# Solution of Homework 2

**Solution 1:**

Algorithm:

The key observation is the following.
In a stack the elements are stored in last in first out order from the top of the stack.
However, if we pop the elements in stack1 one by one and push them onto stack2, then the elements are stored in the stack 2 in a first in first out order from the top of the stack.
This is why two stacks re-order the elements in first in first out order like a queue.

The key idea of the algorithm is to use stack1 for en_queue, and the stack2 for de_queue. When en_queue a new element, we will push the element onto stack 1.
When de_queue an element, we will pop stack2.
If stack 2 is empty, we will pop all elements from stack 1 and push them onto stack 2.

The running time for the en_queue and de_queue operation is O(1) amortized.

**Solution 2**

Copy all the values of H1 and H2 to array H of size m+n.
This takes O(m+n) time.

```
for(j=(m+n)/2 to 1)
     Heapify(H[j]);
```

This will take O(m+n) time and turn H into a heap.

problem-3:

Ans:

For a balanced binary search tree, let $h$ be the height of the tree and there are $n$ nodes. Then left subtree has $2^{h-1}$ nodes and right subtree has $2^{h-2}$ nodes. So we can write —

$$n \geq 1 + 2^{h-1} + 2^{h-2}$$

$$\geq 1 + \frac{2^h}{2} + \frac{2^h}{4}$$

$$\Rightarrow n > \frac{3 \cdot 2^h}{4}$$

$$\Rightarrow n > 2^{h/2}$$

$$\Rightarrow \log n > h/2$$

$$\Rightarrow h < 2 \log n$$

Therefore we can prove that $h$ is bounded by $2\log n$.

**Solution 4**

problem-4:

Ans:

As many elements have duplicates and there are only $k << n$ distincts so duplicate number is an important factor here.

In insertion sort, worst case scenerio is $O(n^2)$. So we don't consider insertion sort. Other 3 (heapsort, mergesort and BST) has complexity of $O(n \log n)$

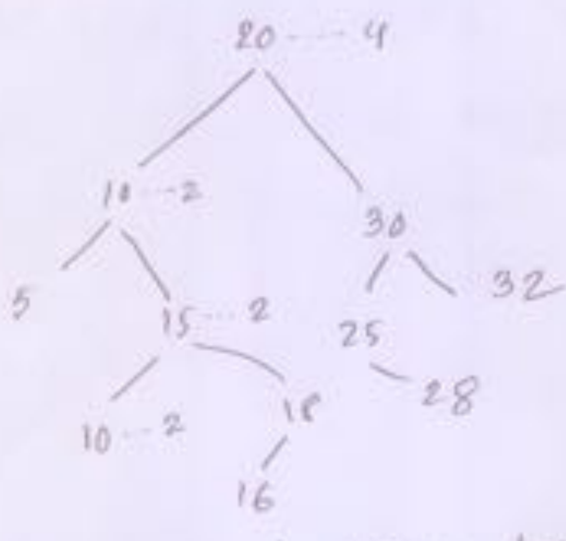
In respect of duplicate or no duplicates mergesort will always take $O(n \log n)$.

For heapsort it takes $n \log n$.

For a binary search tree we can modify modify the algorithm and use extra memory for duplicates and rearrange the tree building.

For the following values : 20, 10, 30, 5, 15, 20, 32, 10, 15, 12, 16, 18, 20, 25, 20, 28

We can rearrange the BST while building.




So the level is lower, height is less than the original tree and number of nodes is K. In this case running time is O(K log K) to sort the tree. so in this case BST sort is preferable prob: 5

Ans:

Let $n = 2k+1$

$P = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$

$Q = b_0 + b_1x + b_2x^2 + \cdots + b_nx^n$

$P = a_0 + a_2x^2 + \cdots + a_{n-1}x^{n-1}$

$+ a_1x + a_3x^3 + \cdots + a_nx^n$

$= (a_0 + a_2x^2 + \cdots + a_{n-1}x^{n-1})$

$+$

$x(a_1 + a_3x^2 + \cdots + a_nx^n)$

Let, $y = x^2$

So, $P = (a_0 + a_1y + \cdots + a_{n-1}y^k) +$

$x\ (a_1 + a_3y + \cdots + a_ny^k)$

$= P_0(y) + x\,P_1(y)$

Similarly, $Q = Q_0(y) + x\,Q_1(y)$

$PQ = P_0Q_0 + (P_0Q_1 + P_1Q_0)x + P_1Q_1(y)$

since, $p_0q_1 + p_1q_2 = (p_0+p_1)(q_0+q_1) - p_0q_0 - p_1q_1$

we can reduce the multiplication of 2 n-degree polynomials of $x$ to $3 \frac{n}{2}$ degree.

