Programming Assignment № 3

CS18B006, CS18B057, CS18B062 26-02-20

Question - 1

The Hadamard matrices $H_0, H_1, H_2, ...$ are defined as follows:

- H_0 is the 1×1 identity matrix [1].
- For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$\begin{bmatrix} \mathrm{H}_{k-1} & \mathrm{H}_{k-1} \\ \mathrm{H}_{k-1} & -\mathrm{H}_{k-1} \end{bmatrix} \tag{1}$$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

Solution - 1

If the column vector v is split into two halves (v_1, v_2), then the matrix-vector product $H_k v$ becomes $\begin{bmatrix} H_{k-1}v_1+H_{k-1}v_2 \\ H_{k-1}v_1-H_{k-1}v_2 \end{bmatrix}$. We can now compute the matrix-vector product recursively as described in the following pseudocode:

Listing 1: Code

```
vector<int> Matproduct(v){
    if (|v|==1)
        return v;
    vector v1=v[0..(|v|/2-1)];
    vector v2=v[(|v|/2)...(|v|-1)];
    r1=Matproduct(v1);
    r2=Matproduct(v2);
    vector ans[|v|];
    for(int i=0; i<|v|/2; i++){
        ans[i]=r1[i]+r2[i];
    }
    for(int i=|v|/2; i<|v|; i++){
        ans[i]=r1[i-|v|/2]-r2[i-|v|/2];
    }
    return ans;
}
```

Correctness and Explanation:-

The function above splits the vector into two halves and returns the same vector if it's size is one. Else it recursively calls itself to get r1 and r2 and returns the product vector in its simplified $\begin{bmatrix} H_{k-1}v_1+H_{k-1}v_2 \\ H_{k-1}v_1-H_{k-1}v_2 \end{bmatrix}$ form.

Running Time Analysis:-

The time complexity is given by

$T(n) = 2T(n/2) + O(n)$

which is the same as mergesort, and thus it's time complexity is $O(n \log n)$.

Question - 2

Let T be a table of n relative integers. Give an algorithm to find the maximum sum of contiguous elements, namely, two indices i and j ,$(1 \leq i \leq j \leq n)$ that maximizes $\sum_{k=i}^{j} T[k]$.

Solution - 2

Listing 2: Code

```
#include <bits/stdc++.h>
int max_sum(int arr[], int N) {
    int maxsum = 0;          // To store the maximal subsequence sum.
    int thissum = 0;        // Stores the present subsequence sum.
    for (int i = 0; i < N; ++i) {
        thissum += arr[i];  // Keep on adding the next element.
        if (thissum > maxsum) {
            maxsum = thissum;   // If max<current, then update max.
        } else if (thissum < 0) {
            thissum = 0;        // If sum becomes<0, then start again
        }                       // i.e, make current sum=0.
    }
    return maxsum;
}
int main() {
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
    cin>>arr[i];
    }
    cout<<max_sum(arr,n);
    return 0;
}
```

Correctness and Explanation:-

We keep track of the subsequence sum starting from index 0. Also keep updating Max-Sum accordingly. Let us assume the maximal subsequence sum if from index i to j. So that must mean that for any index $i_0<i$, the sum from i_0 to i must be negative since otherwise, the maximal sum would be from i_0 to j. This is being taken care of here, as since as soon as the sum becomes -ve, change the starting index, and change current sum to 0. This way we get the maximal subsequence sum in the most optimal way.

Running Time Analysis:-

We only have one iteration through the array, and therefore time complexity is O(n).

Question - 3

Solution - 3

Correctness and Explanation:-

Given:

f(1) = c1.

f(2) = c2.

If n>2 then f(n) has to be broken into some f(1) and f(2). This will be done using a recursion tree.

Let there be X no.of 1's and Y no.of 2's for the minimum value of f(n).

Let Z be the number of internal nodes of the tree.

Then following the recurrence relation the equations are:

X + Y - 1 = Z (Leaves = Internal Nodes + 1 : Complete Tree)

X + 2Y = n (Decomposition of n into 1's & 2's)

Xc1 + Yc2 + Zc3 = f(n)

(where c3 is the cost of each internal node)

On substituting Z = X + Y - 1 we get

f(n) = X(c1 + c3) + Y(c2 + c3) - c3

On substituting Z = n - 2Y

f(n) = n(c1 + c3) - Y(2c1 + c3 - c2) - c3

Therefore the only variable cost in f(n) is value of Y and this decides our way of splitting

2 Cases:

Case 1: c2 < 2c1 + c3

if n = 2k (even) then f(n) = k*(c2 +c3) - c3

if n = 2k+1 (odd) then f(n) = k*(c2 +c3) + c1

Here as c2 < 2c1 + c3 take as many 2s as possible and put that many c2 till either 0 or 1 is left.

This will result in the optimal splitting.

Proof:

If this is not optimum then only way to optimise it is to put two 1s instead of a 2. If the initial sum is s then now new sum will be s - c2 + 2c1 + c3 because each 1 will contribute a c1 and the division will contribute a c3.But from our case condition

$$\begin{aligned} s - c2 + c1 + c1 + c3 &= s + (2 * c1 + c3 - c2) \\ &> s \end{aligned}$$

Therefore s is the minimum. Hence proved that above is optimum.

Case 2: c2>= 2c1 + c3

In this case minimum is (n-2)c1 + (n-2)c3 + c2

Here we take as many 1's as possible till we reach 2 and then terminate there.

This will result in the optimal splitting.

Proof:

If this is not optimum then only way to optimise it is to put a 2 instead of two 1s. If the initial sum is s then now new sum will be s + c2 -2c1 - c3 because each of the 2 c1 will be replaced by a c2 and a division will be decreased so that is a reduction of c3.But from our case condition

$$s + c2 - 2c1 - c3 > s$$

Therefore s is the minimum. Hence proved that above is optimum.

Question - 4

Given a binary number i with binary representation $b_{k-1}...b_1b_0$, defined rev$_k(i)$ to be the number that has binary representation $b_0b_1...b_{k-1}$. (Example: rev$_3(3) = 6$ since 3 and 6 have binary representations "011" and "110" respectively.)
Given $n = 2^k$, consider the problem of computing the sequence rev$_k(0)$, rev$_k(1)$,..., rev$_k(n-1)$. (Example for $n = 8$, the output is (0,4,2,6,1,5,3,7) as the binary representations of these numbers are 000, 100, 010, 110, 001, 101, 011, 111.)
Give an $O(n)$-time algorithm by using divide and conquer. (This is faster than the trivial algorithm which reverses each number one by one in total time $O(nk) = O(n \log n)$.) Remember, arithmetic operations involving two integers are performed in $O(1)$ time.

