Tutorial - 3

Implement program of Divide & conquer & find Time complexity

q.1) Inversion count of array

given array, find the number of inversions of it. If (i<j) and (A[i] > A[j]), then pair (i,j) is called Inversion of an array. We need to count all such pairs in the array.

for example:

Input : A[] : [1,9,6,4,5]

output : The inversion count is 5

There are 5 inversions in the array

(9,6), (9,4), (9,5), (6,4), (6,5)

→

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

int merge1(int a[], int temp[], int l, int m, int r)
{
  int i,j,k,inv=0;
  i=l;
  j=m;
  k=l;
  while((i<=m-1) && (j<=r)){
      if(a[i]<=a[j]){
          temp[k++]=a[i++];
      }
      else{
          temp[k++]=a[j++];
          inv+=(mid-i);
      }
  }
  while(i<=m-1){
      temp[k++]=a[i++];
```

```
while (j<=r){
    temp[k++] = a[j++];
}
for (int ml=l; ml<=r; ml++){
    a[ml] = temp[ml];
}
return inv;
}

int mergesort(int a[], int temp[], int l, int r){
    int inv=0;
    if (l<r){
        int m = (l+r)/2;
        inv += mergesort(a, temp, l, m);
        inv += mergesort(a, temp, m+1, r);
        inv += merge(a, temp, l, m+1, r);
    }
    return inv;
}

int main(){
    int n;
    cin>>n;
    int A[n];
    for(int i=0; i<n; i++){
        cin>>A[i];
    }
    int temp[n];
    cout << mergesort(a, temp, 0, n-1);
}
```

Here, we have used mergesort to count inversion pair.

mergesort algorithm

```
Algo Mergesort (a, l, h)                    ---- T(n)
{
   if(l<h)
   { mid = (l+h)/2;                         ---- 1
     mergesort (l, mid);                    ---> T(n/2)
     mergesort (mid+1, h);                  ---> T(n/2)
     merge (l, mid, h);                     ---> n
   }
}
```

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

solving recurrence relation

$$T(n) = \begin{cases} 1 & n=1 \\ 2T\left(\frac{n}{2}\right)+n & n>1 \end{cases}$$

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right)+\frac{n}{2}\right]+n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + n+n$$

$$= 2^2\left[2T\left(\frac{n}{2^3}\right)+\frac{n}{2^2}\right]+2n$$

$$= 2^3T\left(\frac{n}{2^3}\right)+n+2n$$

$$= 2^3T\left(\frac{n}{2^3}\right)+3n$$

$$\vdots$$

$$= 2^kT\left(\frac{n}{2^k}\right)+kn$$

assume, $n = 2^k \Rightarrow k = \log_2 n$

Date

$= 2^k \cdot T(1) + kn$

$= 2^k + (\log n)\, n$

$= n + n \log n$

$= \Theta(n \log n)$

Time complexity of this program is $O(n \log n)$

Q.2) find the number of rotations in a circularly sorted array.

Given a circularly sorted integer array, find the number of times the array is rotated. Assume there are no duplicate in the array, and the rotation is in anti-clockwise direction.

for example:-

Input - arr = [8, 9, 10, 2, 5, 6]
output - The array is rotated 3 times

Input - arr = [2, 5, 6, 8, 9, 10]
output - The array is rotated 0 times.

→ code:-

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

int rotation (int A[], int l, int r, int n) {
    int mid=(l+r)/2;
    int p = (mid+n-1)%n;
    int nm= (mid+1)%n;
    if( A[p]>=A[mid] && A[mid]<=A[nm])
    { return mid;
    }
    if( A[mid]<=A[r]){
        return rotation(A,l,mid-1,n);
    }
    if( A[l]<=A[mid]){
        return rotation( A,mid+1,r,n);
    }
}
```

```
int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    cout<<rotation(arr,0,n-1,n);
}
```

→ Here we use binary search ~~Apt~~ Algorithm approach to find no. of rotation.

recurrence relation calculation

```
int rotation(int arr[], int f, int l, int n) ⟶ T(n)
{
    int mid = (f+l)/2;                          — 1
    int prev = (mid+n-1)%n;                     — 1
    int next = (mid+1)%n;                       — 1

    if(arr[prev]>=arr[mid] &&
        arr[mid]<=arr[next])                    — 1
    { return mid;
    }
    if(arr[mid]<=arr[l]){
      return rotation(arr,f,mid-1,n);           — T(n/2)
    }
    if(arr[f]<=arr[mid]){                       — T(n/2)
      return rotation(arr,mid+1,l,n);
    }
}
```

$T(n) = 2T\left(\frac{n}{2}\right) + 4$

solving reccurence relation

$$T(n) = \begin{cases} 1 & n=1 \\ T\left(\frac{n}{2}\right)+4 & n>1 \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right)+4$$

$$= T\left(\frac{n}{2^2}\right)+4+4$$

$$= T\left(\frac{n}{2^3}\right)+4+4+4$$

$$= T\left(\frac{n}{2^3}\right)+3(4)$$

$$= T\left(\frac{n}{2^k}\right)+k\cdot(4)$$

as, $\frac{n}{2^k}=1 \quad \therefore n=2^k \quad \therefore k=\log_2 n$

$$= T(1)+\log n(4)$$

$$= 4\log n+1$$

Time complexity = $O(\log n)$

Here we use binary search Algorithm Approach to find minimum element in given array. The index of minimum element is equal to no. of rotation it take to become sorted array.

Q.3) maximum sub array sum using Divide and conquer.

Given an integer array, find the maximum sum array among all sub arrays possible. The problem differs from the problem of finding maximum subsequence sum unlike subsequences, sub arrays are required to occupy consecutive position within the original array.

for example:

Input A[] = [2, -4, 1, 9, -6, 7, -3]

output : The maximum sum of sub array is 11.

—A program :-

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

int max_sum (int arr[], int n){
    if (n==1) return arr[0];
    int m= n/2;
    int left_max = max_sum(arr, m);
    int right_max = max_sum(arr+m, n-m);
    int left_sum = INT_MIN;
    int right_sum = INT_MIN;
    int sum=0;
    for (int i=m; i<n; i++)
    {  sum += arr[i];
       right_sum = max(right_sum, sum);
    }
```

```
sum=0;
for(int i=m-1;i>=0;i--)
{
  sum+=arr[i];
  left_sum= max(left_sum,sum);
}

int ams = max(left_max, right_max);
return max(ams, left_sum+right_sum);
}

int main(){
  int arr[]={2,-4,1,9,-6,7,-3};
  cout<<"sum is:"<<max_sum(arr,7);
}
```

Here we get recurrence relation as

$$T(n) = \begin{cases} c & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + c'n & \text{if } n>1 \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + c'n$$

$$= 2\left[2T\left(\frac{n}{2^2}\right) + c'\left(\frac{n}{2}\right)\right] + c'n$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + c'n + c'n$$

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + c'\left(\frac{n}{2^2}\right)\right] + 2c'n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + c'n + 2c'n$$

$$= 2^k T\left(\frac{n}{2^k}\right) + k \cdot (c'n)$$

we know $\frac{n}{2^k} = 1 \Rightarrow T(1) = c$

$\therefore$ $n = 2^k$

$\therefore k = \log_2 n$

$$= n \cdot c + \log_2 n \, (c'n)$$

$$= cn + c'n \log n$$

Time complexity = $O(n \log n)$