Data Structure and Algorithm Training Program

Week 2 : Pseudo Code Solution

Problem 1 : Write a Program to compute the value of power function x^n.

Brute force Iterative Approach

- Time Complexity : O(n)
- Space Complexity : O(1)

```
int pow(x, n)
{
    p=1
    i=0
    while (i<n)
      {
        p = p*x
        i = i+1
      }

   return p
}
```

Recommended Problems
1] Iterative solution for power(x, n)
2] Power in Modular Arithmetic
3] Karatsuba algorithm for fast multiplication

Brute force Divide and conquer approach

- T(n)= 2T(n/2) + c
  Time Complexity : O(n)
  Space Complexity : O(logn)
  [Recursion call stack]
- Condition check : Odd and even values of n
- We are Solving same subproblem Twice. Can we improve the time complexity via Solving subproblem only once **(Think!)**

```
int pow(x, n)
{
   if (n == 0)
         return 1

   else if (n is even)
         p = pow(x, n / 2) * pow(x, n / 2)
   else
         p = x * pow(x, n / 2) * pow(x, n / 2)

  return p
}
```

Efficient Divide and conquer approach

- T(n)= T(n/2) + c
  Time Complexity : O(logn)
  Space Complexity : O(logn)
  [Recursion call stack]
- Condition check : Odd and even values of n
- **Key learning** : Reducing subproblems in divide and conquer approach helps to improve the time complexity really well.

```
int pow( x, n)
 {
    if( n == 0)
        return 1

    p = pow(x, n/2)
    if ( if n is even)
        p = p*p
    else
        p=x*p*p

    return p
 }
```

Problem 2 : Fixed point in a sorted Array

```
int fixedPoint(A[], l, r)
{
    if(r >= l)
      {
        mid = l + (r - l)/2

        if(mid == A[mid])
          return mid

        if(mid < A[mid])
          return fixedPoint(A, l , mid-1)

        else
          return fixedPoint(A, mid+1 , r)
      }
  return -1
}
```

Brute force Solution : Linear Search

- Time Complexity : O(n)
- Space Complexity : O(1)

Efficient Solution via Divide and conquer approach of binary search

- Time Complexity : O(logn)
- Space Complexity : O(logn) [Recursion call stack]

Problem 3 : Maximum value in an array which is first increasing and then decreasing

```
int getMax(A[], l , r)
{
   if (l== r)
      return A[l]
  else if (r == l + 1)
    {   if(A[l] < A[r])
          return A[r]
        else
          return A[l]
    }
   mid = l + (r - l)/2

   if ( A[mid] > A[mid + 1]  &&  A[mid] > A[mid - 1])
      return A[mid]

   else if (A[mid] > A[mid + 1]  &&  A[mid] < A[mid - 1])
      return getMax(A, l, mid-1)

   else
      return getMax(A, mid + 1, r)
```

Recommended Problems
1] Given a list of n-1 integers and these integers are in the range of 1 to n.Find the missing integer.
2] Given a sorted array(elements can be repeated) and a number x, find the frequency of x in the array.
3] A given integer K appears more than n/2 times in a sorted array of n integers.Find the integer K.

Problem 4 : Square root of a given integer K . If K is not the perfect square then truncate the decimal digits

```
int truncatedSqrt( n )
{
    if (n == 0 || n == 1)
        return x
    k = 1
    value = 1
    while (value <= n)
     {
        k= k+1
        value = k * k
     }

  return k - 1
}
```

Brute force Solution : Approach of linear search

- Time Complexity : O(√ n) **(Think!)**
- Space Complexity : O(1)
- Check for every possible integer from start

Efficient Solution : Approach of binary search

- Time Complexity : O(logn)
- Space Complexity : O(1)
- There are several old methods for finding square roots for an integer : Babylonian method and Newton–Raphson method.

```
int truncatedSqrt(n)
 {
    if (n == 0 || n == 1)
        return n
    l = 1
    r = x
   while (l<= r)
   {
     mid = (l + r) / 2
     if (mid*mid == n)
         return mid
     else if (mid*mid > n)
         end = mid-1
     else
       {
           l = mid + 1
           value = mid
       }
   }
return value
}
```

Iterative Steps similar to binary search where dividing the number till we find the square root (floor value)

Think and try to visualize this code via few examples!

