Dynamic Programming | Set 1 (Overlapping Subproblems Property)

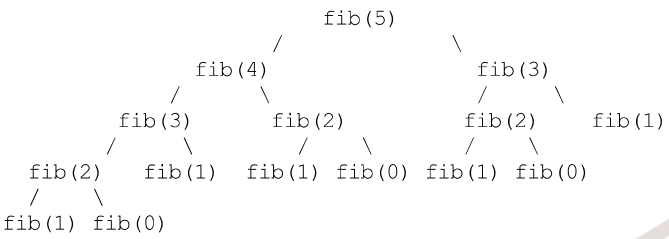
Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Subproblems
2) Optimal Substructure

1) Overlapping Subproblems:
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these dont have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesnt have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}
```

Recursion tree for execution of *fib(5)*



We can see that the function f(3) is being called 2 times. If we would have stored the value of f(3), then instead of computing it again, we would have reused the old stored value. There are following two different ways to store the values so that these values can be reused.

a) Memoization (Top Down):
b) Tabulation (Bottom Up):

a) Memoization (Top Down): The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

```
/* Memoized version for nth Fibonacci number */
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{
  int i;
  for (i = 0; i < MAX; i++)
    lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
   if(lookup[n] == NIL)
   {
    if ( n <= 1 )
      lookup[n] = n;
    else
      lookup[n] = fib(n-1) + fib(n-2);
   }
```

Aman Barnwal

```
   return lookup[n];
}

int main ()
{
  int n = 40;
  _initialize();
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  return 0;
}
```

b) Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table.

```
/* tabulated version */
#include<stdio.h>
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}

int main ()
{
  int n = 9;
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  return 0;
}
```

Both tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in tabulated version, starting from the first entry, all entries are filled one by one. Unlike the tabulated version, all entries of the lookup table are not necessarily filled in memoized version. For example, memoized solution of LCS problem doesnt necessarily fill all entries.

To see the optimization achieved by memoized and tabulated versions over the basic recursive version, see the time taken by following runs for 40th Fibonacci number.

Simple recursive program
Memoized version
tabulated version

Also see method 2 of Ugly Number post for one more simple example where we have overlapping subproblems and we store the results of subproblems.

We will be covering Optimal Substructure Property and some more example problems in future posts on Dynamic Programming.

Try following questions as an exercise of this post.
1) Write a memoized version for LCS problem. Note that the tabular version is given in the CLRS book.
2) How would you choose between Memoization and Tabulation?

References:
http://www.youtube.com/watch?v=V5hZoJ6uK-s

Aman Bansal

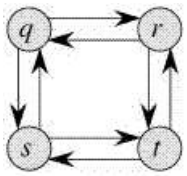
Dynamic Programming | Set 2 (Optimal Substructure Property)

As we discussed in Set 1, following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Subproblems
2) Optimal Substructure

We have already discussed Overlapping Subproblem property in the Set 1. Let us discuss Optimal Substructure property here.

2) Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
For example the shortest path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like FloydWarshall and BellmanFord are typical examples of Dynamic Programming.
On the other hand the Longest path problem doesnt have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the CLRS book. There are two longest paths from q to t: q -> r ->t and q ->s->t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q->r->t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q->s->t->r.



We will be covering some example problems in future posts on Dynamic Programming.

References:
http://en.wikipedia.org/wiki/Optimal_substructure
CLRS book

Aman Barnwal

Dynamic Programming | Set 3 (Longest Increasing Subsequence)

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

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()
```

Aman Barnwal

```
{
    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 ));
    return 0;
}
```

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 , 41 , 60]
n = len(arr)
print "Length of LIS is ", lis(arr)
# This code is contributed by NIKHIL KUMAR SINGH
```

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

Aman Barnwal

```
  /
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 %d\n", lis( arr, n ) );
  return 0;
}
```

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
```

Aman Barnwal

```
# 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
```

```
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)

Aman Barnwal

Dynamic Programming | Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 and Set 2 respectively. We also discussed one example problem in Set 3. Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, abc, abg, bdf, aeg, acefg, .. etc are subsequences of abcdefg. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:
LCS for input Sequences ABCDGH and AEDFHR is ADH of length 3.
LCS for input Sequences AGGTAB and GXTXAYB is GTAB of length 4.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

1) Optimal Substructure:
Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Examples:
1) Consider the input strings AGGTAB and GXTXAYB. Last characters match for the strings. So length of LCS can be written as:
L(AGGTAB, GXTXAYB) = 1 + L(AGGTA, GXTXAY)

2) Consider the input strings ABCDGH and AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:
L(ABCDGH, AEDFHR) = MAX (L(ABCDG, AEDFHR), L(ABCDGH, AEDFH))

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

2) Overlapping Subproblems:
Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

C/C++

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

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
     return 0;
   if (X[m-1] == Y[n-1])
     return 1 + lcs(X, Y, m-1, n-1);
   else
     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
```

Aman Barnwal

```
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);

  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  return 0;
}
```

Python

```
# A Naive recursive Python implementation of LCS problem

def lcs(X, Y, m, n):

    if m == 0 or n == 0:
       return 0;
    elif X[m-1] == Y[n-1]:
       return 1 + lcs(X, Y, m-1, n-1);
    else:
       return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));


# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X , Y, len(X), len(Y))



Length of LCS is 4
```

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.
Considering the above implementation, following is a partial recursion tree for input strings AXYT and AYZX

```
                         lcs("AXYT", "AYZX")
                        /                  \
                lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
                /                \                  /                \
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

In the above partial recursion tree, lcs(AXY, AYZ) is being solved twice. If we draw the complete recursion tree, then 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 tabulated implementation for the LCS problem.

C/C++

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

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   int L[m+1][n+1];
   int i, j;

   /* Following steps build L[m+1][n+1] in bottom up fashion. Note
      that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
   for (i=0; i<=m; i++)
   {
     for (j=0; j<=n; j++)
     {
       if (i == 0 || j == 0)
         L[i][j] = 0;

       else if (X[i-1] == Y[j-1])
         L[i][j] = L[i-1][j-1] + 1;

       else
         L[i][j] = max(L[i-1][j], L[i][j-1]);
     }
   }
```

Aman Barnwal

```
   /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
   return L[m][n];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);

  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  return 0;
}
```

Python

```
# Dynamic Programming implementation of LCS problem

def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
#end of function lcs


# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X, Y)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.
Printing Longest Common Subsequence

References:
http://www.youtube.com/watch?v=V5hZoJ6uK-s
http://www.algorithmist.com/index.php/Longest_Common_Subsequence
http://www.ics.uci.edu/~eppstein/161/960229.html
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

Aman Barnwal

Dynamic Programming | Set 5 (Edit Distance)

Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert str1? into str2?.

a. Insert
b. Remove
c. Replace

All of the above operations are of equal cost.

Examples:

```
Input:   str1 = "geek", str2 = "gesek"
Output:  1
We can convert str1 into str2 by inserting a 's'.

Input:   str1 = "cat", str2 = "cut"
Output:  1
We can convert str1 into str2 by replacing 'a' with 'u'.

Input:   str1 = "sunday", str2 = "saturday"
Output:  3
Last three and first characters are same.  We basically
need to convert "un" to "atur".  This can be done using
below three operations.
Replace 'n' with 'r', insert t, insert a
```

What are the subproblems in this case?
The idea is process all characters one by one staring from either from left or right sides of both strings.
Let we traverse from right corner, there are two possibilities for every pair of character being traversed.

