((1+\lg q)\lg b)$ bit operations.

(b) The reduction requires us to compute $a \bmod b$. By carrying out ordinary "paper and pencil" long division we can compute the remainder. The time to do this, by part (a), is bounded by $k(1+\lg q)(\lg b)$ for some constant k. In the worst case, q has ($\lg a - \lg b$) bits. Thus, we get the bound $k(1+\lg a - \lg b)(\lg b)$. Next, we compute $\mu(a, b) - \mu(b, a \bmod b) = (1+\lg a)(1+\lg b) - (1+\lg b)(1+\lg (a \bmod b))$. This is smallest when $a \bmod b$ is small, so we have a lower bound of $(1+\lg a)(1+\lg b) - (1+\lg b)$. Assuming $\lg b \geq 1$, we can take $c = k$ to obtain the desired inequality.

(c) As shown in part (b), Euclid takes at most c($\mu(a, b)$ - $\mu(b, a \bmod b)$) operations in the first recursive call, at most c($\mu(b, a \bmod b)$ - $\mu(a \bmod b, b \bmod (a \bmod b))$) operations on the second recursive call, and so on.
Summing over all recursive calls gives a telescoping series, resulting in $c\mu(a, b) + O(1) = O(\mu(a, b))$.
When applied over two β-bit inputs, the runtime is $O(\mu(a, b)) = O((1 + \beta)(1 + \beta)) = O(\beta^2)$.

(d) Our analysis makes use of Fibonacci numbers F_k.
Theorem: For any integer $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, then the call Euclid(a, b) makes fewer than k recursive calls.
Proof: We can show that the upper bound of theorem is the best possible by showing that the call Euclid(F_{k+1}, F_k) makes exactly $k - 1$ recursive calls when $k \geq 2$. We use induction on k. For the base case, k = 2, and the call Euclid(F_3, F_2) makes exactly one recursive call, to Euclid(1, 0). (We

have to start at k = 2, because when k = 1 we do not have $F_2 > F_1$.) For the inductive step, assume that Euclid(F_k, F_{k-1}) makes exactly k - 2 recursive calls. For k > 2, we have $F_k > F_{k-1} > 0$ and $F_{k+1} = F_k + F_{k-1}$, and we have F_{k+1} mod $F_k = F_{k-1}$. Thus, we have
gcd(F_{k+1}, F_k) =gcd(F_k, F_{k-1} mod F_k) = gcd(F_k, F_{k-1})
Therefore, the call Euclid(F_{k+1}, F_k) recurses one time more than the call Euclid(F_k, F_{k-1}), or exactly k - 1 times, meeting the upper bound of theorem.

Since F_k is less than $\phi^k/\sqrt{5}$, we get that for all k, if $b < F_{k+1} < \phi^{k+1}/\sqrt{5}$, then it takes fewer than k steps. If we let $k = \log_\phi b + 1$, then, since $b < \phi^{\log_\phi b+2}/\sqrt{5} = \frac{\phi^2}{\sqrt{5}}.b$, we have that it only takes $1 + \log_\phi(b)$ steps.
We can improve this bound to $1 + log_\phi(b/\text{gcd}(a, b))$. This is because we know that the algorithm will terminate when it reaches gcd(a, b). We will show a different claim that Euclid's algorithm takes k recursive calls, then $a \geq$ gcd$(a, b)F_{k+2}$ and $b \geq$ gcd$(a, b)F_{k+1}$. We will do induction on k. If it takes one recursive call, and we have $a > b$, we have $a \geq 2$gcd(a, b) and $b =$ gcd(a, b).
Now, suppose it holds for $k - 1$, we want to show it holds for k. The first call that is made is of Euclid(b, a mod b). Since this then only needs $k - 1$ recursive calls, we can apply the inductive hypothesis to get that $b \geq$ gcd$(a, b)F_{k+1}$ and a mod $b \geq$ gcd$(a, b)F_k$. Since we had that $a > b$, we have that $a \geq b + (a \text{ mod } b) \geq$ gcd$(a, b)(F_{k+1} + F_k) =$ gcd$(a, b)F_{k+2}$ completing the induction.
Since we have that we only need k steps so long as $b <$ gcd$(a, b)F_{k+1} <$ gcd$(a, b)\phi^{k+1}$. We have that $\log_\phi(b/\text{gcd}(a, b)) < k + 1$. This is satisfied if we set $k = 1 + \log_\phi(b/\text{gcd}(a, b)$. Thus we have improved the bound to $1 + \log_\phi(b/\text{gcd}(a, b)$.

(Reference- CLRS, and https://sites.math.rutgers.edu/ ajl213/CLRS/Ch31.pdf)

Question 3

Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values A[1], A[2], . . . , A[n] is unimodal: For some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry A[j] on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum, and then fall from there on.) You'd like to find the "peak entry" p without having to read the entire array-in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most O(log n) entries of A.

Solution - 3

Listing 1: Code

```
#include <bits/stdc++.h>
using namespace std;

int get_index(int* arr, int n){
    int low = 0;
    int high = n;
    int mid;
    while(true){
        mid = int((low+high)/2);
        if(arr[mid]>arr[mid-1] && arr[mid]>arr[mid+1]){
            return mid;
        }
        else{
            if(arr[mid-1]<arr[mid] && arr[mid]<arr[mid+1]){
                low = mid;      // Peak is to the right of mid.
            }
            if(arr[mid-1]>arr[mid] && arr[mid]>arr[mid+1]){
                high = mid;     // Peak is to the left of mid.
            }
        }
        if(mid == 0 || mid == n){
            return mid;         // Peak is one of the corner elements.
        }
    }
}

int main() {
    int n;
    int x;
    cin>>n;
    int arr[n+1]={0};        // Initializing the array to 0
    for(int i=0;i<n;i++){
        cin>>x;
        arr[i]=x;            // Populating the array.
    }
    cout<<get_index(arr,n)<<endl;
    return 0;
}
```

Correctness and Explanation:-

One naive approach would be to iterate through the array and check for which index is that index the local maxima. But this way we would achieve our goal in **O(n)** operations which is not desired. A better and efficient way would be to use an algorithm similar to binary search. We maintain low, high as the indices for which the middle element is checked wheather it is the peak in the question. If it is the peak then we return that value. If it is not then we need to check if that mid element is on the increasing part of the graph(Slope = +ve) or the decreasing part(Slope = -ve). If we know that mid was on the increasing part of the graph then the peak **must** be on the right side of the mid, while if mid was on the decreasing part of the graph then the peak **must** be on the left side of the mid and therfore we update low or high accordingly and check for mid and do this all over again until we find the peak. If mid becomes '0' or 'n' it means that the graph is either monotonically increasing or monotonically decreasing and we will return the appropriate start or end index.

Running Time Analysis:-

Each time we run the loop we half the area of the array we want to check narrowing down the places the peak can be. Therefore the complexity is exactly that of binary search and we get **O(logn)** complexity by this approach.