Solution - 4

Claim: The required sequence $S(n)$ is
$S(n) = S_1(n/2).S_2(n/2)$, where n is a multiple of two
Where $S_1(n)$ is the sequence obtained when every element of $S(n)$ is multiplied by 2, and $S_2(n)$ is the sequence where every element in $S_1(n)$ is incremented by 1, and "." is the concatenation operator.
Proof: The first $n/2$ numbers would remain the same if the bits were not reversed, as they don't depend on the value of the left-most bit. But for the sequence with reversed bits, it would be an addition of a '0' bit to the right, that is a left shift. Therefore, the first $n/2$ numbers of the required sequence would be the sequence $S(n/2)$ with each element multiplied by 2.
The next $n/2$ numbers are 1 in the most significant bit followed by the the bits of all the previous numbers in order. Therefore, in the in the reverse sequence of bits, the numbers would be the numbers in the first half plus 1 in the same order.
Consider the recursive function that does this to give the array S_n.

Listing 3: Code

```
void RevSeq (S, n){
    if(n==1) {
        S[0]=0; return;
    }
    else{
        RevSeq(S, n/2);
        for(int i=0; i<n/2; i++){
            S[i] = 2*S[i];
        }
        for(int i=n/2; i<n; i++){
            S[i]=S[i - n/2] + 1;
        }
        return;
    }
}
```

The problem with this recurrence is that we are updating the values multiple times. It is possible to compute the final sequence given only the value of n. The first element in $S(n)$ is always 0 and the second element, if it exists, is 2^{k-1}. This is because the recurrence will run on $S[0]k$ times and $S[1]k-1$ times to give 0 and 2^{k-1}. In fact, every time a 1 is encountered, the next $n/2-1$ elements are $S[1,..,n/2-1]$ incremented by one, and each time this goes into the recursion, it is multiplied by 2. So, for computing the reverse sequence consider the following code:

Listing 4: Code

```
void RevSeqIter (S, n, k){
    S[0] = 0;
    S[1] = pow(2, k-1);
    int marker=S[1];
    if(n==0) return;
    int i=2;
    while(marker>0){
        S[i]=marker/2;
        int lim=i;
        i++;
        for(int j=1; j<lim; j++){
            S[i]=S[j] + S[lim];
            i++;
        }
        marker/=2;
    }
    return;
}
```

Correctness and Explanation:-

The former algorithm does what we described in the claim. It is correct, as shown in the proof earlier. The latter algorithm iteratively does the recursion, but usews the observations from the recursion to do it efficiently.

Running Time Analysis:-

The former function is a recursive one, and the complexity is given by the recurrence relation:
$T(n) = T(n/2)+n$ where $T(n/2)$ represents the recursive call, and n represents the $nO(1)$ arithmetic operations.
The solution of this recurrence relation gives the time complexity: $T(n) = O(n \log n)$
For the latter, each iteration of the while loop and the nested for loop increments the index in the sequence, and each iteration has only $O(1)$ operations. Therefore the time complexity for this algorithm is $O(n)$.

Question - 5

We are given an index k and two sorted arrays A and B of sizes n_a and n_b, respectively. Describe an O(log(n_a) + log(n_b)) time algorithm that can find the k-th smallest element among all n_a + n_b elements in the two arrays. Example: If A = (3, 9, 15) and B = (2, 4, 8, 10, 11), and k = 5, the output would be 9. You may assume that all elements are distinct. Hint: Aim for a recurrence of the form T(N) = T(N/2) + O(1), where N = $n_a n_b$. Think in terms of the product of the two arrays. More hint: Let m_a and m_b be the middle elements of A and B, respectively. If k < $(n_a + n_b)/2$, and $m_a < m_b$, can we prune half of one of the arrays? What about the other cases?

Solution - 5

Listing 5: 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 n1,int n2↩
    ,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,n1,↩
                n2,k - mid2 - 1);
        }
        else{
           return kth_smallest(arr1 + mid1 + 1, arr2, end1, end2,n1,n2↩
               ,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,n1,n2,k)↩
                ;
        }
        else{
            return kth_smallest(arr1, arr2, end1, arr2 + mid2,n1,n2,k)↩
                ;
        }
    }
}
int main() {
    int n1,n2;
    int x;
    int k;
    cin>>n1>>n2>>k;
    int arr1[n1+1];
    int arr2[n2+1];
    for(int i=0;i<n1;i++){
        cin>>x;
        arr1[i]=x;      // Get the 1st array.
    }
    for(int i=0;i<n2;i++){
        cin>>x;
        arr2[i]=x;      // Get the second array.
    }
    cout<<kth_smallest(arr1,arr2,arr1+n1,arr2+n2,n1,n2,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,basically just pruning to half of one of the array's. 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 $O(\log n_a) + O(\log n_b)$ time complexity since in the worst case, we need to go through both the arrays completely in a binary search kind of fashion.

Question - 6

(Computer screen display problem.) You are to design and implement a program, using divide-and-conquer and runnning in time O(n log n), to display objects on the screen. To make your life simple, all objects are rectangles and standing on the same horizontal line (say y = 0). The i-th object is specified as (Li;Hi;Ri) where Li and Ri are its leftmost and rightmost coordinates, respectively, and Hi is its height. The input is a sequence of triples of integers. The output should show the outline of the objects such that hidden parts of the objects are not shown.
For example, if the input is (you can assume 3 numbers in a line, separated by spaces):
(1,11,5),(2,6,7),(3,13,9),(12,7,16),(14,3,25),(19,18,22),(23,13,29),(24,4,28)
the output should be:
(1,11), (3,13), (9,0), (12,7), (16,3), (19,18), (22,3), (23,13), (29,0)

Solution - 6

Correctness and Explanation:-

Listing 6: Class Declaration

```
class node{
    public:
    int x;
    int h;
}
class input_node{
    public:
    int x;
    int h;
    int y;
}
```

Listing 7: Merging Function

```
vector<node >merge(vector<node>v1,vector<node>v2){
    int i=0;j=0,h1=-1;h2=-1;
    int n1=v1.size(),n2=v2.size();
    vector <node > v;
    while (i < n1 && j < n2)
    {
        if (v1[i] <= v2[j])
        {
            h1=v1[i].h;
            v1[i].h=max(v1[i].h,h2);
            v.push_back(v1[i]);
            i++;
        }
        else
        {
            h2=v2[j].h;
            v2[j].h=max(v2[j].h,h1);
            v.push_back(v2[j]);
            j++;
        }
    }
    while (i < n1)
    {
        v.push_back(v1[i]);
        i++;
    }
    while (j < n2)
    {
        v.push_back(v2[j]);
        j++;
    }
    return v;
}
```

Listing 8: Recursive Function Call Function

```
vector <node> merger(input_node a[],int start,int end){
    if(start==end){
        node n1,n2;
        n1.x=a[end].x;
        n2.x=a[end].y;
        n1.h=a[end].h;
        n2.h=0;
        vector <node > v;
        v.push_back(n1);
        v.push_back(n2);
        return v;
    }
    else{
        vector <node >a=merger(a,start,(start+end)/2);
        vector <node >b=merger(a,(start+end)/2+1,end);
        return merge(a,b);
    }
}
```

Listing 9: Main

```
int main()
{
    for(int ii=0; ii<t; ii++){
        input_node a[n];                              //input array
        vector <node> ans= merger(a,0,n-1);
    }
}
```

Running Time Analysis:-

Question - 7

You are given a pile of n coins and you may flip the first $m \leq n$ coins as a group.