```
m: Length of str1 (first string)
n: Length of str2 (second string)
```

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on str1?, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
    a. Insert: Recur for m and n-1
    b. Remove: Recur for m-1 and n
    c. Replace: Recur for m-1 and n-1

Below is C++ implementation of above Naive recursive solution.

C++

```
// A Naive recursive C++ program to find minimum number
// operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
   return min(min(x, y), z);
}

int editDist(string str1 , string str2 , int m ,int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0) return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0) return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1[m-1] == str2[n-1])
        return editDist(str1, str2, m-1, n-1);

    // If last characters are not same, consider all three
```

Aman Barnwal

```
    // operations on last character of first string, recursively
    // compute minimum cost for all three operations and take
    // minimum of three values.
    return 1 + min ( editDist(str1,  str2, m, n-1),    // Insert
                     editDist(str1,  str2, m-1, n),   // Remove
                     editDist(str1,  str2, m-1, n-1) // Replace
                   );
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDist( str1 , str2 , str1.length(), str2.length());

    return 0;
}
```

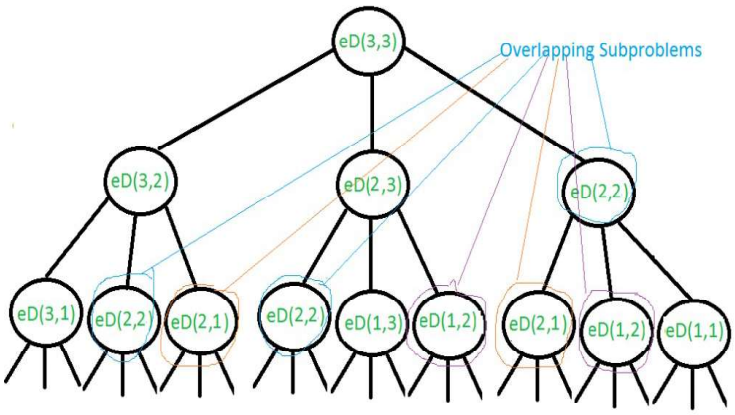
Python

```
# A Naive recursive Python program to fin minimum number
# operations to convert str1 to str2
def editDistance(str1, str2, m , n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m==0:
         return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n==0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1]==str2[n-1]:
        return editDistance(str1,str2,m-1,n-1)

    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1),    # Insert
                   editDistance(str1, str2, m-1, n),    # Remove
                   editDistance(str1, str2, m-1, n-1)    # Replace
                   )

# Driver program to test the above function
str1 = "sunday"
str2 = "saturday"
print editDistance(str1, str2, len(str1), len(str2))

# This code is contributed by Bhavya Jain
```

```
3
```

The time complexity of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.

Aman Barnwal



Worst case recursion tree when m = 3, n = 3.
Worst case example str1="abc" str2="xyz"

We can see that many subproblems are solved again and again, for example eD(2,2) is called three times. Since same suproblems are called again, this problem has Overlapping Subprolems property. So Edit Distance problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subpriblems.

C++

```
// A Dynamic Programming based C++ program to find minimum
// number operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m+1][n+1];

    // Fill d[][] in bottom up manner
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // isnert all characters of second string
            if (i==0)
                dp[i][j] = j;  // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            // If last character are different, consider all
            // possibilities and find minimum
            else
                dp[i][j] = 1 + min(dp[i][j-1],  // Insert
                                   dp[i-1][j],  // Remove
                                   dp[i-1][j-1]); // Replace
        }
    }

    return dp[m][n];
}

// Driver program
int main()
{
```

Aman Barnwal

```
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDistDP(str1, str2, str1.length(), str2.length());

    return 0;
}
```

Python

```
# A Dynamic Programming based Python program for edit
# distance problem
def editDistDP(str1, str2, m, n):
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    # Fill d[][] in bottom up manner
    for i in range(m+1):
        for j in range(n+1):

            # If first string is empty, only option is to
            # isnert all characters of second string
            if i == 0:
                dp[i][j] = j    # Min. operations = j

            # If second string is empty, only option is to
            # remove all characters of second string
            elif j == 0:
                dp[i][j] = i    # Min. operations = i

            # If last characters are same, ignore last char
            # and recur for remaining string
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]

            # If last character are different, consider all
            # possibilities and find minimum
            else:
                dp[i][j] = 1 + min(dp[i][j-1],        # Insert
                                   dp[i-1][j],        # Remove
                                   dp[i-1][j-1])    # Replace

    return dp[m][n]

# Driver program
str1 = "sunday"
str2 = "saturday"

print(editDistDP(str1, str2, len(str1), len(str2)))
# This code is contributed by Bhavya Jain
```

Output:

```
3
```

Time Complexity: O(m x n)
Auxiliary Space: O(m x n)

Applications: There are many practical applications of edit distance algorithm, refer Lucene API for sample. Another example, display all the words in a dictionary that are near proximity to a given word\incorrectly spelled word.

Thanks to Vivek Kumar for suggesting above updates.

Aman Barnwal

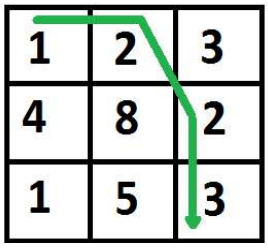
Dynamic Programming | Set 6 (Min Cost Path)

Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns cost of minimum cost path to reach (m, n) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach (m, n) is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1) and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.

For example, in the following figure, what is the minimum cost path to (2, 2)?

1	2	3
4	8	2
1	5	3

The path with minimum cost is highlighted in the following figure. The path is (0, 0) > (0, 1) > (1, 2) > (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).



1) Optimal Substructure
The path to reach (m, n) must be through one of the 3 cells: (m-1, n-1) or (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as minimum of the 3 cells plus cost[m][n].

minCost(m, n) = min (minCost(m-1, n-1), minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

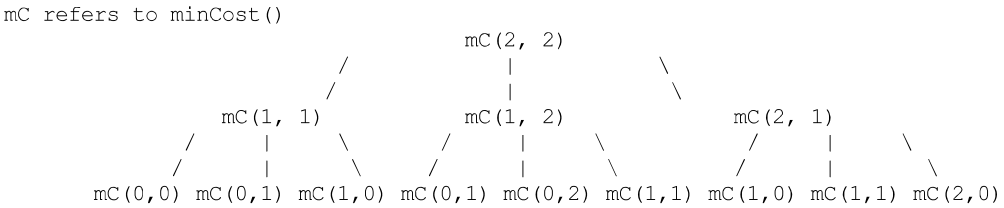
2) Overlapping Subproblems
Following is simple recursive implementation of the MCP (Minimum Cost Path) problem. The implementation simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of MCP(Minimum Cost Path) problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

/* Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]*/
int minCost(int cost[R][C], int m, int n)
{
   if (n < 0 || m < 0)
      return INT_MAX;
   else if (m == 0 && n == 0)
      return cost[m][n];
   else
      return cost[m][n] + min( minCost(cost, m-1, n-1),
                               minCost(cost, m-1, n),
                               minCost(cost, m, n-1) );
}

/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
   if (x < y)
      return (x < z)? x : z;
   else
      return (y < z)? y : z;
}

/* Driver program to test above functions */
int main()
{
   int cost[R][C] = { {1, 2, 3},
                      {4, 8, 2},
```

Aman Barnwal

```
                      {1, 5, 3} };
   printf(" %d ", minCost(cost, 2, 2));
   return 0;
}
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, there are many nodes which apear more than once. Time complexity of this naive recursive solution is exponential and it is terribly slow.



So the MCP problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array tc[][] in bottom up manner.

C++

```
/* Dynamic Programming implementation of MCP problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

int minCost(int cost[R][C], int m, int n)
{
     int i, j;

     // Instead of following line, we can use int tc[m+1][n+1] or
     // dynamically allocate memoery to save space. The following line is
     // used to keep te program simple and make it working on all compilers.
     int tc[R][C];

     tc[0][0] = cost[0][0];