Question 4

Consider an n-node complete binary tree T, where n = 2^d - 1 for some d. Each node v of T is labelled with a real number x_v. You may assume that the real numbers labelling the nodes are all distinct. A node v of T is a local minimum if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. You are given such a complete binary tree T, but the labelling is only specified in the following implicit way: for each node v, you can determine the value x_v by probing the node v. Show how to find a local minimum of T using only O(log n) probes to the nodes of T.

Listing 2: Code

```
int FindLocalMin(Node *p){
    if(p->left==NULL && p->right==NULL)                   //Leaf
        return p->data;
    if(p->data<p->left->data && p->data<p->right->data)   //Root is Min
        return p->data;
    if(p->left->data < p->right->data)
        return FindLocalMin(p->left);
    return FindLocalMin(p->right);
}
```

Correctness and Explanation:-

We start at the root of the Complete Tree. If both the children nodes are larger than the root then the root itself is a local minimum, hence it is the answer. If any of the children are smaller than the root then we recursively call the same function in the sub-tree of the child which is smaller as compared to the other child.

We have to acknowledge the fact that when we call the function in a sub-tree its root is smaller that its parent, which is why function was called on it. Therefore we have to compare the current node only with its children and the parent doesn't need to be checked again.

Now we have to argue that this will terminate with a return value. To show that let us assume that we cannot find such node, which implies that each time the root was not the local minimum otherwise it would have been returned as answer. This implies that even when the node at the second last level was the root it was not a local minimum, hence at least one of its children was smaller than it, but this implies that that child itself is the local minimum. Hence we prove the termination of the program with a legitimate return value.

Running Time Analysis:-

As we call the function only once at each level of the complete binary tree and perform constant number of operations, the time complexity can easily be shown as O(Height of Tree) = O(log(n)).

Question 5

Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i - k| + |j - l| = 1$.
We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Solution - 5

We use a divide and conquer algorithm to reduce the size to a quadrant of the matrix recursively as follows:
Step 1: We look at all the elements at the boundary of the matrix, and in the center row and column,(that is, in a window) and find the minimum among them. If that element is a minima, we've found our answer.
Step 2: If it is not a minima, then we find it's smallest neighbor. We know that that neighbor cannot be in the elements already checked. We now recurse in that quadrant which contains the neighbor, also including the boundary which we have already checked.
The pseudocode for the function that does this is given below:

Listing 3: Code

```
int findMinima( int M[n][], n){
    (i, j) = minInWindow(M, n);  //finds the minimum element in the ↩
       window
    if ((i, j).isMinima()) return (i, j);  //returns the location of ↩
       the minima
    else{
        (p,q) = findMinNeighbor(i, j);  //finds minimum neighbor
        quad(m1, n1) = getQuadrant(p, q, n); //finds which quadrant ↩
           the min neighbor is in by comparisons with (n/2)
        return findMinima(M[m1][], n1);   //recursive call to that ↩
           quadrant
    }
}
```

Correctness and Explanation:-

Lemma: If you enter a quadrant, it contains a minima.
This can easily be verified using recursion on the size of matrix. Base case with n=3, and n=4 is viewed easily. Our induction hypothesis will be that this holds for all matrix sizes till $k \times k$. In our

induction step, we show that it also holds for a matrix of size $(k+1) \times (k+1)$.

Invariant: Minimum element of window never decreases as we descend in recursion.

As we also include the boundary in the quadrant we recurse into, the minimum element of the window never changes.

Minima in visited quadrant is also minima in overall matrix. In this way, we find a minima in the matrix.

Running Time Analysis:-

We reduce an $n \times n$ matrix into approximately $\frac{n}{2} \times \frac{n}{2}$ in $O(n)$ time(as we check the boundaries and middle row and column). Therefore

$T(n) = T(\frac{n}{2}) + cn$

$T(n) = T(\frac{n}{4}) + c.\frac{n}{2} + c.n$

...

$T(n) = T(1) + c(1 + 2 + ... + \frac{n}{2} + n)$

Therefore, $T(n) = O(n)$. Thus we have found a local minimum in a matrix using only $O(n)$ probes.

(Reference- MIT course on algorithms)

Question 6

Majority Element.

(a) Let T[1 . . . n] be an array of n elements. An element x is said to be a majority element in T if |i | T[i] = x| > n/2. Give an algorithm that can decide whether an array T[1 . . . n] includes a majority element (it cannot have more than one), and if so find it. Your algorithm must run in linear time.

(b) Rework the above problem with the supplementary constraint that the only comparisons allowed between elements are tests of equality. You may therefore not assume that an order relation exists between the elements.

Solution - 6(a)

Listing 4: Code

```
#include <bits/stdc++.h>
using namespace std;
int get_majority(int* arr,int n){
    // Store the count of elements in the array using map,
    map<int,int>count;
    for (int i = 0; i < n; i++)
    // The first element is the value of arr[i] and the second is the ↩
       count of that element in the array.
        count[arr[i]]++;
    int ans = 0, res = -1;
    // Find the maximum of count of all the elements.
    for (auto i : count) {
        if (ans < i.second) {
            // Store the majority element in res.
            res = i.first;
            // Store the value of count of that element in ans.
            ans = i.second;
        }
    }
     // If the maximal count is <= n/2 then there is no majority ↩
        element.
    if(ans > n/2){
        return res;
    }
    // Return -1 if there is no majority element.
    return -1;
}
int main() {
    int n;
    int x;
    cin>>n;
    int arr[n+1];
    for(int i=0;i<n;i++){
        cin>>x;                         // Take input
        arr[i]=x;
    }
    cout<<get_majority(arr,n)<<endl;
    return 0;
}
```

Correctness and Explanation:-

The most efficient way to do this is to use hashing.We create a hash table and store elements and their frequency counts as key value pairs. Finally we traverse the hash table and check if the element with the maximum count is indeed the majority element by comparing the count with n/2. If our max count is < n/2 then there is no majority element and return -1. If the count is >n/2 then we return that element value.

Running Time Analysis:-

We travel through the array once, incrementing the count of that array element by 1 each time we encounter that value. Finally we go through the map again checking for the value with the highest count. So we have 2 O(n) operations thereby making it O(n) in time complexity. Also we are using O(n) space complexity.

Solution - 6(b)

Refer to the solution of Q.No - 13.

Question 7

Suppose S = $\{1, 2, .., n\}$ and f : S $\to$ S. If R $\subset$ S, define f(R) = $\{f(x) \mid x \in R\}$. Device an $O(n)$ algorithm for determining the largest R $\subset$ S, such that $f(R) = R$.