Recommended Problems
1] Find square root of number upto given precision
2] GCD of two numbers a and b where a > b

Problem 5 : Kth smallest element of two sorted array of size m and n

Brute force approach of merging two sorted array using extra memory

- Time Complexity : O(m+n)
- Space Complexity : O(m+n)
- **Critical Question** : Can be reduce the space complexity of this solution to O(1)? **Think and try !**

```
int kth_Smallest(A[], B[],m, n, k)
{
    Take extra memory C[m + n]
    i = 0, j = 0, K = 0
    while (i < m && j <n)
    {
        if (A[i] < B[j])
        {
            C[k]=A[i]
            i = i + 1
            k = k + 1
        }
        else
        {
            C[k] = B[j]
            j = j + 1
            k = k + 1
        }
    }
    while (i<m)
    {
        C[k]=A[i]
        i = i + 1
        k = k + 1
    }
    while (j<n)
    {
        C[k]=Y[j]
        j = j + 1
        k = k + 1
    }
    return C[k-1]
}
```

Better Solution via DivIde and Conquer Approach of binary search

- Time Complexity : O(logn + logm)
- Space Complexity : O(logn(max(m,n)) [Recursion call stack] **is this correct?**

Think about the following aspects of the solution

- Base cases and Function Parameters during recursive calls
- Comparison of mid1+mid2 with k
- Comparison of A[mid1] and B[mid2]
- Can be improve the time complexity of this solution to O(logk)? **Try this challenge!**

```
int kth_Smallest(A[], B[], l1 , l2, r1, r2, k)
{
    if (l1 == r1)
        return B[k]
    if (l2 == r2)
        return A[k]
    mid1 = l1 + (r1 - l1)/2
    mid2 = l2 + (r2 - l2)/2
    if (mid1 + mid2 > k)
    {
        if (A[mid1] > B[mid2])
            return kth_Smallest(A,B ,l1, mid1, l2 ,r2, k)
        else
            return kth_Smallest (A,B ,l1, r1, l2, mid2, k)
    }
    else
    {
        if (A[mid1] > B[mid2])
            return kth_Smallest (A, B ,l1, r1, mid2+1, r2,
                                 k - mid2 - 1)
        else

            return kth_Smallest (A,B, mid1+1, r1, l2, r2,
                                 k - mid1 - 1)
    }
}
```

Recommended Problems
1] Find rank of element in a unsorted array. Rank is the position of the element when it is sorted.
2] Find the max and second max in the an array of integers

Problem 6 : Write a program to check array B[] is subset of array A[] or not

Brute force approach

- Assuming m>n
- Searching every value of B[] in A[] via linear search
- Time Complexity :O(mn)
- Space Complexity : O(1)

```
int array_Subset(A[], B[], m, n)
{
 for (i = 0 to n-1)
  {
    for (j = 0 to m-1)
    {
     if(B[i] == A[j])
       break
    }
   if (j == m)
    return -1
  }
 return 1
}
```

Better approach

- After sorting array A[], Searching every value of B[] in A[] via binary search
- Time Complexity : O(mlogm + nlogm)
- Space Complexity : O(logm) [Recursion call stack of Quick sort]

```
int array_Subset(A[], B[], m, n)
{
 quickSort(A, 0, m-1)
 for (i=0 to n-1)
 {
  k = binarySearch(A, 0, m - 1, B[i])
  if (k==-1)
    return -1
 }
return 1
}
```

A very good problem to understand the idea of improving time complexity via different approach!