     /* Initialize first column of total cost(tc) array */
     for (i = 1; i <= m; i++)
        tc[i][0] = tc[i-1][0] + cost[i][0];

     /* Initialize first row of tc array */
     for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];

     /* Construct rest of the tc array */
     for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j-1], tc[i-1][j], tc[i][j-1]) + cost[i][j];

     return tc[m][n];
}

/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
   if (x < y)
      return (x < z)? x : z;
   else
      return (y < z)? y : z;
}

/* Driver program to test above functions */
int main()
{
   int cost[R][C] = { {1, 2, 3},
                      {4, 8, 2},
                      {1, 5, 3} };
   printf(" %d ", minCost(cost, 2, 2));
   return 0;
}
```

Python

Aman Barnwal

```
# Dynamic Programming Python implementation of Min Cost Path
# problem
R = 3
C = 3

def minCost(cost, m, n):

    # Instead of following line, we can use int tc[m+1][n+1] or
    # dynamically allocate memoery to save space. The following
    # line is used to keep te program simple and make it working
    # on all compilers.
    tc = [[0 for x in range(C)] for x in range(R)]

    tc[0][0] = cost[0][0]

    # Initialize first column of total cost(tc) array
    for i in range(1, m+1):
        tc[i][0] = tc[i-1][0] + cost[i][0]

    # Initialize first row of tc array
    for j in range(1, n+1):
        tc[0][j] = tc[0][j-1] + cost[0][j]

    # Construct rest of the tc array
    for i in range(1, m+1):
        for j in range(1, n+1):
            tc[i][j] = min(tc[i-1][j-1], tc[i-1][j], tc[i][j-1]) + cost[i][j]

    return tc[m][n]

# Driver program to test above functions
cost = [[1, 2, 3],
        [4, 8, 2],
        [1, 5, 3]]
print(minCost(cost, 2, 2))

# This code is contributed by Bhavya Jain
```

```
8
```

Time Complexity of the DP implementation is O(mn) which is much better than Naive Recursive implementation.

Aman Barnwal

Dynamic Programming | Set 7 (Coin Change)

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesnt matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

1) Optimal Substructure
To count total number solutions, we can divide all set solutions in two sets.
1) Solutions that do not contain mth coin (or Sm).
2) Solutions that contain at least one Sm.
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

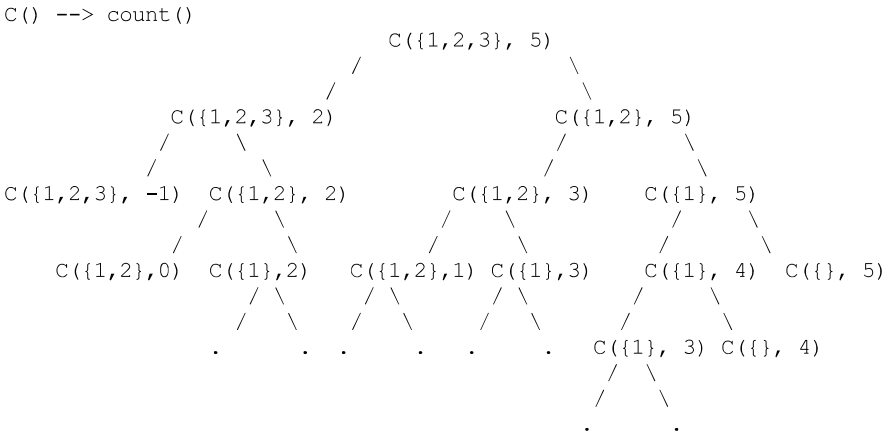
Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

2) Overlapping Subproblems
Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

```
#include<stdio.h>

// Returns the count of ways we can sum  S[0...m-1] coins to get sum n
int count( int S[], int m, int n )
{
    // If n is 0 then there is 1 solution (do not include any coin)
    if (n == 0)
        return 1;

    // If n is less than 0 then no solution exists
    if (n < 0)
        return 0;

    // If there are no coins and n is greater than 0, then no solution exist
    if (m <=0 && n >= 1)
        return 0;

    // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]
    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}

// Driver program to test above function
int main()
{
    int i, j;
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    printf("%d ", count(arr, m, 4));
    getchar();
    return 0;
}
```

Aman Barnwal

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {1, 2, 3} and n = 5.
The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.



Since same suproblems are called again, this problem has Overlapping Subprolems property. So the Coin Change problem has both properties

(see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array table[][] in bottom up manner.

Dynamic Programming Solution

C

```
#include<stdio.h>

int count( int S[], int m, int n )
{
    int i, j, x, y;

    // We need n+1 rows as the table is consturcted in bottom up manner using
    // the base case 0 value case (n = 0)
    int table[n+1][m];

    // Fill the enteries for 0 value case (n = 0)
    for (i=0; i<m; i++)
        table[0][i] = 1;

    // Fill rest of the table enteries in bottom up manner
    for (i = 1; i < n+1; i++)
    {
        for (j = 0; j < m; j++)
        {
            // Count of solutions including S[j]
            x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;

            // Count of solutions excluding S[j]
            y = (j >= 1)? table[i][j-1]: 0;

            // total count
            table[i][j] = x + y;
        }
    }
    return table[n][m-1];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    int n = 4;
    printf(" %d ", count(arr, m, n));
    return 0;
}
```

Python

```
# Dynamic Programming Python implementation of Coin Change problem
def count(S, m, n):
    # We need n+1 rows as the table is consturcted in bottom up
    # manner using the base case 0 value case (n = 0)
    table = [[0 for x in range(m)] for x in range(n+1)]

    # Fill the enteries for 0 value case (n = 0)
    for i in range(m):
        table[0][i] = 1

    # Fill rest of the table enteries in bottom up manner
    for i in range(1, n+1):
        for j in range(m):
            # Count of solutions including S[j]
            x = table[i - S[j]][j] if i-S[j] >= 0 else 0

            # Count of solutions excluding S[j]
            y = table[i][j-1] if j >= 1 else 0

            # total count
            table[i][j] = x + y

    return table[n][m-1]

# Driver program to test above function
arr = [1, 2, 3]
m = len(arr)
```

Aman Barnwal

```
n = 4
print(count(arr, m, n))

# This code is contributed by Bhavya Jain
```

```
4
```

Time Complexity: O(mn)

Following is a simplified version of method 2. The auxiliary space required here is O(n) only.

```
int count( int S[], int m, int n )
{
    // table[i] will be storing the number of solutions for
    // value i. We need n+1 rows as the table is consturcted
    // in bottom up manner using the base case (n = 0)
    int table[n+1];

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 1;

    // Pick all coins one by one and update the table[] values
    // after the index greater than or equal to the value of the
    // picked coin
    for(int i=0; i<m; i++)
        for(int j=S[i]; j<=n; j++)
            table[j] += table[j-S[i]];

    return table[n];
}
```

Thanks to Rohan Laishram for suggesting this space optimized version.

References:
http://www.algorithmist.com/index.php/Coin_Change

Aman Barnwal

Dynamic Programming | Set 8 (Matrix Chain Multiplication)

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

```
(ABC)D = (AB)(CD) = A(BCD) = ....
```

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 30 matrix, B is a 30 5 matrix, and C is a 5 60 matrix. Then,

```
(AB)C = (10305) + (10560) = 1500 + 3000 = 4500 operations
A(BC) = (30560) + (103060) = 9000 + 18000 = 27000 operations.
```

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

```
Input: p[] = {40, 20, 30, 10, 30}
Output: 26000
There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30}
Output: 30000
There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

Input: p[] = {10, 20, 30}
Output: 6000
There are only two matrices of dimensions 10x20 and 20x30. So there
is only one way to multiply the matrices, cost of which is 10*20*30
```

1) Optimal Substructure:
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 way to place first set of parenthesis: A(BCD), (AB)CD and (ABC)D. So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

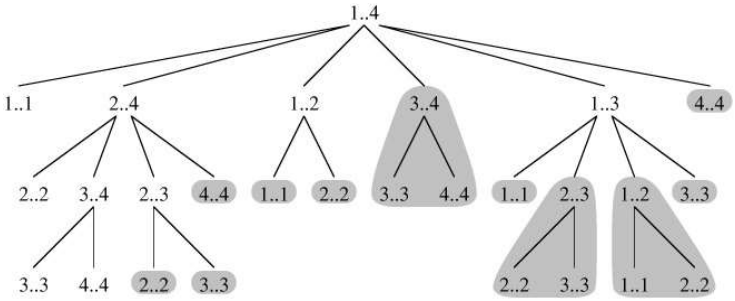
2) Overlapping Subproblems
Following is a recursive implementation that simply follows the above optimal substructure property.