- What is the least number of flips needed to ensure that the coins are all heads up in the worst case?
- What is the least average number of flips needed to ensure that the coins are all heads up if each arrangement of heads and tails occurs with uniform probability?
- What are the worst and average costs if the cost of each flip is proportional to the number of coins flipped?

Solution - 7

For each first m elements that we chose to flip, we make sure that all have the same parity to optimize the least number of flips required.
(a) The worst case would be when the coins are alternate heads and tails and the last coin is a tail. In that case, we flip the the first 1 coin so then the first 2 coins have the same parity, and then flip the first two coins such that the first three show the same face and so on. This requires n flips (first 1 coin, first 2 coins, ... first n coins).

(b) Let F denote the number of flips. We can write:
$F(n) = (\frac{1}{2} \times 1 + \frac{1}{2} \times 0) + F(n-1)$ with $F(0) = \frac{1}{2}$
We do a flip if the first element in $F(n-1)$ is different from the first element in $F(n)$ and don't do a flip if it's not. The probability of that happening is $\frac{1}{2}$. Solving this recurrence relation gives the least average number of flips required $= \frac{n}{2}$.

(c) If the cost is proportional to the number of coins flipped, then
For the worst case, we flip 1 coin, then 2 coins, and so on till n coins. Thus the cost would be $\sum_{i=1}^{n} i = \frac{(n)(n-1)}{2}$.
For the average case, the we take the first m coins and in each case the probability of flipping the coins is $\frac{1}{2}$. So the cost would be
$\sum_{i=1}^{n} \frac{i}{2} = \frac{(n)(n-1)}{4}$.

Question - 8

An $m \times n$ array A of real numbers is a Monge array if for all i, j, k, and l such that $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$ we have $A[i,j] + A[k,l] \leq A[i,l] + A[k,j]$. In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

$$\begin{array}{ccccc}
10 & 17 & 13 & 28 & 23 \\
17 & 22 & 16 & 29 & 23 \\
24 & 28 & 22 & 34 & 24 \\
11 & 13 & 6 & 17 & 7 \\
45 & 44 & 32 & 37 & 23 \\
36 & 33 & 19 & 21 & 6 \\
75 & 66 & 51 & 53 & 34
\end{array}$$

(a). Prove that an array is Monge if and only if for all i = 1, 2, . . . , m - 1, and j = 1, 2, . . . , n - 1 we have $A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$. (*Hint:* For the "if" part, use induction seperately on rows and columns.)

(b). The following array is not Monge. Change one element in order to make it Monge. (*Hint:* Use part (a).)

$$\begin{array}{cccc}
37 & 23 & 22 & 32 \\
21 & 6 & 7 & 10 \\
53 & 34 & 30 & 31 \\
32 & 13 & 9 & 6 \\
43 & 21 & 15 & 8
\end{array}$$

(c). Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \leq f(2) \leq \cdots \leq f(m)$ for any $m \times n$ Monge array.

(d). Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A:

Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row in A'. Then compute the leftmost minimum in the odd-numbered rows of A. Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in O$(m+n)$ time.

(e). Write the recurrence describing the running time of the algorithm described in part (d). Show that its solution is $O(m + n \log m)$.

Solution - 8(a)

The "only if" part is trivial, it follows form the definition of Monge array.

As for the "if" part, let's first prove that

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$$
$$\Rightarrow A[i,j] + A[k,j+1] \leq A[i,j+1] + A[k,j],$$

where $i < k$.
Let's prove it by induction. The base case of k = i + 1 is given. As for the inductive step, we assume it holds for k = i + n and we want to prove it for k + 1 = i + n + 1 If we add the given to the assumption, we get

$$A[i,j] + A[k,j+1] \leq A[i,j+1] + A[k,j] \qquad \text{(assumption)}$$
$$A[k,j] + A[k+1,j+1] \leq A[k,j+1] + A[k+1,j] \qquad \text{(given)}$$
$$\Rightarrow A[i,j] + A[k,j+1] + A[k,j] + A[k+1,j+1] \leq A[i,j+1] + A[k,j] + A[k,j+1] + A[k+1,j]$$
$$\Rightarrow A[i,j] + A[k+1,j+1] \leq A[i,j+1] + A[k+1,j]$$

Solution - 8(b)

We know that $A[i,j]+A[i+1,j+1] \leq A[i,j+1]+A[i+1,j]$, so we can check for what i,j does this inequality not hold. Clearly for $i = 1, j = 0$, we have $A[i][j] = 23$, $A[i+1,j+1] = 7$, $A[i,j+1] = 6$ and $A[i+1,j] = 22$. Therefore we have the L.H.S to be 30 and the R.H.S to be 28. Since L.H.S is > R.H.S, the inequality does not hold. So we should either decrease $A[i,j]/A[i+1,j+1]$ or increase $A[i,j+1]/A[i+1,j]$ with a factor of 2 or more.
If we decrease 23 by 2 or more , the inequality will be broken for $i,j = 0,0$. However if we increase 22 to 24, then we see that all the inequations are satisfied $\Rightarrow$

$$\begin{array}{cccc} 37 & 23 & \mathbf{24} & 32 \\ 21 & 6 & 7 & 10 \\ 53 & 34 & 30 & 31 \\ 32 & 13 & 9 & 6 \\ 43 & 21 & 15 & 8 \end{array}$$

is a valid Monge array with only one change at the element 22 becoming 24.

Solution - 8(c)

Let a_i and b_j be the leftmost minimal elements on rows a and b and let's assume that $i > j$. Then we have $A[j,a] + A[i,b] \leq A[i,a] + A[j,b]$.But

$$A[j,a] \geq A[i,a] (a_i \text{ is minimal})$$
$$A[i,b] \geq A[j,b] (b_j \text{ is minimal})$$

Which implies that

$$A[j,a] + A[i,b] \geq A[i,a] + A[j,b]$$
$$A[j,a] + A[i,b] = A[i,a] + A[j,b]$$

Which in turn implies that either:

$$A[j,b] < A[i,b] \Rightarrow A[i,a] > A[j,a] \Rightarrow a_i \text{ is not minimal}$$
$$A[j,b] = A[i,b] \Rightarrow b_j \text{ is not the leftmost minimal}$$

Solution - 8(d)

If μ_i is the index of the i-th row's leftmost minimum, then we have $\mu_{i-1} \leq \mu_i \leq \mu_{i+1}$. For i = 2k + 1, $k \geq 0$, finding μ_i takes $\mu_{i+1} - \mu_{i-1} + 1$ steps at most, since we only need to compare with those numbers. Thus

$$\begin{aligned}
T(m,n) &= \sum_{i=0}^{m/2-1} (\mu_{2i+2} - \mu_{2i} + 1) \\
&= \sum_{i=0}^{m/2-1} \mu_{2i+2} - \sum_{i=0}^{m/2-1} \mu_{2i} + m/2 \\
&= \sum_{i=1}^{m/2} \mu_{2i} - \sum_{i=0}^{m/2-1} \mu_{2i} + m/2 \\
&= \mu_m - \mu_0 + m/2 \\
&= n + m/2 \\
&= O(m+n).
\end{aligned}$$

Solution - 8(e)

The divide time is O(1), the conquer part is T(m/2) and the merge part is O(m + n). Thus,

$$\begin{aligned}
T(m) &= T(m/2) + cn + dm \\
&= cn + dm + cn + dm/2 + cn + dm/4 + \cdots \\
&= \sum_{i=0}^{\lg m-1} cn + \sum_{i=0}^{\lg m-1} \frac{dm}{2^i} \\
&= cn \lg m + dm \sum_{i=0}^{\lg m-1} \\
&< cn \lg m + 2dm \\
&= O(n \lg m + m).
\end{aligned}$$

Question - 9

A Toeplitz matrix is an NXN matrix A = (aij) such that a[i][j] = a[i-1][j-1] for i =2,3,...n and j =2,3,...n.