Better approach

- After sorting array A[] and B[], using two pointer approach to search each elements of B[] in array A[]
- Time Complexity : O(mlogm + nlogn)
- Space Complexity : O(logm + logn) [Recursion call stack for quick sort]

```
int check_Array_Subset(A[], B[], m, n)
{
 quickSort(A,0, m-1)
 quickSort(B,0, n-1)
 while (i < m && j < n )
  {
    if( A[i] < B[j] )
      i = i+1
    else if( A[i] == B[j] )
    {
      j = j+1
      i = i+1
    }
    else if( A[i] > B[j] )
      return -1
  }
  if (j<n)
   return -1
  else
   return 1
}
```

Think about this idea!

Why this!

Efficient approach via Hashing

- Inserting all the elements of array A[] in hash table and then searching each element of array B[] in Hash Table
- Time Complexity : O(m+n)
- Space Complexity : O(m)

```
int array_Subset(A[], B[], m, n)
 {
   Take HashTable H of Size m
   for(i = 0 to m-1)
    {
     H.insert(A[i])
    }

   for(int i = 0 to n-1)
    {
     if(H.Search(B[i]) is false)
      return -1
    }
  return 1
 }
```

We Will learn the idea of Hashing very soon!

Problem 7 : Write a program to check for pair in A[] with sum equal to K

Brute force approach

- Use two nested loops and check A[i] + A[j] == K for each pair (i, j) in array A[]
- Time Complexity :O(n^2)
- Space Complexity : O(1)

```
int find_Sum_Pair( A[], n, K)
{
    for (i = 0 to n - 1)
      {
        for (j = i + 1 to n -1)
         {
           if (A[i] + A[j] == K)
             return 1
         }
      }
   return -1
}
```

A very good problem to understand the idea of improving time complexity via different approach!

Better Approach than Brute Force

- Sort the array A[] and use two pointer approach to search a pair (i, j)
- Time Complexity : O(nlogn)
- Space Complexity : O(logn) [Recursion call stack of quick sort]

```
int find_Sum_Pair( A[], n, K)
{
   quickSort(A, 0 , n-1)

   i = 0
   j = n - 1
   while (i < j)
   {
     if(A[i] + A[j] == K)
        return 1
     else if(A[i] + A[j] < K)
        i = i+1
     else
        j = j-1
   }
  return -1
}
```

Why initializing the pointers with start and end? Think

Think about this idea!

Efficient approach via Hashing

- Insert all the elements of array A[] in hash table and then search K-A[i] for each elements in Hash Table
- Time Complexity : O(n)
- Space Complexity : O(n)

```
int find_Sum_Pair(A[], n, K)
{
   Take HashTable H of Size n
   for(i = 0 to n-1)
    {
        H.insert(A[i])
    }
   for (i = 0 to n-1)
    {
       value = K - A[i]
       if(H.Search(value) is true)
         return 1
    }
return -1

}
```

Inserting elements in Hash Table

Searching K-A[i] in Hash Table

Problem 8 : Inversion count in an Array

Brute force approach

- Use two nested loops and check A[i] > A[j] for each pair i<j in A[] and count the inversion.
- Time Complexity :O(n^2)
- Space Complexity : O(1)

```
int inversion_Count(A[], n)
{
  icount = 0
   for (i = 0 to < n - 1)
     {
        for (j = i + 1 to < n)
        {
            if (A[i] > A[j])
            icount = icount + 1
        }
     }
   return icount
}
```

Think about this!

```
int inversion_Count(A[], l,  r)
{
  icount1=0, icount2=0, icount3=0
  mid = l + (r - l)/2

  icount1 = inversion_Count(A[],l, mid)
  icount2 = inversion_Count(A[],mid+1, r)
  quickSort( A, l, mid)
  quickSort( A, mid+1, r)
   i = l
   j = mid +1
   while (i < mid && j <r)
   {
      if (A[i] < A[j])
          i = i + 1
      else
       {
          j = j + 1
          icount3 = icount3 + (mid - i)
       }
   }

  return (icount1 + icount2 + icount3)
}
```