```
/* A naive recursive implementation that simply follows the above optimal
 substructure property */
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between first and last matrix,
    // recursively calculate count of multiplcations for each parenthesis
    // placement and return the minimum count
    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
```

Aman Barnwal

```
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                          MatrixChainOrder(arr, 1, n-1));

    getchar();
    return 0;
}
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 4) is called two times. We can see that there are many subproblems being called more than once.



Since same suproblems are called again, this problem has Overlapping Subprolems property. So Matrix Chain Multiplication problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array m[][] in bottom up manner.

Dynamic Programming Solution
Following is C/C++ implementation for Matrix Chain Multiplication problem using Dynamic Programming.

C

```
// See the Cormen book for details of the following algorithm
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one extra row and one extra column are
       allocated in m[][].  0th row and 0th column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar multiplications needed to compute
       the matrix A[i]A[i+1]...A[j] = A[i..j] where dimention of A[i] is
       p[i-1] x p[i] */

    // cost is zero when multiplying one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<=n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
```

Aman Bernwal

```
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}

int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                       MatrixChainOrder(arr, size));

    getchar();
    return 0;
}
```

Python

```
# Dynamic Programming Python implementation of Matrix Chain Multiplication
# See the Cormen book for details of the following algorithm
import sys

# Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
def MatrixChainOrder(p, n):
    # For simplicity of the program, one extra row and one extra column are
    # allocated in m[][].  0th row and 0th column of m[][] are not used
    m = [[0 for x in range(n)] for x in range(n)]

    # m[i,j] = Minimum number of scalar multiplications needed to compute
    # the matrix A[i]A[i+1]...A[j] = A[i..j] where dimention of A[i] is
    # p[i-1] x p[i]

    # cost is zero when multiplying one matrix.
    for i in range(1, n):
        m[i][i] = 0

    # L is chain length.
    for L in range(2, n):
        for i in range(1, n-L+1):
            j = i+L-1
            m[i][j] = sys.maxint
            for k in range(i, j):

                # q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                if q < m[i][j]:
                    m[i][j] = q

    return m[1][n-1]

# Driver program to test above function
arr = [1, 2, 3 ,4]
size = len(arr)

print("Minimum number of multiplications is " + str(MatrixChainOrder(arr, size)))
# This Code is contributed by Bhavya Jain
```

```
Minimum number of multiplications is 18
```

Time Complexity: O(n^3)
Auxiliary Space: O(n^2)

References:
http://en.wikipedia.org/wiki/Matrix_chain_multiplication
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm

Aman Barnwal

Dynamic Programming | Set 9 (Binomial Coefficient)

Following are common definition of Binomial Coefficients.
1) A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.

2) A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

The Problem
Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

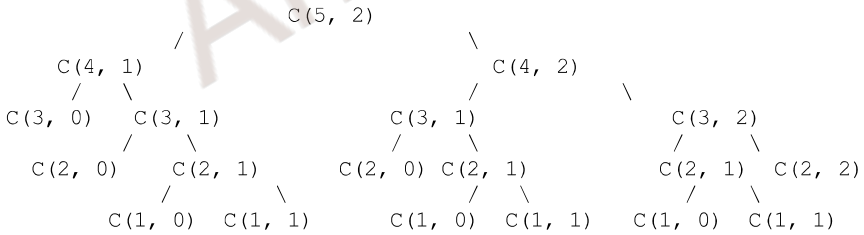
1) Optimal Substructure
The value of C(n, k) can recursively calculated using following standard formula for Binomial Cofficients.

```
   C(n, k) = C(n-1, k-1) + C(n-1, k)
   C(n, 0) = C(n, n) = 1
```

2) Overlapping Subproblems
Following is simple recursive implementation that simply follows the recursive structure mentioned above.

```
// A Naive Recursive Implementation
#include<stdio.h>

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
  // Base Cases
  if (k==0 || k==n)
    return 1;

  // Recur
  return  binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k);
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k));
    return 0;
}
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for n = 5 an k = 2. The function C(3, 1) is called two times. For large values of n, there will be many common subproblems.



Since same suproblems are called again, this problem has Overlapping Subprolems property. So the Binomial Coefficient problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array C[][] in bottom up manner. Following is Dynamic Programming based implementation.

C

```
// A Dynamic Programming based solution that uses table C[][] to
// calculate the Binomial Coefficient
#include<stdio.h>

// Prototype of a utility function that returns minimum of two integers
int min(int a, int b);

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int C[n+1][k+1];
    int i, j;
```

Aman Barnwal

```
    // Caculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (j = 0; j <= min(i, k); j++)
        {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previosly stored values
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }

    return C[n][k];
}

// A utility function to return minimum of two integers
int min(int a, int b)
{
    return (a<b)? a: b;
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );
    return 0;
}
```

Python

```
# A Dynamic Programming based Python Program that uses table C[][]
# to calculate the Binomial Coefficient

# Returns value of Binomial Coefficient C(n, k)
def binomialCoef(n, k):
    C = [[0 for x in range(k+1)] for x in range(n+1)]

    # Calculate value of Binomial Coefficient in bottom up manner
    for i in range(n+1):
        for j in range(min(i, k)+1):
            # Base Cases
            if j == 0 or j == i:
                C[i][j] = 1

            # Calculate value using previosly stored values
            else:
                C[i][j] = C[i-1][j-1] + C[i-1][j]

    return C[n][k]

# Driver program to test above function
n = 5
k = 2
print("Value of C[" + str(n) + "][" + str(k) + "] is "
      + str(binomialCoef(n,k)))

# This code is contributed by Bhavya Jain



Value of C[5][2] is 10
```

Time Complexity: O(n*k)
Auxiliary Space: O(n*k)

Following is a space optimized version of the above code. The following code only uses O(k). Thanks to AK for suggesting this method.

