Printing Items in 0/1 Knapsack & Unbounded Knapsack
Temple Offerings & Egg Dropping Puzzle
Dice Throw & Word Break Problem
Vertex Cover Problem & Tile Stacking Problem
Box Stacking Problem
Highway Billboard Problem & Largest Independent Set Problem
Print equal sum sets of array (Partition problem) | Set 1 & Set 2
Longest Bitonic Subsequence
Printing Longest Bitonic Subsequence
Print Longest Palindromic Subsequence
Longest palindrome subsequence with O(n) space
Explore more...
Hard Problems
Palindrome Partitioning & Word Wrap Problem
Mobile Numeric Keypad Problem & The painter's partition problem
Boolean Parenthesization Problem & Bridge and Torch problem
A Space Optimized DP solution for 0-1 Knapsack Problem
Matrix Chain Multiplication & Printing brackets in Matrix Chain Multiplication Problem
Number of palindromic paths in a matrix
Largest rectangular sub-matrix whose sum is 0
Largest rectangular sub-matrix having sum divisible by k
Maximum sum bitonic subarray
K maximum sums of overlapping contiguous sub-arrays
Maximum profit by buying and selling a share at most k times
Maximum points from top left of matrix to bottom right and return back
Check whether row or column swaps produce maximum size binary sub-matrix with all 1s
Minimum number of elements which are not part of Increasing or decreasing subsequence in array
Count ways to increase LCS length of two strings by one
Count of AP (Arithmetic Progression) Subsequences in an array
Explore more...

Longest Increasing Subsequence | DP-3

We have discussed Overlapping Subproblems and Optimal Substructure properties.

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, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.

arr[ ]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

More Examples:

```
Input  : arr[] = {3, 10, 2, 1, 20}
Output : Length of LIS = 3
The longest increasing subsequence is 3, 10, 20

Input  : arr[] = {3, 2}
Output : Length of LIS = 1
The longest increasing subsequences are {3} and {2}
```

```
Input : arr[] = {50, 3, 10, 7, 40, 80}
Output : Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Optimal Substructure:

Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

Then, L(i) can be recursively written as:

L(i) = 1 + max(L(j)) where 0 < j < i and arr[j] < arr[i]; or

L(i) = 1, if no such j exists.

To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.

Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

Following is a simple recursive implementation of the LIS problem. It follows the recursive structure discussed above.

C/C++

```
/* A Naive C/C++ 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 result */
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if (n == 1)
        return 1;

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

    /* Recursively get all LIS ending with arr[0], arr[1] ...
       arr[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 %dn",
           lis( arr, n ));
    return 0;
}
```

Run on IDE

Java

```
/* A Naive Java Program for LIS Implementation */
class LIS
{
   static int max_ref; // stores the LIS

   /* 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 result */
static int _lis(int arr[], int n)
{
    // base case
    if (n == 1)
        return 1;

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

     /* Recursively get all LIS ending with arr[0], arr[1] ...
        arr[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);
         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()
 static int lis(int arr[], int n)
 {
     // The max variable holds the result
      max_ref = 1;

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

     // returns max
     return max_ref;
 }

 // driver program to test above functions
 public static void main(String args[])
 {
     int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
     int n = arr.length;
     System.out.println("Length of lis is "
                        + lis(arr, n) + "\n");
    }
 }
/*This code is contributed by Rajat Mishra*/
```

Run on IDE

Python

```
# A naive Python implementation of LIS problem

""" 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 result """

# global variable to store the maximum
global maximum

def _lis(arr , n ):

    # to allow the access of global variable
    global maximum

    # Base Case
    if n == 1 :
        return 1

    # maxEndingHere is the length of LIS ending with arr[n-1]
    maxEndingHere = 1

    """Recursively get all LIS ending with arr[0], arr[1]..arr[n-2]
       IF arr[n-1] is maller than arr[n-1], and max ending with
       arr[n-1] needs to be updated, then update it"""
    for i in xrange(1, n):
        res = _lis(arr , i)
        if arr[i-1] < arr[n-1] and res+1 > maxEndingHere:
            maxEndingHere = res +1

    # Compare maxEndingHere with overall maximum. And
    # update the overall maximum if needed
    maximum = max(maximum , maxEndingHere)

    return maxEndingHere

def lis(arr):