(a) Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

(b) Describe how to represent a Toeplitz matrix so that you can add two NXN Toeplitz matrices in O(n) time.

(c) Give an O(nlogn)-time algorithm for multiplying an NXN Toeplitz matrix by a vector of length n. Use your representation from part(b).

(d) Give an efficient algorithm for multiplying two NXN Toeplitz matrices. Analyze its running time.

Solution - 9-a

Addition:

The sum of 2 Toeplitz Matrices is a Toeplitz Matrix. This can be easily shown.

Let us take A and B as 2 Toeplitz Matrices of respective order, and C = A + B.

By matrix addition we know that C[i][j] = A[i][j] + B[i][j] for i,j belonging to (1,..n).- - - - - - -(1*)

As we have A[i][j] = A[i-1][j-1] and B[i][j] = B[i-1][j-1] for i = 2,3,4...n. , because A and B are Toeplitz. This implies that A[i][j] + B[i][j] = A[i-1][j-1] + B[i-1][j-1] for i = 2,3,4...n.

Now using (1*) we have that C[i][j] = C[i-1][j-1] for i = 2,3,4...n.

This proves that C is also Toeplitz.

Multiplication:

The product of 2 Toeplitz Matrices need not be a Toeplitz Matrix.

This can be easily shown by an example:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 1 \\ 2 & 3 \end{bmatrix}$$

$$C = A * B$$

$$C = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 3 & 1 \\ 2 & 3 \end{bmatrix}$$

$$C = \begin{bmatrix} 5 & 4 \\ 5 & 4 \end{bmatrix}$$

This is not a Toeplitz Matrix as C[1][1] != C[2][2]

Hence proven.

Solution - 9-b

The Toeplitz Matrix property implies that all the elements with indices x,y such that x-y = c, where c belongs to (-(n-1),-(n-2).....,0,1,2....(n-1)) have the same value.

Thus we can save the values as an array of size (size of set of values of c), such that we can get the value of A[i][j] by getting the value corresponding to the index of the value i-j.
As there are at max 2n-1 different values of c, hence size of array is also 2n-1.
Therefore now we can store an NXN matrix as an array of size 2n-1.
Addition:
Addition of 2 Toeplitz Matrices can now be understood as the addition of 2 arrays of size 2n-1.
Adding the values corresponding to the same index will result in the Addition Matrix.
C[index] = A[index] + B[index]
Thus the addition operation is now reduced to having a time complexity of O(size of array) = O(n).

Solution - 9-c

Using above representation we can assume this as multiplication of 2 polynomials $P(x) = c_0 + c_1x + ... + c_{2n2}x^{2n2}$ and $Q(x) = b_{n-1} + b_{n-2}x + ... + b_0x^{n-1}$. so P(x) and Q(x) can be calculated in O(log(n)) time using Fast fourier transformation (As explained above). The result will be a polynomial whose coefficients can give the elements in the product of the 2 arrays. The product will be such that $x^{n-k+n-1}$ coefficient is

$$\sum_{i=0}^{n-1} c_{n-k+1}b_i = \sum_{i=0}^{n-1} a_{k,i}b_i = y_k$$

Solution - 9-d

Multiplication of 2 $n \times n$ matrix can be broken down to multiplication of a $n \times n$ matrix with a column matrix of length n, n times. Here for the j^{th} column we can get directly from Ab_j. From above solution each of these multiplications as they are Toeplitz matrices takes O(nlog(n)) times. So for doing n such multiplications for n corresponding columns it takes

$$\begin{aligned} T(n) &= n * O(nlog(n)) \\ &= O(n^2log(n)) \end{aligned}$$

Question - 10

Given a polynomial $A(x)$ of degree-bound n, we define its tth derivative by

Solution - 10

Listing 10: Code.

```
```

Correctness and Explanation:-
Running Time Analysis:-

Question - 11

We have seen how to evaluate a polynomial of degree-bound n at a single point in $O(n)$ time using Horner's rule. We also have discovered how to evaluate such a polynomial at all n complex roots of unity in $O(nlgn)$ time using the FFT. We shall now show how to evaluate a polynomial of degree-bound n at n arbitrary points in $O(nlg^2n)$ time. To do so, we shall assume that we can compute the polynomial remainder when one such polynomial is divided by another in $O(nlgn)$ time, a result that we state without proof. For example, the remainder of $3x^3 + x^2 - 3x + 1$ when divided by $x^2 + x + 2$ is $(3x^3 + x^2 - 3x + 1)$mod$(x^2 + x + 2 = -7x + 5)$

Given the coefficient representation of a polynomial $A(x) = \sum_{k=0}^{n-1} a_k x^k$ and n point $x_0, x_1, ..., x_{n1}$, we wish to compute n values $A(x_0), A(x_1), ...A(x_{n-1})$. For $0 \leq i \leq j \leq k$, define the polynomials $\prod_{k=i}^{j}(x - x_k)$ and $Q_{ij}(x) = A(x)$mod$P_{ij}(x)$. Note that $Q_{ij}(x)$ has degree at most $j - i$.

(a) Prove that $A(x)$mod$(x - z) = A(z)$ at any point z.

(b) Prove that $Q_{kk}(x) = A(x_k)$ and that $Q_{0,n-1}(x) = A(x)$

(c) Prove that for $i \leq k \leq j$, we have $Q_{ik}(x) = Q_{ij}(x)$mod$P_{ik}(x)$ and $Q_{kj}(x) = Q_{ij}(x)$mod$P_{kj}(x)$.

(d) Give an $O(nlg^2n)$-time algorithm to evaluate $A(x_0), A(x_1), ..., A(x_{n-1})$.