```
// A space optimized Dynamic Programming Solution
int binomialCoeff(int n, int k)
{
    int* C = (int*)calloc(k+1, sizeof(int));
    int i, j, res;

    C[0] = 1;
```

Aman Barnwal

```
    for(i = 1; i <= n; i++)
    {
        for(j = min(i, k); j > 0; j--)
            C[j] = C[j] + C[j-1];
    }

    res = C[k];  // Store the result before freeing memory

    free(C);  // free dynamically allocated memory to avoid memory leak

    return res;
}
```

Time Complexity: O(n*k)
Auxiliary Space: O(k)

References:
http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2015%20-%20Dynamic%20Programming%20Binomial%20Coefficients.htm

Aman Barnwal

Dynamic Programming | Set 10 (0-1 Knapsack Problem)

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or dont pick it (0-1 property).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure:
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.

C/C++

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   // Base Case
   if (n == 0 || W == 0)
       return 0;

   // If weight of the nth item is more than Knapsack capacity W, then
   // this item cannot be included in the optimal solution
   if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

   // Return the maximum of two cases:
   // (1) nth item included
   // (2) not included
   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1)
                  );
}

// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Python

```
#A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):
    # Base Case
    if n == 0 or W == 0 :
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
```

Aman Barnwal

```
        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                   knapSack(W , wt , val , n-1))
# end of function knapSack

# To test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W , wt , val , n)
# This code is contributed by Nikhil Kumar Singh



220
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

```
In the following recursion tree, K() refers to knapSack().  The two
parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

                        K(3, 2)         ---------> K(n, W)
                      /            \
                    /                \
                /                      \
            K(2,2)                       K(2,1)
          /       \                      /    \
        /           \                  /        \
       /             \               /           \
     K(1,2)        K(1,1)          K(1,1)       K(1,0)
     /  \          /   \           /   \
   /      \      /       \       /       \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)   K(0,0)
Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.
```

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

C++

```
// A Dynamic Programming based solution for 0-1 Knapsack problem
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];

   // Build table K[][] in bottom up manner
   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }

   return K[n][W];
}

int main()
```

Amar Barnwal

```
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Pyhton

```
# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]

# Driver program to test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))

# This code is contributed by Bhavya Jain
```

```
220
```

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

References:
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf
http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf

Aman Barnwal

Dynamic Programming | Set 11 (Egg Dropping Puzzle)

The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

..An egg that survives a fall can be used again.
..A broken egg must be discarded.
..The effect of a fall is the same for all eggs.
..If an egg breaks when dropped, then it would break if dropped from a higher floor.
..If an egg survives a fall then it would survive a shorter fall.
..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Source: Wiki for Dynamic Programming

In this post, we will discuss solution to a general problem with n eggs and k floors. The solution is to try dropping an egg from every floor (from 1 to k) and recursively calculate the minimum number of droppings needed in worst case. The floor which gives the minimum value in worst case is going to be part of the solution.
In the following solutions, we return the minimum number of trails in worst case; these solutions can be easily modified to print floor numbers of every trials also.

1) Optimal Substructure:
When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesnt break.

1) If the egg breaks after dropping from xth floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to x-1 floors and n-1 eggs
2) If the egg doesnt break after dropping from the xth floor, then we only need to check for floors higher than x; so the problem reduces to k-x floors and n eggs.

Since we need to minimize the number of trials in *worst* case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

```
  k ==> Number of floors
  n ==> Number of Eggs
  eggDrop(n, k) ==> Minimum number of trails needed to find the critical
                    floor in worst case.
  eggDrop(n, k) = 1 + min{max(eggDrop(n - 1, x - 1), eggDrop(n, k - x)):
                  x in {1, 2, ..., k}}
```

2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.

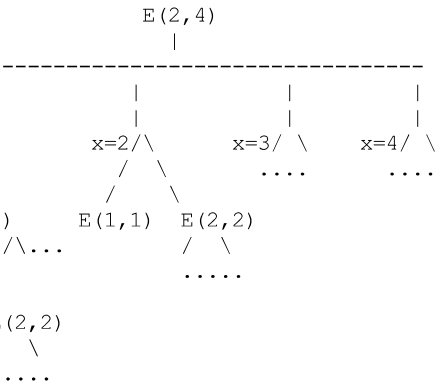
```
# include <stdio.h>
# include <limits.h>

// A utility function to get maximum of two integers
int max(int a, int b) { return (a > b)? a: b; }

/* Function to get minimum number of trails needed in worst
  case with n eggs and k floors */
int eggDrop(int n, int k)
{
    // If there are no floors, then no trials needed. OR if there is
    // one floor, one trial needed.
    if (k == 1 || k == 0)
        return k;

    // We need k trials for one egg and k floors
    if (n == 1)
        return k;

    int min = INT_MAX, x, res;

    // Consider all droppings from 1st floor to kth floor and
    // return the minimum of these values plus 1.
    for (x = 1; x <= k; x++)
```

Aman Barnwal

```
    {
        res = max(eggDrop(n-1, x-1), eggDrop(n, k-x));
        if (res < min)
            min = res;
    }

    return min + 1;
}

/* Driver program to test to pront printDups*/
int main()
{
    int n = 2, k = 10;
    printf ("\nMinimum number of trials in worst case with %d eggs and "
             "%d floors is %d \n", n, k, eggDrop(n, k));
    return 0;
}
```

```
Output:
Minimum number of trials in worst case with 2 eggs and 10 floors is 4
```

It should be noted that the above function computes the same subproblems again and again. See the following partial recursion tree, E(2, 2) is being evaluated twice. There will many repeated subproblems when you draw the complete recursion tree even for small values of n and k.




```
Partial recursion tree for 2 eggs and 4 floors.
```

Since same suproblems are called again, this problem has Overlapping Subprolems property. So Egg Dropping Puzzle has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array eggFloor[][] in bottom up manner.

Dynamic Programming Solution
Following are C++ and Python implementations for Egg Dropping problem using Dynamic Programming.

C++

```
# A Dynamic Programming based C++ Program for the Egg Dropping Puzzle
# include <stdio.h>
# include <limits.h>

// A utility function to get maximum of two integers
int max(int a, int b) { return (a > b)? a: b; }

/* Function to get minimum number of trails needed in worst
  case with n eggs and k floors */
int eggDrop(int n, int k)
{
    /* A 2D table where entery eggFloor[i][j] will represent minimum
       number of trials needed for i eggs and j floors. */
    int eggFloor[n+1][k+1];
    int res;
    int i, j, x;

    // We need one trial for one floor and0 trials for 0 floors
    for (i = 1; i <= n; i++)
    {
        eggFloor[i][1] = 1;
        eggFloor[i][0] = 0;
    }

    // We always need j trials for one egg and j floors.
    for (j = 1; j <= k; j++)
        eggFloor[1][j] = j;
```

Aman Barnwal

```
    // Fill rest of the entries in table using optimal substructure
    // property
    for (i = 2; i <= n; i++)
    {
        for (j = 2; j <= k; j++)
        {
            eggFloor[i][j] = INT_MAX;
            for (x = 1; x <= j; x++)
            {
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);
                if (res < eggFloor[i][j])
                    eggFloor[i][j] = res;
            }
        }
    }

    // eggFloor[n][k] holds the result
    return eggFloor[n][k];
}

/* Driver program to test to pront printDups*/
int main()
{
    int n = 2, k = 36;
    printf ("\nMinimum number of trials in worst case with %d eggs and "
             "%d floors is %d \n", n, k, eggDrop(n, k));
    return 0;
}
```

Python

```
# A Dynamic Programming based Python Program for the Egg Dropping Puzzle
INT_MAX = 32767

# Function to get minimum number of trails needed in worst
# case with n eggs and k floors
def eggDrop(n, k):
    # A 2D table where entery eggFloor[i][j] will represent minimum
    # number of trials needed for i eggs and j floors.
    eggFloor = [[0 for x in range(k+1)] for x in range(n+1)]

    # We need one trial for one floor and0 trials for 0 floors
    for i in range(1, n+1):
        eggFloor[i][1] = 1
        eggFloor[i][0] = 0

    # We always need j trials for one egg and j floors.
    for j in range(1, k+1):
        eggFloor[1][j] = j

    # Fill rest of the entries in table using optimal substructure
    # property
    for i in range(2, n+1):
        for j in range(2, k+1):
            eggFloor[i][j] = INT_MAX
            for x in range(1, j+1):
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x])
                if res < eggFloor[i][j]:
                    eggFloor[i][j] = res

    # eggFloor[n][k] holds the result
    return eggFloor[n][k]

# Driver program to test to pront printDups
n = 2
k = 36
print("Minimum number of trials in worst case with" + str(n) + "eggs and " \
       + str(k) + " floors is " + str(eggDrop(n, k)))

# This code is contributed by Bhavya Jain
```

```
Output:
Minimum number of trials in worst case with 2 eggs and 36 floors is 8
```

Time Complexity: O(nk^2)
Auxiliary Space: O(nk)

As an exercise, you may try modifying the above DP solution to print all intermediate floors (The floors used for minimum trail solution).

Aman Barnwal

References:

http://archive.ite.journal.informs.org/Vol4No1/Sniedovich/index.php

Aman Barnwal

Dynamic Programming | Set 12 (Longest Palindromic Subsequence)

Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is BBABCBCAB, then the output should be 7 as BABCBAB is the longest palindromic subseuqnce in it. BBBBB and BBCBB are also palindromic subsequences of the given sequence, but not the longest ones.

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:
Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].

If last and first characters of X are same, then L(0, n-1) = L(1, n-2) + 2.
Else L(0, n-1) = MAX (L(1, n-1), L(0, n-2)).

Following is a general recursive solution with all cases handled.