Solution - 7

Listing 5: Code

```
// Function that adds a Cycle in a Vector of Vectors
void AddtoAnsVector(int i){
    vector <int > LocalAnsVector;
    int localcount=1;
    LocalAnsVector.push_back(i);
    temp=f[i];
    while(temp!=i){
        LocalAnsVector.push_back(temp);
        localcount++;
    }
    AnsVectorofVector.push_back(LocalAnsVector);
}

// Function that finds the Cycle in the DFS from that vertex
void FindCycleinConnectedComp(int i,int c){
    visited[i]=c;
    if(visited[f[i]]!=-1){
        if(visited[f[i]]==c){
            AddtoAnsVector(f[i]);
        }
    }
    else{
        FindCycleinConnectedComp(f[i],c);
    }
}

// In Main
int i;
for(i=0;i<IncomingDegZero.size();i++){
    FindCycleinConnectedComp(IncomingDegZero[i],i);
}
for(int j=0;j<SizeofSet;j++){
    if(visited[j]==-1){
        FindCycleinConnectedComp(j,i);
        i++;
    }
}
if(IncomingDegZero.size()){
    for(int x=0;x<AnsVectorofVector.size();x++){
        for(int y=0;y<AnsVectorofVector[x].size();y++){
            cout<<AnsVectorofVector[x][y]<<" ";
        }
    }
    cout<<"\n";
}
else {
    if(AnsVectorofVector.size()==1){
        cout<<"No elements in R\n";
    }
    else{
        int flag=FindMinVectorIndexinAnsVector();
        for(int x=0;x<AnsVectorofVector.size();x++){
            if(x!=flag){
                for(int y=0;y<AnsVectorofVector[x].size();y++){
                    cout<<AnsVectorofVector[x][y]<<" ";
                }
            }
        }
        cout<<"\n";
    }
}
```

Correctness and Explanation:-

We assume the function in the form of a graph, with n vertices and n edges.
First we have IncomingDegZero a vector of the vertices which have incoming degree=0. We call DFS on all these vertices using the FindCycleinConnectedComp function, with a flag value (i). As soon as a vertex is encountered which has been previously visited and marked with the same flag value in the Visited array, we find a loop. Then we call AddtoAnsVector function which traverses that loop again, putting each of the cycle vertices into a vector and pushing it into a vector of vectors. But if in the DFS a vertex is encountered which has different flag value implies that that vertex has already been traversed and we do not need to traverse it again. There might be the case that some vertices have still not been traversed, which implies that they are part of cycles with no branching, i.e each vertex has incoming deg = 1 = outgoing deg. Therefore we also traverse on vertices which have still not been visited. These cycles are also then added in the global answer Vector of vectors.
Now traversing of vertices is over.
We now check if there were vertices which had incoming deg = 0, as these would not be a part of the cycles and then we can add all the cycles to the set R, which will then become a proper subset of S. In-case we have no vertices with incoming deg = 0, then we have to not add the cycle with the lowest size, therefore we first check whether we have more than 1 cycle or not. If we have only one cycle then R is Null set, as we would have to include all vertices or none. In-case we have more than 1

cycle we just don't add the cycle with the smallest size.

Arguing the correctness, we know that if f(R) = R, then it implies that members of R are discrete cycle vertices, as we are given with only n directed edges and n vertices, which implies that each connected component has a cycle Therefore our approach to calculate the cycles in the graph is appropriate and correct.

Running Time Analysis:-

We apply DFS on the graph on each of its connected components, which is O(V+E). And twice on the cycles, but still the coomplexity remains O(V+E), and given the knowledge that E = V, each vertex has only one edge out of it, we have complexity as O(n).

Question 8

You are a contestant on the hit game show "Beat Your Neighbours!" You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbours in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni$^{\text{TM}}$, to which you are hopelessly addicted.
(a) Suppose m = 1. Describe an algorithm that finds a number that is bigger than any of its neighbours. How many boxes does your algorithm open in the worst case?
(b) Suppose m = n. Describe an algorithm that finds a number that is bigger than any of its neighbours. How many boxes does your algorithm open in the worst case?
(c) Prove that your solution to part (b) is optimal up to a constant factor.

Solution - 8

(a) When m=1, we have an array with n elements and we have to find a local maxima. To do this, consider the pseudocode:

```
maxima(A, start, end){
    m=(start+end)/2;
    if (A[m]>A[m-1] && A[m]>a[m+1])
        return m;  //m is location of maxima
    else if(A[m-1] > A[m])  //there exists a peak before m
        return maxima(A, start, m-1);
    else if(A[m+1]>A[m])   //there exists a peak after m
        return maxima(A, m+1, end);
}
```

We check the middle element to reduce the problem into half its size. We chose that direction to consider where the elements are locally rising. If the numbers are locally rising in one direction, then there is a maxima in that sub-array.
We check the $\lfloor \frac{n}{2} \rfloor^{\text{th}}$ element in each iteration. Therefore, the maximum number of times we open a box is $3\lceil \log n \rceil$ (worst case for $T(n) = T(\frac{n}{2}) + 3$)
(b) When m=n, we have to find the local maxima in a $n \times n$ matrix. Refer to Question 5 for this, by changing minima with maxima.
When the size of the matrix to be considered is n, we open $(6n - 9 + 2) = (6n - 7)$ boxes in the worst case at each iteration.Therefore total boxes opened are $(6n - 9\lceil \log n \rceil)$ boxes in worst case.

Question 9

Suppose we are given two sorted arrays A[1 . . . n] and B[1 . . . n] and an integer k. Describe an algorithm to find the k-th smallest element in the union of A and B in O(log n) time. (For example, if k = 1, your algorithm should return the smallest element of AUB; if k = n, your algorithm should return the median of AUB.) You can assume that the arrays contain no duplicate elements. [Hint: First solve the special case k = n.]

(a) Now suppose we are given three sorted arrays A[1 . . . n], B[1 . . . n], and C[1 . . . n], and an integer k. Describe an algorithm to find the k-th smallest element in AUBUC in O(log n) time.

(b) Finally, suppose we are given a two dimensional array A[1 . . . m][1 . . . n] in which every row A[i][ ] is sorted, and an integer k.Describe an algorithm to find the k-th smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? [Hint: Use the linear-time Select algorithm as a subroutine.]

Solution - 9

Listing 6: Code

```
#include <bits/stdc++.h>
using namespace std;

// Recursive function to find the kth minimum in 2 sorted arrays.
int kth_smallest(int* arr1,int* arr2,int* end1,int* end2,int n,int k){
    \\ If the first array is completed, then simply return kth element↩
        of the second array.
    if(arr1==end1){
        return arr2[k];
    }
    \\ If the second array is completed, then simply return kth ↩
       element of the first array.
    if(arr2==end2){
        return arr1[k];
    }
    // Get the mid indices.
    int mid1 = (end1 - arr1) / 2;
    int mid2 = (end2 - arr2) / 2;
    // If mid1 + mid2 < k, then start of the array having the smaller ↩
       element at mid becomes mid + 1.
    if (mid1 + mid2 < k)
    {
        if (arr1[mid1] > arr2[mid2]){
            return kth_smallest(arr1, arr2 + mid2 + 1, end1, end2,n,k ↩
                - mid2 - 1);
        }
        else{
           return kth_smallest(arr1 + mid1 + 1, arr2, end1, end2,n,k -↩
               mid1 - 1);
        }
    }
    // If mid1 + mid2 > k then end of the array with the larger value ↩
       at mid becomes mid - 1.
    else
    {
        if (arr1[mid1] > arr2[mid2]){
           return kth_smallest(arr1, arr2, arr1 + mid1, end2,n,k);
        }
        else{
           return kth_smallest(arr1, arr2, end1, arr2 + mid2,n,k);
        }
    }
}
int main() {
    int n;
    int x;
    int k;
    cin>>n>>k;
    int arr1[n+1];
    int arr2[n+1];
    for(int i=0;i<n;i++){
        cin>>x;
        arr1[i]=x;      // Get the 1st array.
    }
    for(int i=0;i<n;i++){
        cin>>x;
        arr2[i]=x;      // Get the second array.
    }
    cout<<kth_smallest(arr1,arr2,arr1+n,arr2+n,n,k-1)<<endl;
    return 0;
}
```