    # to allow the access of global variable
```

```
    global maximum

    # lenght of arr
    n = len(arr)

    # maximum variable holds the result
    maximum = 1

    # The function _lis() stores its result in maximum
    _lis(arr , n)

    return maximum
# Driver program to test the above function
arr = [10 , 22 , 9 , 33 , 21 , 50 , 41 , 60]
n = len(arr)
print "Length of lis is ", lis(arr)

# This code is contributed by NIKHIL KUMAR SINGH
```

Run on IDE

```
Length of lis is 5
```

Overlapping Subproblems:

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.

C/C++

```
/* Dynamic Programming C/C++ 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 %dn", lis( arr, n ) );
    return 0;
}
```

Run on IDE

Java

```
/* Dynamic Programming Java implementation of LIS problem */

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

          /* 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];

           return max;
    }

    public static void main(String args[])
    {
        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
            int n = arr.length;
            System.out.println("Length of lis is " + lis( arr, n ) + "n" );
    }
}
/*This code is contributed by Rajat Mishra*/
```

Run on IDE

Python

```
# Dynamic programming Python implementation of LIS problem

# lis returns length of the longest increasing subsequence
# in arr of size n
def lis(arr):
    n = len(arr)

    # Declare the list (array) for LIS and initialize LIS
    # values for all indexes
    lis = [1]*n

    # Compute optimized LIS values in bottom up manner
    for i in range (1 , n):
        for j in range(0 , i):
            if arr[i] > arr[j] and lis[i]< lis[j] + 1 :
                lis[i] = lis[j]+1

    # Initialize maximum to 0 to get the maximum of all
    # LIS
    maximum = 0

    # Pick maximum of all LIS values
    for i in range(n):
        maximum = max(maximum , lis[i])

    return maximum
# end of lis function

# Driver program to test above function
arr = [10, 22, 9, 33, 21, 50, 41, 60]
print "Length of lis is", lis(arr)
# This code is contributed by Nikhil Kumar Singh
```

Run on IDE

Output:

```
Length of lis is 5
```

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

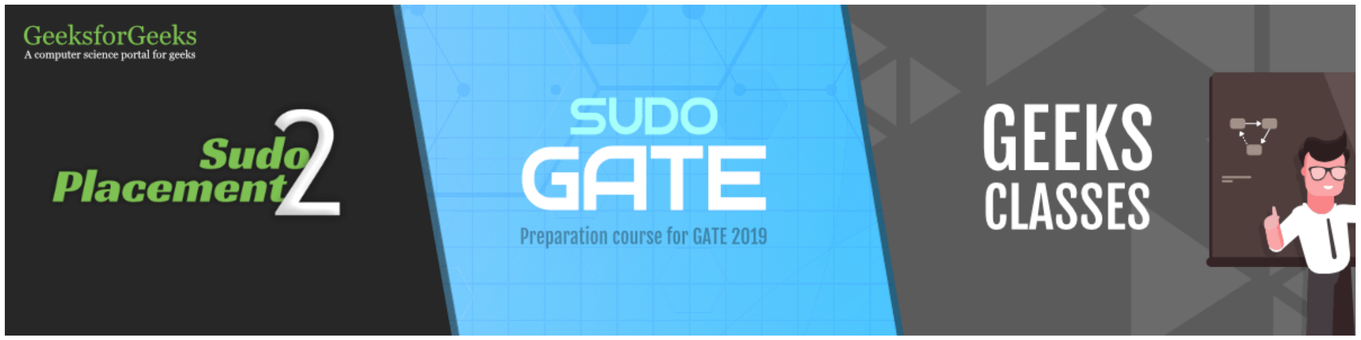
Longest Increasing Subsequence Size (N log N)

- Printing LIS of array
- Recent articles based on LIS!

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Practice Tags : Dynamic Programming

Article Tags : Dynamic Programming LIS



Improve this Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Recommended Posts:

Longest increasing subarray
Longest Common Subsequence | DP-4
Min Cost Path | DP-6
Edit Distance | DP-5
Maximum Sum Increasing Subsequence | DP-14
Overlapping Subproblems Property in Dynamic Programming | DP-1
Dynamic Programming vs Divide-and-Conquer
Paytm Interview experience for FTE (On-Campus)
Balanced expressions such that given positions have opening brackets | Set 2
Number of ways a convex polygon of n+2 sides can split into triangles by connecting vertices

Logged in as **Dragan Nikolic**(Logout)

3.1 Average Difficulty : **3.1/5.0**
Based on **488** vote(s)

Basic Easy Medium Hard Expert

Add to TODO List
Mark as DONE

Give Feedback | Add your Notes | Improve Article

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments | Share this post!