Solution - 11(a)

To calculate $a_{n-1}x^{n-1}$mod$(x - z)$, we first multiply $(x - z)$ with $a_{n-1}x^{n-2}$ to get $a_{n-1}x^{n-1}$ with an extra $+z * a_{n-1}x^{n-2}$ term, so we then multiply $(x - z)$ with $z * a_{n-1}x^{n-3}$ to get rid of that term, but again we get an extra $+z^2 * a_{n-1}x^{n-3}$ term, and to get rid of that we again multiply $(x - z)$ with $z^2 * a_{n-1}x^{n-4}$, to get another unwanted extra term and so on so forth...until when the term being multiplied becomes $z^{n-2} * a_{n-1}$, with an extra $z^{n-1} * a_{n-1}$ term. Therefore the remainder when $a_{n-1}x^{n-1}$ is divided by $(x - z)$ is $z^{n-1} * a_{n-1}$, and so we have $a_{n-1}x^{n-1}$mod$(x - z) = z^{n-1} * a_{n-1}$. Similarly for $a_{n-2}x^{n-2}$ we have the remainder when divided by $(x - z)$ to be $z^{n-2} * a_{n-2}$ and so on...till the last term a_0 where we have $a_0$mod$(x - z) = a_0$. Therefore we have $A(x)mod(x - z) = (\sum_{k=0}^{n-1} a_k x^k)mod(x - z) = z^{n-1} * a_{n-1} + z^{n-2} * a_{n-2} + z^{n-3} * a_{n-3} + ... + a_0 = A(z)$.

Hence proved.

Solution - 11(b)

$Q_{kk}(x) = A(x)$mod$P_{kk}(x)$ => $Q_{kk}(x) = A(x)$mod$(x - x_k)$, which from the previous result is equal to $A(x_k)$ (Let z = x_k). Hence $Q_{kk}(x) = A(x_k)$.

For the second part, we have $P_{0,n-1}(x) = (x - x_0) * (x - x_1) * ... * (x - x_{n-1})$, which is of order n. Also, $A(x) = \sum_{k=0}^{n-1} a_k x^k$ is of order $n - 1(< n)$ and so we have $Q_{0,n-1}(x) = A(x)$mod$P_{0,n-1}(x) = A(x)$.

Hence Proved.

Solution - 11(c)

Consider $A(x)$, let $A(x)$ be equal to $f(x) * P_{ij} + c_{j-i}(x)$, where $c_{j-i}(x)$ represents a polynomial of order <= $j - i$.This is true because the degree fo P_{ij} is given by $j - i + 1$ and so the remainder when divided by P_{ij} must be of lesser degree. Therefore we have, $Q_{ij}(x) = A(x)\text{mod}P_{ij} = c_{j-i}(x)$. Now, the R.H.S becomes $Q_{ij}(x)modP_{ik}(x) = c_{j-i}(x)\text{mod}P_{ik}(x)$. The L.H.S is given by $A(x)\text{mod}P_{ik}$, and since we know that $A(x) = f(x) * P_{ij} + c_{j-i}(x)$, and that j >= k, $P_{ij}\text{mod}P_{ik} = 0$, and so the L.H.S becomes $c_{j-i}(x)\text{mod}P_{ik}(x)$.Clearly L.H.S = R.H.S.
Hence Proved the first part.
Now similarly for the second part, we have R.H.S = $Q_{ij}(x)modP_{kj}(x) = c_{j-i}(x)\text{mod}P_{kj}(x)$. The L.H.S is given by $A(x)\text{mod}P_{kj}$, and since we know that $A(x) = f(x) * P_{ij} + c_{j-i}(x)$, and that k >= i, $P_{ij}\text{mod}P_{kj} = 0$, and so the L.H.S becomes $c_{j-i}(x)\text{mod}P_{kj}(x)$.Clearly L.H.S = R.H.S.
Hence Proved.

Solution - 11(d)

To get $A(x_0), A(x_1), ..., A(x_{n-1})$, it is sufficient to find $Q_{00}, Q_{11}, Q_{22}, ...Q_{n-1,n-1}$, since $Q_{k,k} = A(x_k)$ from part b. Now, to get $Q_{k,k}$, we can use a recursive code as follows:-

Listing 11: Pseudo Code

```
// Let the values of A(x) at the points be stored int the array arr[].
void get_val(int i,int j, Poly Q(ij)){
    if(i==j){
        arr[i]= Q(i,i);              // A(x(i)) = arr[i] = Q(ii).
    }
    else{
        int k = (i+j)/2;
        Q(i,k) = Q(i,j)modP(i,j);        // O(nlogn).
        Q(k+1,j) = Q(i,j)modP(k+1,j);    // O(nlogn).
        get_val(i,k,Q(ik));
        get_val(k+1,j,Q(k+1,j));
    }
}
int main(){
    get_val(0,n-1,Q(0,n-1));
    return 0;
}
```

Correctness and Explanation:-

We use a binary type of recursive function to get the values of Q(kk) at every k $\in$ [0,n-1]. If i==j, then just store the value, if not then split it into 2 halves and call the functions for each half.

Running Time Analysis:-

We have $T(n) = 2T(n/2) + O(n \log(n))$, which by master's theorem leads to $T(n) = O(n \log^2(n))$. Therefore we have a $O(n \log^2(n))$ solution to find the value of the polynomial at n points.

Question - 12

An inversion in an array A[1..n] is a pair of indices (i,j) such that i <j and A[i] >A[j]. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in O(nlogn) time. [Hint: Modify mergesort.]

Solution - 12

Correctness and Explanation:-

We use the basic Mergesort Function and make a very subtle modification in it's Merging Function. If we know the number of inversions in both the sub-arrays, the inversions not accounted for in them are the inversions occurring during the merging part. Thus we just need to count those inversions and then call it recurively on the sub-arrays as well.

While merging we check which of the elements is smaller and greater. During this process when we encounter such a case that an element in the right array to be merged is smaller than the current element in the left array to be merged i.e L[i] > R[j], it is a case of inversion with R[j]. Now as the 2 arrays are sorted we know that for i<=k<=(size_of_L) all elements L[k] are also greater than R[j]. Therefore we add all these vertices pairs in the no_of_inversions global variable. We repeat this over all merging operations and update the count respectively.

Listing 12: Global Variable

```
int no_of_inversions=0;
```

Listing 13: Modified Merging Function

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
            //No.of elmts greater than R[j] in L are (n1-i)
            no_of_inversions+=(n1-i);
        //**Modification**
            //Each time we find a smaller elmt in the right array,
            //update the inversions count by the no of elmts
            //greater than it.
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Listing 14: Mergesort Function

```
void mergeSort(int arr[], int start, int end)
{
    if (start < end)
    {
        int mid = (start+end)/2;
        mergeSort(arr, l, mid);
        mergeSort(arr, mid+1, r);
        merge(arr, l, mid, r);
    }
}
```

Listing 15: Main

```
int main()
{
    int arr[no_of_vertices];//input array
    mergeSort(arr, 0, no_of_vertices - 1);
    cout<<no_of_inversions<<"\n";
}
```

Running Time Analysis:-

Since the code is exactly same as that of Mergesort, and only incudes a constant time operation O(1) in its merging function, its time complexity still remains the same i.e O(NlogN) .

Question - 13

(a) Suppose you are given two sets of n points, one set $\{p_1, p_2, ..., p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, ..., q_n\}$ on the line $y = 1$. Create a set of n line segments by connecting each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time.
(b) Now suppose you are given two sets $\{p_1, p_2, ..., p_n\}$ and $\{q_1, q_2, ..., q_n\}$ of n points on the unit circle. Connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time.[Hint: Use your solution to part (a)].
(c) Describe an algorithm for part (b) that runs in $O(n \log n)$ time. [Hint:Use your solution from part (b)].