polynomials of $y = x^2$

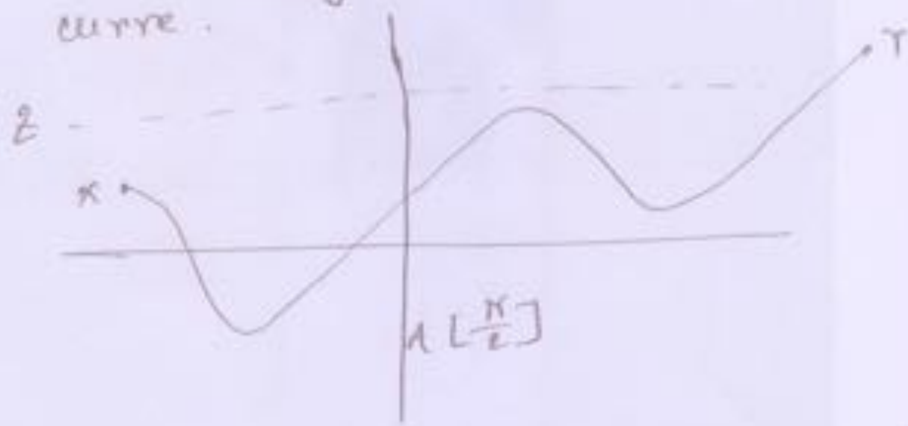
we end up with the recursion,

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

**Solution 6**

prob-6

Ans: The array can be viewed as a continuous curve.




Suppose we go to the middle $A[\frac{N}{2}]$.
if $A[\frac{N}{2}] < z$ then because of the continuity there has to be an entry from $A[\frac{n}{2}+1, \ldots n]$ whose value is $z$.

This is very similar to binary search and will take $O(\log n)$ time to find the entry whose value is $z$.

**Solution 7**

The key here is to output an element once its most significant digit is processed.

Input: An array A[] of N variable length integers.

**Algorithm**:

1. Create an array of 10 queues(b0,b1,b2...b8,b9) called buckets.
2. Create an array sortedResult[] to store data.
3. for (int i = 0; i > N; i++)

```
        for (int j = 0; j < A.length; j++)
       {

         int digit = A[j][i];
         buckets[digit].Enqueue(A[j]);

       }
           A = {};
    for (int k = 0; k< 10; k++)
         {
         while (buckets[k].Count > 0)
         {
              string data = buckets[k].Dequeue();
                   if(data.length>i+1)
              {
                     A.add(data);
                   }
                   else
                   {
                     sortedResult.add(data);
                   }

         }
    }
```

**Running time**:

For this algorithm, the running time will be O(A.length x N). If the A.length is very large and Number of digits N is small then we can write running time O(A.length) which is linear.

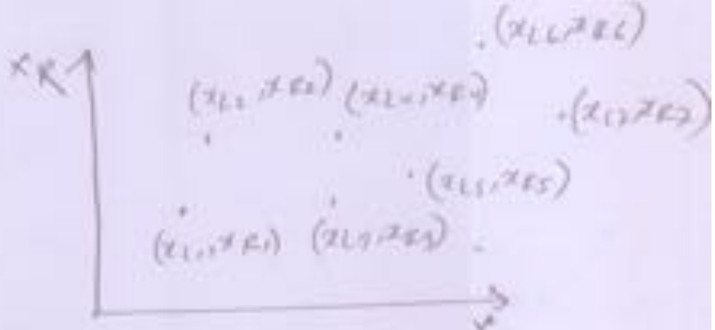
prob - 8

Ans: suppose we have a set of intervals:

$$(x_L, x_R) = \{(x_{L_1}, x_{R_1}), (x_{L_2}, x_{R_2}), (x_{L_3}, x_{R_3}) \cdots (x_{Ln}, x_{Rn})\}$$

we need to design an algorithm to identify all intervals that are contained in another interval from the set.

we can interpret these set of intervals in 2D plane.




The interval that includes most of the interval should remain in the leftmost and topmost manner.

For a coordinate $(x_p, x_q)$ we need to find the longest increasing subsequence of $x_L$ where all other elements $x_R$

is less than $x_q$. For finding the longest increasing subsequence in $O(n \log n)$ time we need to use a BestEnd[] and Binary Search ().

