Dynamic Programming | Set 3 (Longest Increasing Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 andSet 2 respectively.

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.
The longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, length of LIS for { 10, 22, 9, 33, 21, 50, 41, 60, 80 } is 6 and LIS is {10, 22, 33, 50, 60, 80}.

Optimal Substructure:
Let arr[0..n-1] be the input array and L(i) be the length of the LIS till index i such that arr[i] is part of LIS and arr[i] is the last element in LIS, then L(i) can be recursively written as.
L(i) = { 1 + Max (L(j)) } where j < i and arr[j] < arr[i] and if there is no such j then L(i) = 1
To get LIS of a given array, we need to return max(L(i)) where 0 < i < n
So the LIS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

Overlapping Subproblems:
Following is simple recursive implementation of the LIS problem. The implementation simply follows the recursive structure mentioned above. The value of lis ending with every element is returned using max_ending_here. The overall lis is returned using pointer to a variable max.

```
/* A Naive recursive implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* To make use of recursive calls, this function must return two things:
   1) Length of LIS ending with element arr[n-1]. We use max_ending_here
      for this purpose
   2) Overall maximum as the LIS may end with an element before arr[n-1]
      max_ref is used this purpose.
The value of LIS of full array of size n is stored in *max_ref which is our final
*/
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if(n == 1)
        return 1;

    int res, max_ending_here = 1; // length of LIS ending with arr[n-1]

    /* Recursively get all LIS ending with arr[0], arr[1] ... ar[n-2]. If
       arr[i-1] is smaller than arr[n-1], and max ending with arr[n-1] needs
       to be updated, then update it */
    for(int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall max. And update the
    // overall max if needed
    if (*max_ref < max_ending_here)
       *max_ref = max_ending_here;
```

```
    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}

// The wrapper function for _lis()
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

    // The function _lis() stores its result in max
    _lis( arr, n, &max );

    // returns max
    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n",  lis( arr, n ));
    getchar();
    return 0;
}
```

Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[].

```
                      lis(4)
                   /     |     \
          lis(3)       lis(2)    lis(1)
         /     \         /
   lis(2)  lis(1)   lis(1)
   /
lis(1)
```

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabluated implementation for the LIS problem.

```
/* Dynamic Programming implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* lis() returns the length of the longest increasing subsequence in
    arr[] of size n */
int lis( int arr[], int n )
{
   int *lis, i, j, max = 0;
   lis = (int*) malloc ( sizeof( int ) * n );

   /* Initialize LIS values for all indexes */
   for ( i = 0; i < n; i++ )
      lis[i] = 1;

   /* Compute optimized LIS values in bottom up manner */
   for ( i = 1; i < n; i++ )
      for ( j = 0; j < i; j++ )
         if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
            lis[i] = lis[j] + 1;
```

```
   /* Pick maximum of all LIS values */
   for ( i = 0; i < n; i++ )
      if ( max < lis[i] )
         max = lis[i];

   /* Free memory to avoid memory leak */
   free( lis );

   return max;
}

/* Driver program to test above function */
int main()
{
  int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Length of LIS is %d\n", lis( arr, n ) );

  getchar();
  return 0;
}
```

Note that the time complexity of the above Dynamic Programmig (DP) solution is O(n^2) and there is a O(nLogn) solution for the LIS problem (see this). We have not discussed the nLogn solution here as the purpose of this post is to explain Dynamic Programmig with a simple example.