Solution - 13

(a)

We sort the array $P[1..n]$ and permute the array $Q[1..n]$ to maintain correspondence. (This can be done by storing the arrays as a vector of pairs and sorting by first.) This is done in $O(n \log n)$ time. Now, for any indices $i < j$ the line segments intersect if and only if $Q[i] > Q[j]$. Thus we just have to find the number of inversion pairs in Q. See Q-12 for the algorithm to find number of inversion pairs. That algorithm runs in $O(n \log n)$ and so the overall complexity of determining the number of line segments which intersect is $O(n \log n)$.

(b)

Note that the number of intersections do not depend on the actual positions of the points along the circle, but only on their position relative to other points, for example, for each i, which other pairs (p_j, q_j) have exactly one end-point in the arc clockwise from p_i to q_i. Therefore, for simplicity we will assume that the points are numbered and specified according to their appearance in clockwise order starting from some arbitrary location, call it "origin", on the circle. So, for example, if $p_1 = 5$, then p_1 is the fifth point clockwise from the origin in the set $P \cup Q$.
Now we sort the array $P[1..n]$ and permute the array $Q[1..n]$ to maintain correspondence. (This can be done by storing the arrays as a vector of pairs and sorting by first in $O(n \log n)$ time.) We walk through the segments in clockwise direction and for any segment (p_i, q_i), the segment (p_j, q_j) will intersect it if and only if:
$p_i < p_j < q_i < q_j$ or $p_j < p_i < q_j < q_i$
Since we are moving in a sorted array, $p_i < p_j$ for $i < j$. So for every segment i we need to find the number of segments j such that $p_j < q_i$ and $q_i < q_j$.
Divide the two sets of points into two equal halves. The total number of intersections would be the sum of those that intersect in two halves and those which intersect across the halves. To get the number of intersections across the halves,

(c)

Question - 14

Solution - 14(a)

Let $i_0 = log_b(n/n_0)$, we know that $T(n) = aT(n/b) + f(n)$, and so $T(n/b) = aT(n/b^2) + f(n/b)$ => $aT(n/b) = a^2T(n/b^2) + af(n/b)$, and similarly we have

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ aT(n/b) &= a^2T(n/b^2) + af(n/b) \\ a^2T(n/b^2) &= a^3T(n/b^3) + a^2f(n/b^2) \; ... \\ a^{i_0-1}T(n/b^{i_0-1}) &= a^{i_0}T(n/b^{i_0}) + f(n/b^{i_0-1}) \\ a^{i_0}T(n/b^{i_0}) &= a^{i_0}d = a^{i_0}f(n_0) = a^{i_0}f(n/b^{i_0}) \end{aligned}$$

Adding all these equations and cancelling terms from the left hand side and the right hand side, we have

$$\begin{aligned} T(n) &= f(n) + af(n/b) + a^2f(n/b^2) + ... + a^{i_0-1}f(n/b^{i_0-1}) + a^{i_0}f(n/b^{i_0}) \\ &= \sum_{i=0}^{i_0} a^i f(n/b^i) \\ &= \sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i). \end{aligned}$$

Hence proved.

Solution - 14(b)

Let $i_0 = log_b(n/n_0)$,

$$T(n) = \begin{cases} \Theta(n^p) & \text{if } f(n) \in O(n^p/(log(n))^{1+q}) \\ \Theta(f(n)log(n)log(log(n))) & \text{if } f(n) \in \Theta(n^p/log(n)) \\ \Theta(f(n)log(n)) & \text{if } f(n) \in \Theta(n^p(log(n))^{q-1}) \\ \Theta(f(n)) & \text{if } f(n) \in \Theta(n^{p+q}) \end{cases}$$

CASE-1:-

$f(n) \in O(n^p/(log(n))^{1+q})$ => $f(n/b^i) \in O(n^p/b^{ip}(log(n/b^i))^{1+q})$

$$\begin{aligned}
T(n) &= \sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i) \\
&\leq \sum_{i=0}^{log_b(n/n_0)} a^i \frac{n^p}{b^{ip}(log(n/b^i))^{1+q}} \\
&\leq \sum_{i=0}^{log_b(n/n_0)} \frac{n^p}{log(n/b^i)^{1+q}} \quad \text{(Since } a = b^p\text{)}. \\
&\leq \sum_{i=0}^{i_0} \frac{n^p}{(log(n) - log(b^i))^{1+q}}. \\
&\leq \sum_{i=0}^{i_0} \frac{n^p}{(log(n) - ilog(b))^{1+q}}. \\
&\leq \sum_{i=0}^{i_0} \frac{n^p}{(log(n) - (i_0 - i)log(b))^{1+q}}. \\
&\leq \sum_{i=0}^{i_0} \frac{n^p}{(log(n) - i_0 log(b) + ilog(b))^{1+q}}. \\
&\leq \sum_{i=0}^{i_0} \frac{n^p}{(log(n_0) + ilog(b))^{1+q}}. \\
&\leq n^p \sum_{i=0}^{i_0} \frac{1}{(log(n_0) + ilog(b))^{1+q}}. \\
&\leq n^p \sum_{i=0}^{\infty} \frac{1}{i^{1+q}}. \\
&\leq c.n^p. \quad \text{(Since } \frac{1}{i^x}\text{converges for } x > 1\text{and given that q>0).} \\
T(n) &= O(n^p).
\end{aligned}$$

Also we know that $\sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i) \geq a^{i_0} f(n_0) \geq a^{log_b(n/n_0)} \geq (n/n_0)^{log_b a} \geq n^p$ => $T(n) = \Omega(n^p)$.

Therefore $T(n) = \Theta(n^p)$.

Hence proved.

CASE-2:-

$f(n) \in \Theta(n^p/logn)$

$$T(n) = \sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i).$$

$$= \Theta(\sum_{i=0}^{log_b(n/n_0)} a^i \frac{n^p}{b^{ip} log(n/b^i)}).$$

$$= \Theta(\sum_{i=0}^{log_b(n/n_0)} \frac{n^p}{log(n/b^i)}) \quad \text{(Since } a = b^p\text{).}$$

$$= \Theta(\sum_{i=0}^{i_0} \frac{n^p}{(log(n) - log(b^i))}).$$

$$= \Theta(\sum_{i=0}^{i_0} \frac{n^p}{(log(n) - (i_0 - i)log(b))}).$$

$$= \Theta(\sum_{i=0}^{i_0} \frac{n^p}{(log(n) - i_0 log(b) + ilog(b))}).$$

$$= \Theta(\sum_{i=0}^{i_0} \frac{n^p}{(log(n_0) + ilog(b))}).$$

$$= \Theta(\frac{n^p}{log(b)} \sum_{c}^{c+i_0} \frac{1}{i}) \qquad , c = log_b(n_0).$$

$$= \Theta(\frac{n^p}{log(b)} \int_c^{c+i_0} \frac{1}{x} dx).$$

(Since we can write the summation as the riemann sum,

$$= \Theta(n^p \log(\frac{c+i_0}{c})).$$

which is both upper and lower bound).

$$= \Theta(n^p \log(\frac{\log_b(n)}{c})).$$

$$= \Theta(n^p \log(\log(n))).$$

Hence Proved.