Correctness and Explanation:-

We compare the middle elements of arrays arr1 and arr2,let us call these indices mid1 and mid2 respectively. Now if the sum of mid1 and mid2 is less than k then the kth smallest will be after both mid1 and mid2, but we dont know in which array. So now we compare the mid elements and then decrease the size of the array we should be looking at. If the sum if mid1 and mid2 is > k then we can simply look at the first half of the array which has higher of the values at indices mid1 and mid2. After decreasing the size of the array, we can call this function recursively to get the kth smallest element when either one of the start indices equal the end indices.

Running Time Analysis:-

This sort of implementation is a divide and conquer approach very similar to that of binary search.

Here we will have O(logn) + O(logn) = O(logn). Therefore this requires O(logn) time complexity.

Solution - 9(a)

Listing 7: Code

```
#include <bits/stdc++.h>
using namespace std;

int kth_smallest2(int* arr1,int* arr2,int* end1,int* end2,int n,int k)↩
    {
     if(arr1==end1){
         return arr2[k];
     }
     if(arr2==end2){
         return arr1[k];
     }
     int mid1 = (end1 - arr1) / 2;
     int mid2 = (end2 - arr2) / 2;
     if (mid1 + mid2 < k)
     {
         if (arr1[mid1] > arr2[mid2]){
             return kth_smallest2(arr1, arr2 + mid2 + 1, end1, end2,n,k↩
                 - mid2 - 1);
         }
         else{
            return kth_smallest2(arr1 + mid1 + 1, arr2, end1, end2,n,k ↩
               - mid1 - 1);
         }
     }
     else
     {
         if (arr1[mid1] > arr2[mid2]){
             return kth_smallest2(arr1, arr2, arr1 + mid1, end2,n,k);
         }
         else{
             return kth_smallest2(arr1, arr2, end1, arr2 + mid2,n,k);
         }
     }
}
// Recursive call to find kth smallest element in 3 arrays.
int kth_smallest(int* arr1,int* arr2,int* arr3,int* end1,int* end2,int↩
    * end3,int n,int k){
```

```
// If any one of the arrays is done, then simply find kth minimum ↩
    in the remaining two arrays i.e, simply the previous solution.
if(arr1==end1){
    return kth_smallest2(arr2,arr3,end2,end3,n,k);
}
if(arr2==end2){
    return kth_smallest2(arr1,arr3,end1,end3,n,k);
}
if(arr3==end3){
    return kth_smallest2(arr1,arr2,end1,end2,n,k);
}
// Store the mid indices.
int mid1 = (end1 - arr1) / 2;
int mid2 = (end2 - arr2) / 2;
int mid3 = (end3 - arr3) / 2;
// If mid1 + mid2 + mid3 < k, then start of the array having the ↩
    smaller element at mid becomes mid + 1.
if (mid1 + mid2 + mid3 < k)
{
    if (arr1[mid1] < arr2[mid2] && arr1[mid1] < arr3[mid3]){
        return kth_smallest(arr1 + mid1 + 1, arr2, arr3, end1, ↩
           end2, end3,n,k - mid1 - 1);
    }
    else if (arr2[mid2] < arr3[mid3] && arr2[mid2] < arr1[mid1]){
        return kth_smallest(arr1, arr2 + mid2 + 1, arr3, end1, ↩
           end2, end3,n,k - mid2 - 1);
    }
    else if (arr3[mid3] < arr1[mid1] && arr3[mid3] < arr2[mid2]){
        return kth_smallest(arr1, arr2, arr3 + mid3 + 1, end1, ↩
           end2, end3,n,k - mid3 - 1);
    }
}
// If mid1 + mid2 + mid3> k then end of the array with the larger ↩
    value at mid becomes mid - 1.
else
{
    if (arr1[mid1] > arr2[mid2] && arr1[mid1] > arr3[mid3]){
        return kth_smallest(arr1, arr2, arr3, arr1 + mid1, end2, ↩
           end3,n,k);
    }
    else if (arr2[mid2] > arr3[mid3] && arr2[mid2] > arr1[mid1]){
        return kth_smallest(arr1, arr2, arr3, end1, arr2 + mid2, ↩
           end3,n,k);
    }
    else if (arr3[mid3] > arr1[mid1] && arr3[mid3] > arr2[mid2]){
        return kth_smallest(arr1, arr2, arr3, end1, end2, arr3 + ↩
           mid3,n,k);
        }
    }
}
int main() {
    int n;
    int x;
    int k;
    cin>>n>>k;
    int arr1[n+1];
    int arr2[n+1];
    int arr3[n+1];
    for(int i=0;i<n;i++){
        cin>>x;
        arr1[i]=x;                  // Take input.
    }
    for(int i=0;i<n;i++){
        cin>>x;
        arr2[i]=x;                  // Take input.
    }
    for(int i=0;i<n;i++){
        cin>>x;
        arr3[i]=x;                  // Take input.
    }
    cout<<kth_smallest(arr1,arr2,arr3,arr1+n,arr2+n,arr3+n,n,k-1)<<↩
       endl;
    return 0;
}
```

Correctness and Explanation:-

The solution for the b part is very similar to that of the case of 2 arrays. Here instead of looking at the mid of 2 arrays, we consider the 3 arrays. When mid1 + mid2 + mid3 < k, we can simply ignore the first mid part of the array with the smallest value at index = mid. Also when mid1 + mid2 + mid3 > k then we neglect the right portion of the array having the largest value at index = mid. When either of the array reach the end then we have the same question as the previous one with updated lengths and k. We solve this recursively to get the final answer.

Running Time Analysis:-

This solution is also very similar to that of binary search and therefore time complexity is O(logn). Here we will have O(logn) + O(logn) + O(logn) = O(logn).