**Algorithm:**

```
FindIntervalThatIncludesAllInterval()
{

      1. Create arrays L[] and parent[], set L[0]=0, parent[0]=-1;
      2. X_L = { x_L1, x_L2, ....., x_Ln}, X_R = { x_R1, x_R2, ....., x_Rn};
      3. Create BestEndX[]
      4. Create BestEndY[]
      5. index = 1;
      6. BestEndX[1] = X_L [1];
      7. BestEndY[1] = X_R[1];

            for (int i = 1; i < X_L.len; i++)
            {
                if (X_L [i] < BestEndX[1] && X_R [i] > BestEndY[1])
                {
                    BestEndX[1] = X_L[i];
                    BestEndY[1] = X_R[i];
                    L[i] = 1;
                }
                else if (X_L [i] >BestEndX[index] &&
                               X_R[i]<BestEndY[index])
                {
                    BestEndX[index + 1] = X_L[i];
                    BestEndY[index + 1] = X_R[i];
                    L[i] = index + 1;
                    index++;
                }
                else if (X_R[i] >BestEndY[index])
```

```
        {
            int k = binary_search(BestEndX,BestEndY, index,
                                    XL[i],XR[i]);
            BestEndX[k] = XL[i];
            BestEndY[k] = XR[i];
            L[i] = k;
        }
    }
```

```
binary_search(int[] BestEndX, int[] BestEndY, int index, int pLeft,
int pRight)
    {
        int min = 1, max = index, k = max;
        while (min <=max)
        {
            int mid = (min + max) / 2;
            if (pLeft == BestEndX[mid]&&pRight == BestEndY[mid])
            {
                k = mid;
                return k;
            }
            else if (pLeft>BestEndX[mid]&&pRight<BestEndY[mid])
            {
                max = mid - 1;
                k = mid;
            }
            else if(pRight<BestEndY[mid])
            {
                min = mid + 1;
                k = min;
            }
        }
        return k;
    }
```

**Running Time:**

This algorithm running time O(nlogn)

**Solution 9**

prob-9

Ans: 9

For $A = \{9, 2, 5, 3, 7, 11, 8, 10, 13, 6\}$

the longest monotonically subsequence is,

$\{2, 3, 7, 8, 10, 13\}$

To find out the longest increasing subsequence, we need to maintain length array L[] that tracks current length of each element in the sequence.

Algo:

A[]: → data array

L[]: → length "

parent[]:→ stores parent/previous element of an element in a sequence

data[]:

Set $L[0] = 0$

```
For each (element in A[])
    data [i+1] = A [i]
    data [0] = 0;
    for (i = 1, i < data.Length ; i++)
    { for (j=0; j<i; j++)
        { if (data[i] > data[j] &&
              (L[j]+1) > L[i]) )
          { L[i] = L[j]+1);
            parent[i] = j;
          }
        }
    }
```

Now in L[] ; the index which has the max value is the last element in the longest increasing subsequence. By using the parent [] we can easily print the longest increasing subsequence.

This algorithm runs on $O(n^2)$.

Example:

For $A = \langle 1, 5, 3, 6 \rangle$

$data = \langle 0, 1, 5, 3, 6 \rangle$

when $i=1$, we get

$L[\,] = \langle 0, 1 \rangle$

$i=2$, $L[\,] = \langle 0, 1, 2 \rangle$

$i=3$ $L[\,] = \langle 0, 1, 2, 2 \rangle$

$i=4$ $L[\,] = \langle 0, 1, 2, 2, 3 \rangle$

$\therefore$ $L[4]$ has the max value and parent $[4]=5$, using parent $[\,]$ we can get $\langle 1, 5, 6 \rangle$.

**Solution 10**

Recall in the one-knapsack problem. For the item $S_N$ we have to make a decision to include it in the knapsack or not. When there are two knapsacks, we will have to make a decision of whether to include the item $s_N$ in the $1^{st}$ Knapsack, or the $2^{nd}$ Knapsack, or not to use it. This means, instead of building a 2D array as the one knapsack problem, we need to build a 3-dimensional array to solve this problem of size $nK^2$, where each entry indicate the solution to the instance $(i, k_1, k_2)$, i.e., the optimal solution to the two knapsack problem, with the $1^{st}$ knapsack of size $k_1$ and the $2^{nd}$ knapsack of size $k_2$ and i items.

```
int[,,] kp = new int[n+1, K+1, K+1];


for (int i = 1; i <= n; i++)
    for (int j = 1; j <= K; j++)
        for (int k = 1; k <= K; k++)
        {
          kp[i, j, k] = max(kp[i - 1, j, k],
                     kp[i - 1, j - weights[i], k] + values[i],
                     kp[i - 1, j, k-weights[i]] +values[i]);
                  }
              }
          }
```