CASE-3:-

$f(n) \in \Theta(n^p(log(n))^{q-1})$

$$\begin{aligned}
T(n) &= \sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i). \\
&= \Theta(\sum_{i=0}^{log_b(n/n_0)} a^i \frac{n^p(\log(n) - i\log(b))^{q-1})}{b^{ip}}). \\
&= \Theta(\sum_{i=0}^{log_b(n/n_0)} n^p(\log(n) - i\log(b))^{q-1}). \\
&= \Theta(n^p \sum_{i=0}^{log_b(n/n_0)} (\log(n) - (i_0 - i)\log(b))^{q-1}). \\
&= \Theta(k \times n^p \sum_{i=0}^{log_b(n/n_0)} (\log(n_0) + i\log(b))^{q-1}). \\
&= \Theta(k \times n^p \sum_{i=0}^{i_0} (i)^{q-1}). \\
&= \Theta(n^p i_0^q). \\
&= \Theta(n^p \log(n)^q). \\
&= \Theta(f(n)\log(n)). \quad \text{(Since } f(n) = \Theta(n^p(log(n))^{q-1})
\end{aligned}$$

Hence Proved.

CASE-4:-

$f(n) \in \Theta(n^{p+q})$

$$\begin{aligned}
f(n) &\leq c \times n^{p+q}. \\
T(n) &\leq c \sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i). \\
&\leq c \sum_{i=0}^{log_b(n/n_0)} a^i \frac{n^{p+q}}{b^{ip+iq}}. \\
&\leq c \sum_{i=0}^{log_b(n/n_0)} \frac{n^{p+q}}{b^{iq}}. \\
&\leq cn^{p+q} \sum_{i=0}^{log_b(n/n_0)} \frac{1}{b^{iq}}. \\
&\leq cn^{p+q} \sum_{i=0}^{\infty} \frac{1}{b^{iq}}. \\
&\leq cn^{p+q} \frac{1}{1-b^q}. \\
&\leq cn^{p+q}.
\end{aligned}$$

Therefore $T(n) = O(n^{p+q})$.

$$\begin{aligned}
f(n) &\geq c \times n^{p+q}. \\
T(n) &\geq c \sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i). \\
&\geq c \sum_{i=0}^{log_b(n/n_0)} a^i \frac{n^{p+q}}{b^{ip+iq}}. \\
&\geq c \sum_{i=0}^{log_b(n/n_0)} \frac{n^{p+q}}{b^{iq}}. \\
&\geq cn^{p+q} \sum_{i=0}^{log_b(n/n_0)} \frac{1}{b^{iq}}. \\
&\geq cn^{p+q} \times k \quad \text{(Since } \sum_{i=0}^{log_b(n/n_0)} \frac{1}{b^{iq}} \text{converges to some definite value k, as all terms are finite).} \\
&\geq cn^{p+q}.
\end{aligned}$$

Therefore $T(n) = \Omega(n^{p+q}) => T(n) = \Theta(f(n))$.
Hence Proved.

Solution - 14(c)

$T(n) \in (n^p)$whenever$f(n) \in (n^r)$ for some real constant $r < p$.
To prove this it is sufficient to prove that $\Theta(n^r) = O(n^p/\log(n)^{1+q})$, since then $f(n) \in O(n^p/\log(n)^{1+q})$, and using our result from case-1, we have $T(n) \in (n^p)$. To prove $\Theta(n^r) = O(n^p/\log(n)^{1+q})$, consider n^{p-r}, we must have $n^{p-r} > \log(n)^{1+q}$ for some q and sufficiently large n, and so $n^r < n^p/\log(n)^{1+q}$ $=> \Theta(n^r) = O(n^p/\log(n)^{1+q})$.
Hence Proved.

Solution - 14(d)

For $f(n) \in \Theta(n^{p+q}/\log n)$, we have

$$\begin{aligned}
T(n) &= \sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i). \\
&= \Theta(\sum_{i=0}^{log_b(n/n_0)} a^i \frac{n^{p+q}}{b^{ip+iq}\log(n/b^i)}). \\
&= \Theta(\sum_{i=0}^{log_b(n/n_0)} \frac{n^{p+q}}{b^{iq}\log(n/b^i)}). \\
&= \Theta(n^{p+q} \sum_{i=0}^{log_b(n/n_0)} \frac{1}{b^{iq}\log(n/b^i)}). \\
&= \Theta(\frac{n^{p+q}}{\log n} \sum_{i=0}^{log_b(n/n_0)} \frac{1}{b^{iq}(1 - i\log_n b)}).
\end{aligned}$$

Also we know that $\sum_{i=0}^{log_b(n/n_0)} \frac{1}{b^{iq}(1 - i\log_n b)}$ must be bound by some constant k, so

$$\begin{aligned}
T(n) &= \Theta(\frac{n^{p+q}}{\log n} \sum_{i=0}^{log_b(n/n_0)} \frac{1}{b^{iq}(1 - i\log_n b)}). \\
&= \Theta(k \times \frac{n^{p+q}}{\log n}). \\
&= \Theta(\frac{n^{p+q}}{\log n}). \\
&= \Theta(f(n)).
\end{aligned}$$

Hence proved that $T(n) = \Theta(f(n))$.
Now for $f(n) \in \Theta(n^{p+q}\log n)$, we have

$$
\begin{aligned}
T(n) = &\sum_{i=0}^{log_b(n/n_0)} a^i f(n/b^i). \\
&= \Theta(\sum_{i=0}^{log_b(n/n_0)} a^i \frac{n^{p+q}\log(n/b^i)}{b^{ip+iq}}). \\
&= \Theta(\sum_{i=0}^{log_b(n/n_0)} \frac{n^{p+q}\log(n/b^i)}{b^{iq}}). \\
&= \Theta(n^{p+q} \sum_{i=0}^{log_b(n/n_0)} \frac{\log(n/b^i)}{b^{iq}}). \\
&= \Theta(n^{p+q}\log n \sum_{i=0}^{log_b(n/n_0)} \frac{1-\log_n b}{b^{iq}}). \\
&= \Theta(n^{p+q}\log n \times k). \quad \text{(Since } \sum_{i=0}^{log_b(n/n_0)} \frac{1-\log_n b}{b^{iq}} \text{ must be bound by some constant k).} \\
&= \Theta(n^{p+q}\log n). \\
&= \Theta(f(n)).
\end{aligned}
$$

Hence proved that $T(n) = \Theta(f(n))$.
For the 3rd part, we have $g(bn) \geq \alpha g(n)$ for all $n \in X => g(n) \geq \alpha g(n/b)$.
$T(n)$ can be written as $\sum_{i=0}^{log_b(n/n_0)} a^i g(n/b^i)$, since $f(n) \in \Theta(g(n))$. Also $g(n) \geq \alpha g(n/b) => g(n) \geq \alpha^i g(n/b^i) => g(n/b^i) \leq g(n)/\alpha^i$, so we have $\sum_{i=0}^{i_0} g(n/b^i) \leq g(n) \times \sum_{i=0}^{i_0} \frac{1}{\alpha^i}$, and so $\sum_{i=0}^{i_0} a^i g(n/b^i) \leq g(n) \times \sum_{i=0}^{i_0} (\frac{a}{\alpha})^i$.

$$
\begin{aligned}
T(n) = &\sum_{i=0}^{log_b(n/n_0)} a^i g(n/b^i). \\
&= \Theta(g(n) \times \sum_{i=0}^{i_0} (\frac{a}{\alpha})^i). \\
&= \Theta(g(n) \times k). \quad \text{(Since } \sum_{i=0}^{i_0} (\frac{a}{\alpha})^i \text{ will be bounded by a constant k).}
\end{aligned}
$$

Therefore $T(n) = \Theta(g(n)) = \Theta(f(n))$..
Hence Proved.