Solution - 9(b)

Listing 8: Code

```
#include<bits/stdc++.h>
using namespace std;

// Store the row no of the elements in the heap in the order of min ↩
    heap.
int row_no[] = {0, 1, 2, 3};
// Store the column number of the element of that row which is present↩
     in the heap.
int col_no[4] = {0};
class min_heap{
  private:
    int *harr;
    int size;
  public:
    min_heap(int n){
        harr = new int[n];
        size = 0;
    }
    int get_parent(int i){return (i-1)/2;}
    int get_left(int i){return (2*i + 1);}
    int get_right(int i){return (2*i + 2);}
    void insert(int val);
    int get_size(){return size;}
    int extract();
    int get_top(){return harr[0];}
    void heapify(int i);
    void swap(int* x,int* y);
    void print(){for(int i=0;i<size;i++){cout<<harr[i]<<" ";}cout<<↩
       endl;}
};
// To swap 2 elements in an array.
void min_heap::swap(int* x,int* y){
    int temp = *x;
    *x = *y;
    *y = temp;
}
// Insert into the heap.
void min_heap::insert(int val){
    harr[size]=val;
    int i=size;
    size++;
    while (i!=0 && harr[get_parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[get_parent(i)]);
        swap(&row_no[i], &row_no[get_parent(i)]);
        i = get_parent(i);
    }
}
// Extract the minimum element and then heapify.
int min_heap::extract(){
    int x = harr[0];
    harr[0]=harr[size-1];
    row_no[0]=row_no[size-1];
    size--;
    heapify(0);
    return x;
}
// Rearrange the heap to get min heap property.
void min_heap::heapify(int i){
    int l = get_left(i);
    int r = get_right(i);
    int min = i;
    if (l < size && harr[l] < harr[i])
        min = l;
    if (r < size && harr[r] < harr[min])
        min = r;
    if (min != i)
    {
        swap(&harr[i], &harr[min]);
        swap(&row_no[i],&row_no[min]);
        heapify(min);
    }
}
int kth_smallest(int arr[4][4],int k){
    min_heap heap(4);
    for(int i=0;i<4;i++){
        heap.insert(arr[0][i]);
    }
    for(int i=0;i<k-1;i++){
        int x,y;
        // Insert till we dont exceed the length of a row.
        if(col_no[row_no[0]]<3){
            col_no[row_no[0]]+=1;
            x = row_no[0];
            y = col_no[row_no[0]];
            heap.extract();
            row_no[heap.get_size()]=x;
            heap.insert(arr[y][x]);
        }
        // If we are finished throught that row, then dont insert ↩
            anything.
        else{
            heap.extract();
        }

    }
    // The kth top element is the final answer.
    return heap.get_top();
}
int main()
{
    int k;
    cin>>k;
    int arr[4][4];          // Assuming we have a 4 x 4 array.
    for(int i=0;i<4;i++){
        for(int j=0;j<4;j++){
            cin>>arr[j][i];
        }
    }
    cout<<kth_smallest(arr,k);
    return 0;
}
```

Correctness and Explanation:-

Firstly the above code is written for an input matrix of 4 x 4 size for simplicity. It can be used for any length n, m by simply changing the values. The logic goes as follows:- We create a min heap of elements in the first column. Then we remove the min, heapify the heap, and insert the next element in the same row as the min element which we removed. Remove the kth min to get the kth smallest element in the 2-D array. Care has to be taken that if sometime we pass through a whole row, then we dont insert the next element and simply find and remove the min again and again.

Running Time Analysis:-

We create a min heap of size n i.e, no of rows. This is done in O(logn). Now removing and inserting 'k' elements will take O(klogn), so our time complexity would be O(n) + (klogn).

Question 10

For this problem, a subtree of a binary tree means any connected subgraph. A binary tree is complete if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the largest complete subtree of a given binary tree. Your algorithm should return the root and the depth of this subtree.

Solution - 10

Listing 9: Code

```
class node{
    public:
    int data;
    node* left;
    node* right;
};
node* createnode(int x){
        node* p = new node;
        p->data=x;
        p->left=NULL;
        p->right=NULL;
        return p;
}
int maxx=-1;
node *universalparentpointer=NULL;
int stud(node *t){
    if(t==NULL)return 0;
    int x=min(stud(t->left),stud(t->right));
    if(x>maxx){universalparentpointer=t;maxx=x;}
    return x+1;
}
void maxcompletesubtree(node *t){
    int temp=stud(t);
    cout<<maxx<<" "<<universalparentpointer->data<<"\n";
}
int main()
{
    int n;cin>>n;
    node* a[n];
    for(int i=0;i<n;i++){
        a[i]=createnode(i);
    }
    int e;cin>>e;
    int x,y,z;
    for(int i=0;i<e;i++){
        cin>>x>>y>>z;
        if(z==0){
            a[x]->left=a[y];
        }
        else{
            a[x]->right=a[y];
        }
    }
    maxcompletesubtree(a[0]);
}
```

Correctness and Explanation:-

We start by calling the function stud on the root of the tree, which in turn calls it on its children. The function returns the height of the maximum complete tree rooted at that node. The NULL pointers return 0. The parent takes the minimum of the return values of its children and then adds 1 to it before returning to its parent.
The reason for taking the minimum is that at that node as root we should have a complete tree, therefore we cannot have varying complete sub-tree sizes, hence we take the minimum of them.
Then at each node we compare it with the global maximum that we have accumulated so far, and then update it if we find a larger complete sub-tree and also change the universal parent pointer to that node, which stores the root of the maximum complete sub-tree.

Running Time Analysis:-

As the function is called at each node and only performs constant time operations at each node after receiving the return value, therefore it has time complexity O(n).

Question 11

You are given a sorted array of numbers where every value except one appears exactly twice; the remaining value appears only once. Design an efficient algorithm for finding which value appears only once.

Here are some example inputs to the problem:

1 1 2 2 3 4 4 5 5 6 6 7 7 8 8

10 10 17 17 18 18 19 19 21 21 23

1 3 3 5 5 7 7 8 8 9 9 10 10

Clearly, this problem can be solved in O(n) time, where n is the number of elements in the array, by just scanning across the elements of the array one at a time and looking for one that isn't paired with the next or previous array element. But can we do better?

Solution - 11

Listing 10: Code

```
int a[100000];
int findodd(int start, int end){
    if(start==end){
        return a[end];
    }
    else{
        int mid=(start+end)/2;
        if((mid-start)%2==1){  //mid is of the form 2k+1
            if(a[mid]==a[mid-1])
                return findodd(mid+1,end);  //2k and 2k+1 are same
            else
                return findodd(start,mid-1);  //check in first half
        }
        else{       //mid is of the form 2k
            if(a[mid]==a[mid+1])
                return findodd(mid+2,end);  //2k and 2k+1 are same
            else if(a[mid]==a[mid-1])
                return findodd(start,mid-2);  //2k and 2k-1 are same
            else
                return a[mid];  //mid is element of single occurence
        }
    }
}
int main(){
    int n;
    cin>>n;
    for(int i=0;i<n;i++){
        cin>>a[i];
    }
    cout<<findodd(0,n-1)<<"\n";
}
```

Correctness and Explanation:-

In the sorted array, if each value had appeared twice, then the entry at each even index would be the same as the next entry. This means that the elements at 2k and 2k+1 positions are the same for all k. The element that occurs only once is encountered at position 2i and thus after that element, the elements at the positions 2k-1 and 2k are same for all k.
Our algorithm makes use of this fact and checks the elements at indices 2k-1, 2k and 2k+1 using binary search. We check the middle three element of the array and if the elements at indices 2k and 2k+1 are same, then the element with the singular occurrence has not yet appeared in the first half. If the elements at 2k-1 and 2k indices are same, then it has appeared in the first half. If the element at 2k is different from both 2k-1 and 2k+1, then that is the element which occurs only once.