Better Approach than Brute force via divide and conquer [Similar to merge sort]

- T(n)= 2T(n/2) + O(nlogn).
- Time Complexity : O(n(logn)^2). **Can we apply master theorem here?**
- Space Complexity : O(logn)[Recursion call stack of quick sort]
- **Divide Part :** Dividing the array into two equal halves via mid
- **Conquer Part :** Recursively calculating the inversion count of the left half and right half
- **Combine Part :** Calculate the inversion count crossing the both smaller array of size n/2 . Brute approach for this would take O(n^2).If both the left and right half would be sorted then we can easily calculate the "Inversion count crossing the both the half" in O(n). **Total Inversion count** = Inversion count left half + Inversion count right half + Inversion count crossing both the half
- **Time complexity of combine part** = For sorting both the half O(nlogn) + For finding inversion count for crossing both the sorted half O(n) = O(nlogn)
- **Think closely :** Instead of this approach, We can directly use merge sort code to count inversion because it can sort both the parts recursively and we can easily customize merging two sorted array to count the inversion.

Continued...

Efficient Approach : Customized merge sort to count the inversion

- T(n)= 2T(n/2) + O(n) ->Time Complexity : O(nlogn) [Better than earlier approach]
- Space Complexity : O(n) for merging two halves of sorted arrays
- Calculate the inversion count during the combine part of merge sort

- **Key Learning** : Sometime, the approach of few problem works directly in solving other problems!

```
int inversion_Count(A[], l ,r )
{
        icount1=0, icount2=0, icount3=0
        if (r > l)
        {
                mid = l + (r - l)/2
                icount1 = inversion_Count(A[], l , mid)
                icount2 = inversion_Count(A[], mid+1 ,r)
                icount3 = inversion_Crossing(A, l, mid, r)
        }
        return (icount1 + icount2 + icount3)
}
```

```
int inversion_Crossing (A[],l, mid, r)
{
    n1= mid-l+1
    n2= r-mid
    take extra array X[n1] and Y[n2]
    for(i = 0 to n1-1)
      X[i] = A[l+i]
    for(j = 0 to n2-1)
      Y[j] = A[mid+1+j]

    icount = 0, i = 0, j = 0, k = 0
    while (i < n1 && j <n2)
      {
        if (X[i] < Y[j])
          {
            A[k]=X[i]
            i = i + 1
            k = k + 1
          }
        else
          {
            A[k]=Y[j]
            j = j + 1
            k = k + 1
            icount = icount + (mid - i)
          }
      }
    while (i<n1)
      {
        A[k]=X[i]
        i = i + 1
        k = k + 1
      }
    while (j<n2)
      {
        A[k]=Y[j]
        j = j + 1
        k = k + 1
      }
    return icount
}
```

Key ideas

- Inversion Count for an array indicates how far or close the array is from being sorted. **Try to explore this via an example.**
- For already sorted array inversion count = 0 (minimum). For reverse sorted array inversion count = n(n-1)/2 (maximum). **Think and try!**
- We can also solve this problem in O(nlogn) by using Self-Balancing Binary Search tree like AVL tree or Red Black Tree [We will learn AVL Tree in this program]
- What is the time complexity of insertion sort when there are O(n) inversions in an array? **Think and try!**

Problem 9 : Maximum Subarray Sum

Brute force approach : Using three nested loops

- There are n(n-1)/2 pair of (i, j) for i<j.
- Use three nested loops and find the sum for all the subarray A[i….j] for each value of i and j where i<j. During this process, update the maximum subarray sum in each iteration.
- Time Complexity :O(n^3)
- Space Complexity : O(1)

```
int max_Subarray_Sum ( A[] , n)
{
   for ( i = 0 to n-1)
     for ( j = i to n-1)
      {
        sum = 0
        for ( k = i to j)
          sum = sum +  A[k]
        if (sum > maxSum)
          maxSum = sum
      }
 return maxSum
}
```