Question - 15

Solution - 15

Correctness and Explanation:-

We know that if we have the ConvexHulls of the left part and the right part and we want to find the ConvexHull of the complete set, we need to merge the individual convex hulls. We achieve this by using the upper-tangent and the lower-tangent method.
Now the problem comes down to finding the convex hulls of the left and the right part. This is where we utilize the paradigm of divide and conquer, we recursively divide the given vertex set into smaller and smaller sets and we bottom out at set size ≤ 5, and then solve it for them using the brute force algorithm. Then finally recursively calling the merge operation on these smaller vertex sets will lead to the Convex Hull of the original Vertex set.

Listing 16: Algorithm for upper tangent:

```
L <- line joining the rightmost point of a
     and leftmost point of b.
while (L crosses any of the polygons)
{
    while(L crosses b)
        L <- L* : the point on b moves up.
    while(L crosses a)
        L <- L* : the point on a moves up.
}
```

Listing 17: Algorithm for lower tangent:

```
L <- line joining the rightmost point of a
      and leftmost point of b.
while (L crosses any of the polygons)
{
    while (L crosses b)
       L <- L* : the point on b moves down.
    while (L crosses a)
       L <- L* : the point on a moves down.
}
```

Listing 18: Merging of 2 Convex Hulls

```
vector<pair<int, int>> merger(vector<pair<int, int> > a,
                              vector<pair<int, int> > b)
```

```
{
    // n1 -> size of polygon a
    // n2 -> size of polygon b
    int n1 = a.size(), n2 = b.size();

    int ia = 0, ib = 0;

    // ia -> rightmost point of a
    for (int i=1; i<n1; i++)
        if (a[i].first > a[ia].first)
            ia = i;

    // ib -> leftmost point of b
    for (int i=1; i<n2; i++)
        if (b[i].first < b[ib].first)
            ib=i;

    // finding the upper tangent
    // using the upper tangent algorithm
    pair<<int><int>> p=UpperTagent(a,b);
    int uppera = p.first, upperb = p.second;

    // finding the lower tangent
    // using the lower tangent algorithm
    pair<<int><int>> p=LowerTagent(a,b);
    int lowera = p.first, lowerb = p.second;

    vector<pair<int, int>> ret;

    //ret contains the convex hull after merging the two convex hulls
    //with the points sorted in anti-clockwise order

    int ind = uppera;
    ret.push_back(a[uppera]);
    while (ind != lowera)
    {
        ind = (ind+1)%n1;
        ret.push_back(a[ind]);
    }

    ind = lowerb;
    ret.push_back(b[lowerb]);
    while (ind != upperb)
    {
        ind = (ind+1)%n2;
        ret.push_back(b[ind]);
    }
    return ret;

}
```

Listing 19: Solving ConvexHull-BruteForce

```
vector<pair<int, int>> solve_Hull_Bruteforce(vector<pair<int, int>> a)
{
    // Brute-Force Method
    // Take any pair of points from the set and check
    // whether it is the edge of the convex hull or not.
    // if all the remaining points are on the same side
    // of the line then the line is the edge of convex
    // hull otherwise not

    set<pair<int, int> >s;
    // Contains the set of vertices in the ConvexHull
    // after solving them by brute-force

    for (int i=0; i<a.size(); i++)
    {
        for (int j=i+1; j<a.size(); j++)
        {
            int x1 = a[i].first, x2 = a[j].first;
            int y1 = a[i].second, y2 = a[j].second;

            int a1 = y1-y2;
            int b1 = x2-x1;
            int c1 = x1*y2-y1*x2;
            int positive = 0, negative = 0;

            for (int k=0; k<a.size(); k++)
            {
                if (a1*a[k].first+b1*a[k].second+c1 <= 0)
                    negative++;
                if (a1*a[k].first+b1*a[k].second+c1 >= 0)
                    positive++;
            }

            if (positive == a.size() || negative == a.size())
            {
                s.insert(a[i]);
                s.insert(a[j]);
            }
        }
    }
    vector<pair<int, int>>ret;
    for (auto e:s)
        ret.push_back(e);

    // Sorting the points in the anti-clockwise order

    mid = {0, 0};
    int n = ret.size();
    for (int i=0; i<n; i++)
    {
        mid.first += ret[i].first;
        mid.second += ret[i].second;
        ret[i].first *= n;
        ret[i].second *= n;
    }

    sort(ret.begin(), ret.end(), compare);

    for (int i=0; i<n; i++)
        ret[i] = make_pair(ret[i].first/n, ret[i].second/n);

    return ret;
}
```

Listing 20: Divide Function

```
// Returns the convex hull for the given set of points
vector<pair<int, int>> divide(vector<pair<int, int>> a)
{
    // If the number of points is less than 6 then the
    // function uses the brute-force algorithm to find the
    // convex hull
    if (a.size() <= 5)
        return solve_Hull_Bruteforce(a);

    // If size>5 we divide it into 2 parts
    // and call divide on both of them recursively
    // left contains the left half points
    // right contains the right half points
    vector<pair<int, int>>left, right;
    for (int i=0; i<a.size()/2; i++)
        left.push_back(a[i]);
    for (int i=a.size()/2; i<a.size(); i++)
        right.push_back(a[i]);
    // convex hull for the left and right sets
    vector<pair<int, int>>left_hull = divide(left);
    vector<pair<int, int>>right_hull = divide(right);

    // merging the convex hulls
    return merger(left_hull, right_hull);
}
```

Listing 21: Int Main

```
int main()
{
    vector<pair<int, int> > input;
    // sorting the set of points according
    // to the x-coordinate
    sort(input.begin(), input.end());
    vector<pair<int, int> >ansvector = divide(input);
}
```

Running Time Analysis:-

We divide the Convex-Hull into exactly 2 parts at every step of the Divide and Conquer approach, (bottoming out at 5) calling it recursively on both parts.
Merging of 2 Convex-Hulls takes O(n) time, therefore giving us the reccurence relation
$T(n) = 2T(n/2) + O(n)$ which on solving gives $T(n) = O(nlogn)$.