Data Structure and Algorithm Training Program

Week 1 : Pseudo Code Solution Part 2

Problem 1 : Recursive program for finding max value in an array

- Recurrence Relation : T(n)=T(n-1) + c
- Time Complexity : O(n)
- Space Complexity : O(n) [Recursion call stack]

```
int recursive_FindMax(A[] , n )
{
      if (n == 1)
        return A[0]

      max = recursive_FindMax(A , n-1)

      if(max < A[n-1])
          max = A[n-1]
 return max
}
```

Problem 3 : Iterative program for binary search

- Time Complexity : O(logn)
- Space Complexity : O(1)
- At every step of iteration, decreasing the input size by half

```
int binarySearch( A[] , l, r, key )
{
    while (l <= r)
     {
       mid =  l+(r-l) / 2

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

       if (A[mid] < key)
            l = mid + 1
       else
            r = mid - 1
     }
 return -1
}
```

Problem 8 : Recursive program for Insertion sort

- Recurrence Relation : T(n)=T(n-1) + cn
- Time Complexity : O(n^2)
- Space Complexity : O(n) [Recursion call stack]

```
recursive_InsertionSort( A[] , n )
  {

       if (n <= 1)
           return

       recursive_InsertionSort ( A , n-1 )
       key = A[n-1]
       i = n-2
       while (i >= 0 && A[i] > key)
        {
           A[i+1] = A[i]
           i = i-1
        }
     A[i+1] = key
  }
```

Problem 9 : Write a Program to find the median of two sorted arrays A and B of size n after merging both the array

Important details related to the problem

- Both the arrays are sorted
- Size of both the arrays are equal

- median of a sorted array **-** A value which divides the whole array into two equal size.
  If n is odd, median = Value at n/2 position.
  If n is even, median = Average of Value at n/2-1 and n/2 position

- Size of the sequence after merging both the array would be 2n which is even. So output should be average of value at (n-1)th and (n)th position

Brute force Solution

- Time Complexity : O(n)
- Space Complexity : O(n)
- Using the merge approach of merge sort

<u>Solution Steps</u>

- Take Extra memory C[] of size 2n
- Merge A and B into a larger array C[]
- Return (C[n-1]+C[n])/2

Improved Version of Brute force solution

- Time Complexity : O(n)
- Space Complexity : O(1)[In place]
- Track n-1th and nth value via k, m1 and m2

```
int arrayMedian(A[] , B[] , n)
  {
     if ( A[n-1] < B[0] )
          return ( A[n-1] + B[0]) / 2
     else if ( B[n-1] < A[0] )
          return ( A[0] + B[n-1]) / 2
     else
       {
          i=0, j=0, k=0
          while(k < n-1)
           {
              if (A[i]<B[j])
               {
                  i = i+1
                  k = k+1
               }
             else
               {
                  j = j+1
                  k = k+1
               }
          }
          if (A[i]>B[j])
            {
               m1=B[j]
               j = j+1
            }
           else
            {
               m1 = A[i]
               i = i+1
            }

          if (A[i] > B[j])
               m2 = B[j]
          else
               m2 = A[i]

          return (m1+m2)/2
       }
  }
```

Corner case (Think!)

Customized loop of merging two sorted array where Incrementing the loop till k reach n-1

find first middle value m1

find second middle value m2

Continued...

Efficient Pseudo Code : Divide and Conquer Approach of binary search

- Time Complexity : O(logn)
- Space Complexity : O(logn) [Recursion Call Stack]
- After every comparison with median of both the arrays, decreasing the input size by half
- Boundary conditions : Odd and even values of n
- After comparison, include the medians of both the array in recursive call because both can be part of the solution!
- Base case : n=2, n=1 and n=0

Base Cases

Divide

Conquer

```
int arrayMedian(A[] , B[] , n)
{
   if (n <= 0)
      return -1
   if (n == 1)
      return ( A[0] + B[0] ) / 2
   if (n == 2)
      return (max( A[0] , B[0] )+
                        min( A[1],  B[1]  )/ 2
   m1 = findMedian( A , n )
   m2 = findMedian( B , n)
   if (m1 == m2)
        return m1
   if (m1 < m2)
     {
      if (n%2 == 0)
          return arrayMedian
                  (A+n/2-1, B , n/2+1)
      else
          return  arrayMedian
                  (A+n/2, B , n/2)
      }
   else
     {
       if (n%2 == 0)
         return arrayMedian
                  (A, B+n/2-1, n/2+1)
       else
        return arrayMedian
                  (A, B+n/2 , n/2)
     }
}
```

```
int findMedian(C[],n)
{
   if (n%2 == 0)
      return (C[n/2] + C[n/2-1])/ 2
   else
      return C[n/2]
}
```

Similar Question : median of two sorted arrays A and B of different size m and n after merging both the array

Problem 10: Find position of an element in a sorted array of infinite numbers

Brute force approach : Linear search from the start

- if A[i]==key, return i
- Time Complexity : O(i)
- Space Complexity : O(1)

Improved Version of Brute force solution : increment the interval size by some constant c

- Track the interval where the key would be present
- Increment the interval size by some constant c = O(1)
- At every iteration, compare the value present at right end with key. If key > A[r],then increment the interval size by c
- After finding the interval, apply the binary search in interval
- Time complexity = interval search + Binary Search = O(i/c) + O(logc) = O(i) + O(1) = O(i)
  [Because c is some constant Value]
- Space Complexity = O(1)

```
int search_InfiniteArray(int A[] , key )
{
      l = 0
      r = c
     while (A[r]< key)
      {
        l = r
        r = r+c
      }
   return binarySearch(A, l, r, key)
}
```

Incrementing the loop by some constant order

Efficient Solution : Approach of exponential search

- At every step, increment the interval size by double
- After finding the interval range after kth step
  Lower bound of interval = 2^k-1
  Uppar bound of interval = 2^k
  Interval Size/ Total Values in Interval = 2^k-1
- Time complexity = interval search + Binary Search = O(logi) + O(logi) = O(logi)
  [Here i is the position of key in sorted array]
- Space Complexity = O(1)

```
int search_InfiniteArray(A[] , key)
{
    l = 0
    r = 1
    while (A[r]< key)
     {
       l = r
       r = 2*r
     }
  return binarySearch(A, l, r, key)
}
```

Incrementing the loop by exponential order

Similar Problems
1] Find the occurrence of first 1 in an infinite sorted array of 0s and 1s
2] Find the point where a monotonically increasing function becomes positive first time

Enjoy Algorithms!

Thank You.