```
// Everay single character is a palindrom of length 1
L(i, i) = 1 for all indexes i in given sequence

// IF first and last characters are not same
If (X[i] != X[j])  L(i, j) =  max{L(i + 1, j),L(i, j - 1)}

// If there are only 2 characters and both are same
Else if (j == i + 1) L(i, j) = 2

// If there are more than two characters, and first and last
// characters are same
Else L(i, j) =  L(i + 1, j - 1) + 2
```

2) Overlapping Subproblems
Following is simple recursive implementation of the LPS problem. The implementation simply follows the recursive structure mentioned above.

```
#include<stdio.h>
#include<string.h>

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *seq, int i, int j)
{
   // Base Case 1: If there is only 1 character
   if (i == j)
     return 1;

   // Base Case 2: If there are only 2 characters and both are same
   if (seq[i] == seq[j] && i + 1 == j)
     return 2;

   // If the first and last characters match
   if (seq[i] == seq[j])
      return lps (seq, i+1, j-1) + 2;

   // If the first and last characters do not match
   return max( lps(seq, i, j-1), lps(seq, i+1, j) );
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKSFORGEEKS";
    int n = strlen(seq);
    printf ("The lnegth of the LPS is %d", lps(seq, 0, n-1));
    getchar();
    return 0;
}
```

Output:

```
The lnegth of the LPS is 5
```

Considering the above implementation, following is a partial recursion tree for a sequence of length 6 with all different characters.

```
                L(0, 5)
              /        \
```

Aman Barnwal

```
            /          \
        L(1,5)          L(0,4)
       /    \            /    \
      /      \          /      \
  L(2,5)    L(1,4)  L(1,4)  L(0,3)
```

In the above partial recursion tree, L(1, 4) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. Since same suproblems are called again, this problem has Overlapping Subprolems property. So LPS problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array L[][] in bottom up manner.

Dynamic Programming Solution

C++

```
# A Dynamic Programming based Python program for LPS problem
# Returns the length of the longest palindromic subsequence in seq
#include<stdio.h>
#include<string.h>

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *str)
{
   int n = strlen(str);
   int i, j, cl;
   int L[n][n];  // Create a table to store results of subproblems


   // Strings of length 1 are palindrome of lentgh 1
   for (i = 0; i < n; i++)
      L[i][i] = 1;

    // Build the table. Note that the lower diagonal values of table are
    // useless and not filled in the process. The values are filled in a
    // manner similar to Matrix Chain Multiplication DP solution (See
    // http://www.geeksforgeeks.org/archives/15553). cl is length of
    // substring
    for (cl=2; cl<=n; cl++)
    {
        for (i=0; i<n-cl+1; i++)
        {
            j = i+cl-1;
            if (str[i] == str[j] && cl == 2)
               L[i][j] = 2;
            else if (str[i] == str[j])
               L[i][j] = L[i+1][j-1] + 2;
            else
               L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }

    return L[0][n-1];
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKS FOR GEEKS";
    int n = strlen(seq);
    printf ("The lnegth of the LPS is %d", lps(seq));
    getchar();
    return 0;
}
```

Python

```
# A Dynamic Programming based Python program for LPS problem
# Returns the length of the longest palindromic subsequence in seq
def lps(str):
 n = len(str)

 # Create a table to store results of subproblems
 L = [[0 for x in range(n)] for x in range(n)]

 # Strings of length 1 are palindrome of length 1
```

Aman Bamwal

```
 for i in range(n):
  L[i][i] = 1

 # Build the table. Note that the lower diagonal values of table are
    # useless and not filled in the process. The values are filled in a
    # manner similar to Matrix Chain Multiplication DP solution (See
    # http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/
    # cl is length of substring
 for cl in range(2, n+1):
  for i in range(n-cl+1):
   j = i+cl-1
   if str[i] == str[j] and cl == 2:
    L[i][j] = 2
   elif str[i] == str[j]:
    L[i][j] = L[i+1][j-1] + 2
   else:
    L[i][j] = max(L[i][j-1], L[i+1][j]);

 return L[0][n-1]

# Driver program to test above functions
seq = "GEEKS FOR GEEKS"
n = len(seq)
print("The length of the LPS is " + str(lps(seq)))

# This code is contributed by Bhavya Jain
```

```
The lnegth of the LPS is 7
```

Time Complexity of the above implementation is O(n^2) which is much better than the worst case time complexity of Naive Recursive implementation.

This problem is close to the Longest Common Subsequence (LCS) problem. In fact, we can use LCS as a subroutine to solve this problem. Following is the two step solution that uses LCS.
1) Reverse the given sequence and store the reverse in another array say rev[0..n-1]
2) LCS of the given sequence and rev[] will be the longest palindromic sequence.
This solution is also a O(n^2) solution.

References:
http://users.eecs.northwestern.edu/~dda902/336/hw6-sol.pdf

Aman Barnwal

Dynamic Programming | Set 13 (Cutting a Rod)

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 1   5   8   9  10  17  17  20
```

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 3   5   8   9  10  17  17  20
```

The naive solution for this problem is to generate all configurations of different pieces and find the highest priced configuration. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:
We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.

Let cutRoad(n) be the required (best possible price) value for a rod of lenght n. cutRod(n) can be written as following.

cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

2) Overlapping Subproblems
Following is simple recursive implementation of the Rod Cutting problem. The implementation simply follows the recursive structure mentioned above.

```
// A Naive recursive solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b;}

/* Returns the best obtainable price for a rod of length n and
   price[] as prices of different pieces */
int cutRod(int price[], int n)
{
   if (n <= 0)
     return 0;
   int max_val = INT_MIN;

   // Recursively cut the rod in different pieces and compare different
   // configurations
   for (int i = 0; i<n; i++)
         max_val = max(max_val, price[i] + cutRod(price, n-i-1));

   return max_val;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}
```

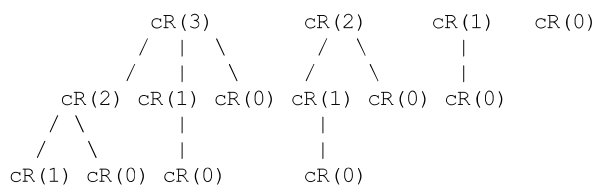
Output:

```
Maximum Obtainable Value is 22
```

Considering the above implementation, following is recursion tree for a Rod of length 4.