Running Time Analysis:- In each step, we perform $O(1)$ operations to compare elements, and reduce the problem size by half, the same as in binary search. Therefore the time complexity is the same as that of binary search, $O(\log n)$.

Question 12

You are given n balls with some of them coloured red and some black. Your mission is to identify a ball with a majority colour (if it exists). Here, majority means strictly more than other colour. You may pick two balls and ask if these two balls have same colour. What is the minimum number of queries needed for the task?

Solution - 12

We correspond the balls with numbers and store them in an array, using only the equality operations we try to find the majority element.

Listing 11: Code

```
FindMajorityElement(vector <int > a){
    int maxx=a[0];              //initialize majority E
    int count=1;                //initialize count
    for(int i=1;i<a.size();i++){
        if(a[i]==maxx){
            count++;
        }
        else{
            count--;
        }
        if(count==0){
            maxx=a[i];
            count=1;
        }
    }
    return maxx
}
```

Correctness and Explanation:-

The algorithm works in 2 steps.
First we get a possible majority element E, using the above mentioned function FindMajorityElement.
Second we check whether the element E is a real majority element or not by going over the array and keeping a count of number of occurences, and if the number of occurences is greater than n/2, then it is a real majority.
Above algorithm loops through each element and maintains a count of maxx. If the next element is same then increment the count, if the next element is not same then decrement the count, and if the count reaches 0 then changes the maxx to the current element and set the count again to 1.
The idea of the Step 1 is that if we cancel out each occurrence of an element E with all the other elements that are different from E then e will exist till end if it is a majority element.

Running Time Analysis:-

Since in Step 1, it makes n-1 comparisons and in Step 2 also it makes n-1 comparisons, the overall complexity of the algorithm is O(n).

(Reference Moore's Voting Algorithm.)

Question 13

You are given n identical looking gold coins where some are genuine and some are fake. Coins of same type have same weight, i.e., all fakes have weight and all genuine ones have weight b, but you do not know the values or relative values of a or b. You are given a balance which can be used to test if two coins have same weight or not. It is known that number of genuine coins is more than number of false coins. Find a genuine coin in as few weighings as possible.

Solution - 13

Refer to Question 12 Solution

Question 14

The problem consists of finding the lowest floor of a building from which a box would break when dropping it. The building has n floors, numbered from 1 to n, and we have k boxes. There is only one way to know whether dropping a box from a given floor will break it or not. Go to that floor and throw a box from the window of the building. If the box does not break, it can be collected at the bottom of the building and reused. The goal is to design an algorithm that returns the index of the lowest floor from which dropping a box will break it. The algorithm returns $n + 1$ if a box does not break when thrown from the nth floor. The cost of the algorithm, to be kept minimal, is expressed as the number of boxes that are thrown (note that re-use is allowed).
(a) For $k > \log(n)$, design an algorithm with $O(log(n))$ boxes thrown.
(b) For $k < \log(n)$, design an algorithm with $O(k + \frac{n}{2^{k-1}})$ boxes thrown.
(c) For $k = 2$, design an algorithm with $O(n)$ boxes thrown.

Solution - 14

(a) For when we have $k > \log(n)$ boxes, we can use binary search that requires at most $\log(n)$ probes as follows:

Listing 12: Code

```
int findLim(int start, int end){
    if (start==end && start==n) return n+1;
    if (start==end) return start;  //termination condition
    int mid = (start+end)/2;
    if (break[mid]==false)  //doesn't break
        return findLim(mid+1, end);
    else return findLim(start, mid);  //breaks
}
```

From the available floors, throw a box from the $\lfloor \frac{n}{2} \rfloor^{\text{th}}$ floor. If it breaks, then the limit lies in the lower half, else it lies in the upper half. As at most $\lceil \log(n) \rceil$ probes are required, we get an $O(log(n))$ algorithm.

(b) For when we have $k > \log(n)$ boxes, we take one box out. now, we have $k-1$ boxes and as we saw in the previous part, we can use $k-1$ boxes to probe 2^{k-1} floors. So we use the one box we took out earlier to identify which block of 2^{k-1} floors the limit is in. For that we throw the box after intervals of 2^{k-1} floors.

The number of such intervals is $\frac{n}{2^{k-1}}$ and the number of probes in that interval of floors is $k-1$. Therefore, we have obtained an algorithm with $O(k + \frac{n}{2^{k-1}})$ boxes thrown.

(c) For $k = 2$, we could just insert the value of k in the expression we obtained in the previous part to get $O(2 + \frac{n}{2})$.

The algorithm corresponding to this would be to throw the box at alternate floors, starting from 0. If the box breaks at level i, check level i-1. If the box breaks at i-1 too, then the limit is i-2, else it is i-1.

Therefore the total number of boxes thrown are $[\frac{n}{2}] + 1$. Thus obtained an algorithm with $O(n)$ boxes thrown.

Question 15

Alice controls 2 cops and Bob controls one thief in an n$\times$n rectangular grid. On a move Alice can move each cop to a neighbour or leave there without moving. Bob can do that for the thief. Show how Alice can nab the thief in 3n moves.

Solution - 15

Let the two cops be denoted by cop1,cop2, and the thief by t. Let us split the problem into 2 parts.
1. First the cop cop1 tries to match his y-coordinate with the thief. The cop cop2 tries to match his x-coordinate with the thief.
2. Then the cops move slowly closer to the thief step by step until he is caught.
We will show that the first step takes a maximum of 'n' steps and the second '2n'.
First step:- The maximum difference between the y-coordinate of the thief and the cop1 can be 'n' and so no matter how the thief moves, cop1 can match his y-coordinate with that of the thief in a maximum of 'n' steps. The same argument holds for the x-coordinate of the thief and cop2 and therefore in n steps the cops would have matched their y,x coordinates respectively.
Second step:- Without loss of generality let us assume that cop1 is to the left of the thief i.e, x-coordinate of the thief is $>$ that of cop1, and that the thief is above cop2. Now for every set of moves the thief makes, we can define the set of moved to be followed by each of the cops to nab the thief. If at any moment any of the cop catches the thief then we are done. Now moving onto the instructions —
If the thief takes a step to the left then the cop cop1 moves right and the cop2 moves left.
If the thief moves right then cop1 moves right and cop2 moves right.
If the thief moves up then the cop1 goes up and cop2 moves up.
If the thief goes down then cop1 moves down and cop2 goes up.
If the thief doesn't move then cop1 moves right and cop2 up.
Now let us assume that the thief is initially at (x,y). Let us denote the number of time the thief goes up by U, left by L, right by R, down by D, and the number of times he stays still by S. We can now get an inequality by performing some logical operations.
Since the thief is at y, $\mathbf{y+U-D <= n}$. Also y is the maximum distance between the thief and cop2 since cop2 is below the thief and can have any y-coordinate from 0 to y-1. Now after every L/R/U movement of the thief, the relative distance between the thief and cop2 doesnt change. After any D movement the relative distance decreases by 2 units, and for any S(still) state the distance decreases by 1 unit. Therefore we have $\mathbf{2*D + S <= y}$. Adding the above two equations we have $\mathbf{S + U + D <= n}$. Similarly using the distance between cop1 and the thief and the x-coordiante, we have $\mathbf{S + R + L <= n}$. Now adding these two, we have $\mathbf{2*S + U + D + R + L <= 2n}$. Therefore $\mathbf{S + U + R + L + D <= 2n}$ which implies that no matter how the thief moves, he will be caught in the Second step by atmost '2n' steps. Combining Step-1 and Step-2, we have that the thief will be caught in a total of '3n' steps.
Hence Proved.