Better approach than Brute force : Using two nested loops

- You can easily find the sum for A[i...j] in just one operation : A[i...j] = A[i….j-1] + A[j] .There is no need of innermost loop in brute force solution. **[Think!].**
- Use two nested loops and find the sum for all the subarray A[i….j] for each value of i and j where i<j. During this process, update the maximum subarray sum in each iteration.
- Time Complexity :O(n^2)
- Space Complexity : O(1)

```
int max_Subarray_Sum ( A[] , n)
 {
   for ( i = 0 to n-1)
    {
     sum=0
     for(j = i to n-1)
      {
        sum = sum +  A[j]
        if (sum > maxSum)
          maxSum = sum
      }
    }
  return maxSum
 }
```

A very good problem to understand the idea of improving time complexity via different approach!

Continued...

Better Approach via divide and conquer

- T(n)= 2T(n/2) + O(n). Overall time Complexity : O(nlogn)
- Space Complexity : O(logn) [Recursion call stack]
- **Best solution** of this problem is via Dynamic Programming in O(n) time and O(1) space complexity. This is very good example of problem solving, famous as Kadane Algorithm. This is an exercise for you to understand. We Will discuss this problem in problem solving session of DP.

```
int max_Subarray_Sum (A[], l, r)
{
  if (l == r)
    return A[l]          Base case (Think!)
else
    mid = l + (r - l)/2
    left_sum = max_Subarray_Sum (A, l, mid)
    right_sum = max_Subarray_Sum (A, mid+1, r)
    crossing_Sum = max_Crossing_Sum(A, l, mid, r)

  return max( Left_sum,Right_sum, Crossing_Sum )
}
```

- **max crossing subarray sum =** max contiguous sum from mid to l + max contiguous sum from mid+1 to r **(Think!)**

Calculating max contiguous sum from mid to l

Calculating max contiguous sum from mid to r

```
int max_Crossing_Sum(A[], l, mid, r)
{
   sum = 0
   lsum= INT_MIN
   for(l=mid to l)
    {
      sum = sum + A[i]
      If (sum > lsum)
      lsum = sum
    }
sum=0
rsum= INT_MIN
for(i=mid+1 to r)
    {
      sum = sum + A[i]
      If (sum > rsum)
      rsum= sum
    }
return (lsum+rsum)
}
```

- **Divide :** Dividing the array into two equal halves via mid
- **Conquer Part :** Recursively calculate the maximum subarray sum of the left half and right half
- **Combine Part :** Calculate the maximum subarray sum for subarray crossing the both the smaller array of size n/2..Time complexity for this is O(n).**(Think!)**
- **Overall maximum subarray sum** = maximum subarray sum left half +maximum subarray sum right half + maximum subarray sum crossing both the half.

Problem 10 : Finding celebrity Problem

- Suppose we have 2 people P and Q. If we ask the question to P “ if does he/she knows Q?. Then we can have 2 possibilities:

  1] Yes, P knows Q which implies P cannot be the celebrity

  2] No, P doesn’t know Q which implies Q cannot be the celebrity

- This means with every question we can eliminate one person from the group as not being a celebrity. So we ask n-1 questions to eliminate n-1 people from the group.
- For the last person, let’s call R, we need to check if he/she is indeed the celebrity. To do this we ask every person apart from R if they know R (n - 1 questions) and we also ask R if he/she knows anyone else (n-1 questions). If R knows nobody and everyone knows R, then we declare R as celebrity otherwise no celebrity exists.
- So in the worst case, we would require (n-1) + (n-1) + (n-1) = 3n-3 questions.
- A very good algorithmic puzzle to understand the problem solving!
- Think about the brute force approach.

Enjoy Algorithms!

Thank You.