```
cR() ---> cutRod()

                          cR(4)
                 /       /     \     \
               /       /         \      \
```

Aman Barnwal



In the above partial recursion tree, cR(2) is being solved twice. We can see that there are many subproblems which are solved again and again. Since same suproblems are called again, this problem has Overlapping Subprolems property. So the Rod Cutting problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array val[] in bottom up manner.

C++

```
// A Dynamic Programming solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b;}

/* Returns the best obtainable price for a rod of length n and
   price[] as prices of different pieces */
int cutRod(int price[], int n)
{
   int val[n+1];
   val[0] = 0;
   int i, j;

   // Build the table val[] in bottom up manner and return the last entry
   // from the table
   for (i = 1; i<=n; i++)
   {
       int max_val = INT_MIN;
       for (j = 0; j < i; j++)
         max_val = max(max_val, price[j] + val[i-j-1]);
       val[i] = max_val;
   }

   return val[n];
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}
```

Python

```
# A Dynamic Programming solution for Rod cutting problem
INT_MIN = -32767

# Returns the best obtainable price for a rod of length n and
# price[] as prices of different pieces
def cutRod(price, n):
    val = [0 for x in range(n+1)]
    val[0] = 0

    # Build the table val[] in bottom up manner and return
    # the last entry from the table
    for i in range(1, n+1):
        max_val = INT_MIN
        for j in range(i):
             max_val = max(max_val, price[j] + val[i-j-1])
        val[i] = max_val

    return val[n]

# Driver program to test above functions
arr = [1, 5, 8, 9, 10, 17, 17, 20]
size = len(arr)
print("Maximum Obtainable Value is " + str(cutRod(arr, size)))
```

Aman Barnwal

```
# This code is contributed by Bhavya Jain


Maximum Obtainable Value is 22
```

Time Complexity of the above implementation is O(n^2) which is much better than the worst case time complexity of Naive Recursive implementation.

Aman Barnwal

Dynamic Programming | Set 14 (Maximum Sum Increasing Subsequence)

Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the intgers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 (1 + 2 + 3 + 100), if the input array is {3, 4, 5, 10}, then output should be 22 (3 + 4 + 5 + 10) and if the input array is {10, 5, 4, 3}, then output should be 10

Solution
This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. We need a slight change in the Dynamic Programming solution of LIS problem. All we need to change is to use sum as a criteria instead of length of increasing subsequence.

Following are C/C++ and Python implementations for Dynamic Programming solution of the problem.

C/C++

```
/* Dynamic Programming implementation of Maximum Sum Increasing
Subsequence (MSIS) problem */
#include<stdio.h>

/* maxSumIS() returns the maximum sum of increasing subsequence
    in arr[] of size n */
int maxSumIS( int arr[], int n )
{
    int i, j, max = 0;
    int msis[n];

    /* Initialize msis values for all indexes */
    for ( i = 0; i < n; i++ )
        msis[i] = arr[i];

    /* Compute maximum sum values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && msis[i] < msis[j] + arr[i])
                msis[i] = msis[j] + arr[i];

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

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 101, 2, 3, 100, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Sum of maximum sum increasing subsequence is %d\n",
           maxSumIS( arr, n ) );
    return 0;
}
```

Python

```
# Dynamic Programming bsed Python implementation of Maximum Sum Increasing
# Subsequence (MSIS) problem

# maxSumIS() returns the maximum sum of increasing subsequence in arr[] of
# size n
def maxSumIS(arr, n):
    max = 0
    msis = [0 for x in range(n)]

    # Initialize msis values for all indexes
    for i in range(n):
        msis[i] = arr[i]

    # Compute maximum sum values in bottom up manner
    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j] and msis[i] < msis[j] + arr[i]:
                msis[i] = msis[j] + arr[i]

    # Pick maximum of all msis values
    for i in range(n):
        if max < msis[i]:
```

Aman Barnwal

```
            max = msis[i]

    return max

# Driver program to test above function
arr = [1, 101, 2, 3, 100, 4, 5]
n = len(arr)
print("Sum of maximum sum increasing subsequence is " +
       str(maxSumIS(arr, n)))

# This code is contributed by Bhavya Jain
```

Output:

```
Sum of maximum sum increasing subsequence is 106
```

Time Complexity: O(n^2)

Source: Maximum Sum Increasing Subsequence Problem

Aman Barnwal

Dynamic Programming | Set 15 (Longest Bitonic Subsequence)

Given an array arr[0 n-1] containing n positive integers, a subsequence of arr[] is called Bitonic if it is first increasing, then decreasing. Write a function that takes an array as argument and returns the length of the longest bitonic subsequence.
A sequence, sorted in increasing order is considered Bitonic with the decreasing part as empty. Similarly, decreasing order sequence is considered Bitonic with the increasing part as empty.

Examples:

```
Input arr[] = {1, 11, 2, 10, 4, 5, 2, 1};
Output: 6 (A Longest Bitonic Subsequence of length 6 is 1, 2, 10, 4, 2, 1)

Input arr[] = {12, 11, 40, 5, 3, 1}
Output: 5 (A Longest Bitonic Subsequence of length 5 is 12, 11, 5, 3, 1)

Input arr[] = {80, 60, 30, 40, 20, 10}
Output: 5 (A Longest Bitonic Subsequence of length 5 is 80, 60, 30, 20, 10)
```

Source: Microsoft Interview Question

Solution
This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. Let the input array be arr[] of length n. We need to construct two arrays lis[] and lds[] using Dynamic Programming solution of LIS problem. lis[i] stores the length of the Longest Increasing subsequence ending with arr[i]. lds[i] stores the length of the longest Decreasing subsequence starting from arr[i]. Finally, we need to return the max value of lis[i] + lds[i] 1 where i is from 0 to n-1.

Following is C++ implementation of the above Dynamic Programming solution.

C++

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

/* lbs() returns the length of the Longest Bitonic Subsequence in
    arr[] of size n. The function mainly creates two temporary arrays
    lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

    lis[i] ==> Longest Increasing subsequence ending with arr[i]
    lds[i] ==> Longest decreasing subsequence starting with arr[i]
*/
int lbs( int arr[], int n )
{
   int i, j;

   /* Allocate memory for LIS[] and initialize LIS values as 1 for
      all indexes */
   int *lis = new int[n];
   for (i = 0; i < n; i++)
      lis[i] = 1;

   /* Compute LIS values from left to right */
   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;

   /* Allocate memory for lds and initialize LDS values for
      all indexes */
   int *lds = new int [n];
   for (i = 0; i < n; i++)
      lds[i] = 1;

   /* Compute LDS values from right to left */
   for (i = n-2; i >= 0; i--)
      for (j = n-1; j > i; j--)
         if (arr[i] > arr[j] && lds[i] < lds[j] + 1)
            lds[i] = lds[j] + 1;


   /* Return the maximum value of lis[i] + lds[i] - 1*/
   int max = lis[0] + lds[0] - 1;
   for (i = 1; i < n; i++)
     if (lis[i] + lds[i] - 1 > max)
         max = lis[i] + lds[i] - 1;
   return max;
```

Aman Barnwal

```
}

/* Driver program to test above function */
int main()
{
  int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
               13, 3, 11, 7, 15};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Length of LBS is %d\n", lbs( arr, n ) );
  return 0;
}
```