Question 16

We have n threads with each one having a bead. We want to line up all the beads vertically by moving them along the threads. The objective is to minimize the total distance moved. Given n numbers denoting the positions of beads, design a linear time algorithm to find the minimum total distance the beads have to be moved in order to line them up.

Solution - 16

Listing 13: Code

```
long long int sum=0;
int m=FindMedianbyOS();
//median of the array by OrderStatistics if unsorted array(O(n))
for(int i=0;i<array.size();i++){
    sum+=abs(array[i]-m);
}
cout<<sum<<"\n";
```

Correctness and Explanation:-

We need to find the median of the given array. We prove the claim that the median is the required number such that the sum of the modulus of the distances of the array elements from it is minimum.
By simple observation it is easy to show that the required number (let it be A) should lie in the range of the array elements. Therefore A > min(array) and A < max(array).
We can show that it doesn't matter where A is in between the extremes of the array , the sum of the distance of the extreme from the number will be the same as D(min,number)=A-min and D(max,number)=max-A. Sum = max-min which is fixed. Therefore removing them doesn't affect the minimisation of the overall distance sum. Thus the problem will be changed to finding the number A such that sum of Distances is minimum and now the array doesn't include the previous array's extremes. Therefore A should also hold similar properties when the extremes are removed, i.e it should still lie within the remaining array extremes.

Continuously removing the extremes will leave us with either 2 or 1 elements. In-case of 2 elements we can choose A to be anything between the 2 numbers. In-case of 1 remaining number, we choose A to be that number itself. Since A obtained has equal number of elements bigger and smaller than it, it is by definition the median of the array elements. Since we have the median = M, the smallest distance can be computed directly by

$$\sum_{n=1}^{n}(|a[i] - M|)$$

Running Time Analysis:-

The summing of distance over the array is O(n), given that we have the median M. If the array is sorted then m=finding median is O(1), but even if the array is unsorted finding median is O(n). As proven in class that the median can be found by Order Statistics in O(n) time, we still have the time complexity as O(n).

Question 17

Let F be an array of size $n \geq 1$ whose elements are 0 or 1. A section $[i, ..., j]$ of consecutive elements of F, with $1 \leq i \leq j \leq n$, is balanced if it contains as many 0 as 1 elements. The length of a balanced section $[i, ..., j]$ is its number of elements, $j - i + 1$.Design an algorithm to find the longest balanced section of F.

Solution - 17

The array is stored in F. We iterate once through the array, and at store the value of the difference in the number of ones and zeroes encountered until now in another array- prefixSum. We now create another array hash with the size as the number of distinct elements in prefixSum which will be its max-min+1 as the elements start from zero and increment or decrement by only one at each step. Now we iterate through the prefixSum array and check for longest balanced section. Consider the pseudocode for this:

Listing 14: Code

```
int n0=0, n1=0;  //init number of 0s and 1s
int prefixSum[n+1];
prefixSum[0]=0;  //no element encountered yet
for(int i=0; i<n; i++){  //first iteration
    if (F[i] == 0) n0++;
    else if(F[i]==1) n1++;
    prefixSum[i+1] = n1-n0;  //difference between number of 1's and 0'↩
       s
}
int maxp=0, minp=0;   //init max and min of prefixSum
for(int i=0; i<n; i++){   //2nd iteration to find max and min of ↩
   prefixSum
    if(prefixSum[i+1] < minp) minp=prefixSum[i+1];
    if(prefixSum[i+1] > maxp) maxp=prefixSum[i+1];
}
int hash[max-min+1]={-1};   //initialize hash
int ans=0;   //init value of final answer
int finalindex=-1;
for(int i=0; i<n+1; i++){   //third iteration to find answer
    int val=prefixSum[i] - min;
    if(hash[val] == -1)
        hash[val]=i;   //first index where that value of prefixSum ↩
           happens.
    else{
        if((i - hash[i]+1) >ans){  //checks length of balanced section
            ans = (i-hash[i]+1);   //updating ans
            finalindex=i;  //final index
        }
    }
}
cout<<"Length of longest balanced section is "<<ans<<"\n";
cout<<"Starting index is "<<hash[finalindex]<<"\n";
cout<<"Ending index is "<<finalindex<<"\n"
```

Correctness and Explanation:-

At the starting and ending index of a balanced section, the total number of 1's minus the total number of 0's are equal.Our algorithm compares the value of this n1-n0 using a hash table. We make the hash table while making use of the fact that the values of the sums till the i^{th} element are consecutive integers. In the third iteration, we store the values of the first index where a value of n1-n0 happens (we subtract min to ensure the values start from zero). If a value of such a prefix computation has already happened, then that section between that hash[i] and i is balanced. We find the maximum of such lengths.

Running Time Analysis:-

Each iteration performs constant time operations over the elements of an array. Thus each iteration is $O(n)$. Therefore the overall time complexity is $O(n)$.
We also make use of additional two arrays, that is extra $O(n)$ space.

Question - 18

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	Both are good or both are bad
B is good	A is bad	At least one is bad
B is bad	A is good	At least one is bad
B is bad	A is bad	At least one is bad

(a) Show that if more than n = 2 chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

(b) Consider the problem of finding a single good chip from among n chips, assuming that more than n = 2 of the chips are good. Show that **floor[n/2]** pairwise tests are sufficient to reduce the problem to one of nearly half the size.

(c) Show that the good chips can be identified with O(n) pairwise tests, assuming that more than n = 2 of the chips are good. Give and solve the recurrence that describes the number of tests.

Solution - 18(a)

Let g be the number of good chips and n - g be the number of bad chips. Also, assume n - g >= g.From this assumption we have that we can always find a set G of good chips and a set B of bad chips of equal size g. Now, assume that the set B of bad chips conspire to fool the professor in the following way: for any test made by the professor, they declare themselves as 'good' and the chips in G as 'bad'. Notice that the chips in G report correct answers and then exhibit a symmetric behaviour: they declare themselves as 'good' and the chips in B as 'bad'. This implies that whichever is the strategy based on the kind of test considered and used by the professor, the behaviour of the chips in G is indistinguishable from the behaviour of the chips in B. This does not allow the professor to determine which chips are good.Hence proved.

Solution - 18(b)

Assume n is even. Then we can match chip in pairs $(c_{2i-1}; c_{2i})$ for i = 1;...; n=2 and test all these pairs. Then we throw away all pairs of chips that do not say 'good, good' and finally we take a chip from each pair. In this way, the final set contains at most n/2 chips and we are still keeping more good chips than bad ones, since all the pairs that we have discarded contain at least a bad chip.

Now, assume n is odd. Then, we test $\lfloor n/2 \rfloor$ pairs constructed as before; this time, however, we don't test one chip, say, c_n. Our final set will contain one chip for every pair tested that reported 'good, good'. Moreover, according to whether the number of such chips has the form 4k + 3 or 4k + 1, we

discard chip c_n or we put it into our final set.Now, why this construction works ?
First of all, as in the case in which n is even, by discarding all pairs of chips that report anything but 'good,good', we discard pairs of chips in which at least one is bad, and thus we still keep more good chips than bad ones.
Now, assume that we have discarded the pairs of chips that report anything but 'good,good', and we are left with 4k+ 3 chips, for some integer k. Then, in this case there are at least 2k + 2 good chips and thus there are at least k + 1 pairs of 'good,good' chips, and at most k pairs of 'bad,bad' chips. Thus chip c_n can also be discarded, and only a chip in each of these 2k + 2 pairs is not discarded.
Finally, assume that we have discarded the pairs of chips that report anything but 'good,good', and we are left with 4k + 1 chips, for some integer k. Then, in this case there are at least 2k + 1 good chips. Now, assume c_n is bad; then there are at least 2k + 2 good chips, that is, k + 1 pairs of 'good,good' chips; and at most k + 1 pairs of bad chips. Thus, taking a single chip from every pair, plus c_n gives us at least k + 1 good chips and at most (k - 1) + 1 = k bad chips; that is, a majority of good chips and our final set is at most of size $\lceil n/2 \rceil$. On the other hand, assume c_n is good. Then we have at least k pairs of 'good,good' chips and taking a chip from each pair plus c_n gives us a majority of good chips and a final set of size at most $\lceil n/2 \rceil$.

Solution - 18(c)

If at least n/2 of the chips are good, using the answer to question b), we can compute a single good chip. Now, in order to find all good chips, we can just test the good chip with all the others: for each pair of this type, knowing that at least one of the two chips in the pair is good is enough to decide whether the other chip is good or not. This last stage takes n - 1 more tests. Then, in order to prove that the goof chips can be found with O(n) pairwise tests, we just need to show that the number of tests to find a single good chip is O(n). To show this, we can write the number of tests T(n) that we do in the answer to question b) as T(n) <= T($\lceil n/2 \rceil$) + $\lfloor n/2 \rfloor$; and T(1) = 0. The solution to this recurrence is T(n) = O(n) and can be computed for instance by arguing and proving by induction that T (n) <= 2^{logn}

Question 19

Same as question 11!

Question 20

There are n identical boxes in which $2n$ balls are equally distributed. The balls are labelled from 1 to $2n$. We don't know which ball is in which box, but do know that each box contains two balls. The objective is to learn the arrangement of balls. For any set of balls $S \subseteq \{1, ..., 2n\}$ we can ask a query of the form 'How many boxes contain the balls in S?'. Prove that we can learn the distribution of balls by asking $O(n \log n)$ queries. Note that we cannot tell which ball is in which box. We only need to report the set of n pairs of balls.

Solution - 20

Algorithm :-

We have $2n$ balls. We take one ball B_i, perform the following operations recursively:
We partition the remaining balls into two sets S_1 and S_2 of size $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$. We now perform two queries, first on the set S_1 and then on the set $S_1 \cup B_i$. If we get the same answer for both, that means that the ball paired with B_i is in that set, else it is in the other set. This means that if the number of boxes that contain the balls in S_1 are k and if the number of boxes does not change if we ask about the set $S_1 \cup B_i$, then that means that the ball B_i goes into a box with one ball which is its pair. We can now partition the set S_1 similarly, or if we determine that B_i's pair is in S_2, we divide that.
We recursively divide the set we determine its pair to be in until we have only one ball left. We then remove these balls from the universal set, and chose another ball to perform this procedure on.

Running Time Analysis:-

For each ball B_i, we perform $O(1)$ queries in one stage, and at each stage the size of the set to be considered is reduced by half. This is similar to binary search in an array, and so the complexity of finding the pair of each B_i is an $O(\log n)$ operation.
We perform this for n elements, and so the overall complexity is $n \times \log n$ that is, $O(n \log n)$ queries.

Question - 21

Given a function $f : S \to S$ (that maps a natural number from a finite set S to another natural number in the same finite set S and an initial value x_0 E N, the sequence of iterated function values: x_0, $x_1 = f(x_0)$, $x_2 = f(x_1)$, . . . , $x_i = f(x_{i-1})$, . . . must eventually use the same value twice, i.e. $\exists i \neq j$ such that $x_i = x_j$. Once this happens, the sequence must then repeat the cycle of values from x_i to x_{j-1}. Let μ (the start of cycle) be the smallest index i and λ (the cycle length) be the smallest positive integer such that $x_\mu = x_{\mu+\lambda}$. The cycle-finding problem is defined as the problem of finding μ and λ given $f(x)$ and x_0.

Solution - 21

Listing 15: Code

```
#include <bits/stdc++.h>
using namespace std;
// Create a class node.
class node{
    int fx;
    int count;
    int visited;
};
// Create and add nodes into the graph.
node* createnode(int val){
    node* temp=new node;
    temp->fx=val;
    int visited=0;
    int count=-1;
    return temp;
}
int len=0;
int dfs(int l){
    if(a[l]->visited==1){return a[l]->count;}
    a[l]->visited=1;
    a[l]->count=len;
    len++;
    return dfs(a[l]->fx);
}
int main()
{
    int t=1;
    for(int ii=0; ii<t; ii++){
        int n;
        cin>>n;
        node* a[n];
        for(int i=0;i<n;i++){
            int x,y;
            cin>>x>>y;
            a[x]=createnode(y);
        }
        int ans=dfs(x0);                    // pass x0 into the function.
        cout<<len-ans<<" "<<ans<<"\n";
    }
}
```

Correctness and Explanation:-

Firstly we start off by storing the given inputs as nodes, with each node containing the value of $f(x)$ and the count of the cycle starting from that position, and a bool indicating if that node has been visited during our DFS. Now simply go through the graph in this manner with node being x_i and the edge relation being $f(x)$. When we encounter a visited node again, that means that at that point x_i = x_j, we then store the data and exit.

Running Time Analysis:-

The time complexity for this code is the same as that of DFS i.e, O(V+E).