Java

```
/* Dynamic Programming implementation in Java for longest bitonic
   subsequence problem */
import java.util.*;
import java.lang.*;
import java.io.*;

class LBS
{
    /* lbs() returns the length of the Longest Bitonic Subsequence in
    arr[] of size n. The function mainly creates two temporary arrays
    lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

    lis[i] ==> Longest Increasing subsequence ending with arr[i]
    lds[i] ==> Longest decreasing subsequence starting with arr[i]
    */
    static int lbs( int arr[], int n )
    {
        int i, j;

        /* Allocate memory for LIS[] and initialize LIS values as 1 for
         all indexes */
        int[] lis = new int[n];
        for (i = 0; i < n; i++)
            lis[i] = 1;

        /* Compute LIS values from left to right */
        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;

        /* Allocate memory for lds and initialize LDS values for
         all indexes */
        int[] lds = new int [n];
        for (i = 0; i < n; i++)
            lds[i] = 1;

        /* Compute LDS values from right to left */
        for (i = n-2; i >= 0; i--)
            for (j = n-1; j > i; j--)
                if (arr[i] > arr[j] && lds[i] < lds[j] + 1)
                    lds[i] = lds[j] + 1;


        /* Return the maximum value of lis[i] + lds[i] - 1*/
        int max = lis[0] + lds[0] - 1;
        for (i = 1; i < n; i++)
            if (lis[i] + lds[i] - 1 > max)
                max = lis[i] + lds[i] - 1;

        return max;
    }

    public static void main (String[] args)
    {
        int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
                     13, 3, 11, 7, 15};
        int n = arr.length;
        System.out.println("Length of LBS is "+ lbs( arr, n ));
    }
}
```

```
Length of LBS is 7
```

Aman Barnwal

Time Complexity: O(n^2)
Auxiliary Space: O(n)

Aman Barnwal

Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

```
       graph[][] = { {0,   5,  INF, 10},
                     {INF,  0,  3,  INF},
                     {INF, INF, 0,   1},
                     {INF, INF, INF, 0} }
which represents the following graph
             10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
             3
Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.
```

Output:

```
Shortest distance matrix
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

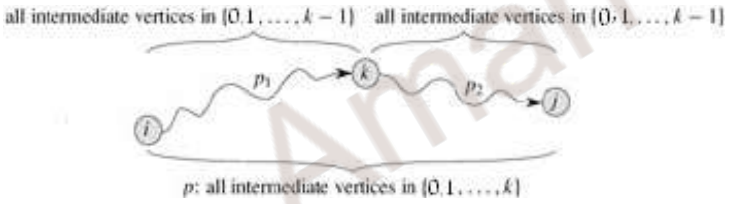
Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

C/C++

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
  for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshell (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
```

Aman Bansal

```
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
      ---> Before start of a iteration, we have shortest distances between all
      pairs of vertices such that the shortest distances consider only the
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added to the set of
      intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances"
            " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
            10
       (0)-------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)-------->(2)
            3           */
    int graph[V][V] = { {0,   5,  INF, 10},
                        {INF, 0,   3, INF},
                        {INF, INF, 0,   1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshell(graph);
    return 0;
}
```

Java

```
// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;
```

Aman Barnwal

```
class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
           Or we can say the initial values of shortest distances
           are based on shortest paths considering no intermediate
           vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate
           vertices.
          ---> Before start of a iteration, we have shortest
               distances between all pairs of vertices such that
               the shortest distances consider only the vertices in
               set {0, 1, 2, .. k-1} as intermediate vertices.
          ----> After the end of a iteration, vertex no. k is added
                to the set of intermediate vertices and the set
                becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++)
        {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++)
            {
                // Pick all vertices as destination for the
                // above picked source
                for (j = 0; j < V; j++)
                {
                    // If vertex k is on the shortest path from
                    // i to j, then update the value of dist[i][j]
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }

        // Print the shortest distance matrix
        printSolution(dist);
    }

    void printSolution(int dist[][])
    {
        System.out.println("Following matrix shows the shortest "+
                           "distances between every pair of vertices");
        for (int i=0; i<V; ++i)
        {
            for (int j=0; j<V; ++j)
            {
                if (dist[i][j]==INF)
                    System.out.print("INF ");
                else
                    System.out.print(dist[i][j]+"   ");
            }
            System.out.println();
        }
    }

    // Driver program to test above function
    public static void main (String[] args)
    {
        /* Let us create the following weighted graph
           10
        (0)------->(3)
        |         /|\
        5 |          |
        |          | 1
        \|/         |
        (1)------->(2)
           3           */
        int graph[][] = { {0,   5,  INF, 10},
                          {INF, 0,   3, INF},
                          {INF, INF, 0,   1},
                          {INF, INF, INF, 0}
```

Aman Barnwal

```
                        };
        AllPairShortestPath a = new AllPairShortestPath();

        // Print the solution
        a.floydWarshall(graph);
    }
}

// Contributed by Aakash Hasija
```

Output:

```
Following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Time Complexity: O(V^3)

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.
Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmatic overflow.

```
#include<limits.h>

#define INF INT_MAX
.........................
if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j])
    dist[i][j] = dist[i][k] + dist[k][j];
.........................
```

Aman Barnwal

Dynamic Programming | Set 17 (Palindrome Partitioning)

Given a string, a partitioning of the string is a *palindrome partitioning* if every substring of the partition is a palindrome. For example, aba|b|bbabb|a|b|aba is a palindrome partitioning of ababbbabbababa. Determine the fewest cuts needed for palindrome partitioning of a given string. For example, minimum 3 cuts are needed for ababbbabbababa. The three cuts are a|babbbab|b|ababa. If a string is palindrome, then minimum 0 cuts are needed. If a string of length n containing all different characters, then minimum n-1 cuts are needed.

Solution
This problem is a variation of Matrix Chain Multiplication problem. If the string is palindrome, then we simply return 0. Else, like the Matrix Chain Multiplication problem, we try making cuts at all possible places, recursively calculate the cost for each cut and return the minimum value.

Let the given string be str and minPalPartion() be the function that returns the fewest cuts needed for palindrome partitioning. following is the optimal substructure property.

```
// i is the starting index and j is the ending index. i must be passed as 0 and j as n-1
minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.
minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

// If none of the above conditions is true, then minPalPartion(str, i, j) can be
// calculated recursively using the following formula.
minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +
                                 minPalPartion(str, k+1, j) }
                           where k varies from i to j-1
```

Following is Dynamic Programming solution. It stores the solutions to subproblems in two arrays P[][] and C[][], and reuses the calculated values.

```
// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>

// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartion(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
       C[i][j] = Minimum number of cuts needed for palindrome partitioning
                 of substring str[i..j]
       P[i][j] = true if substring str[i..j] is palindrome, else false
       Note that C[i][j] is 0 if P[i][j] is true */
    int C[n][n];
    bool P[n][n];

    int i, j, k, L; // different looping variables

    // Every substring of length 1 is a palindrome
    for (i=0; i<n; i++)
    {
        P[i][i] = true;
        C[i][i] = 0;
    }

    /* L is substring length. Build the solution in bottom up manner by
       considering all substrings of length starting from 2 to n.
       The loop structure is same as Matrx Chain Multiplication problem (
       See http://www.geeksforgeeks.org/archives/15553 )*/
    for (L=2; L<=n; L++)
    {
        // For substring of length L, set different possible starting indexes
        for (i=0; i<n-L+1; i++)
        {
            j = i+L-1; // Set ending index

            // If L is 2, then we just need to compare two characters. Else
            // need to check two corner characters and value of P[i+1][j-1]
            if (L == 2)
                P[i][j] = (str[i] == str[j]);
            else
                P[i][j] = (str[i] == str[j]) && P[i+1][j-1];

            // IF str[i..j] is palindrome, then C[i][j] is 0
            if (P[i][j] == true)
```

Aman Barnwal

```
                C[i][j] = 0;
            else
            {
                // Make a cut at every possible localtion starting from i to j,
                // and get the minimum cost cut.
                C[i][j] = INT_MAX;
                for (k=i; k<=j-1; k++)
                    C[i][j] = min (C[i][j], C[i][k] + C[k+1][j]+1);
            }
        }
    }

    // Return the min cut value for complete string. i.e., str[0..n-1]
    return C[0][n-1];
}

// Driver program to test above function
int main()
{
   char str[] = "ababbbabbababa";
   printf("Min cuts needed for Palindrome Partitioning is %d",
           minPalPartion(str));
   return 0;
}
```

Output:

```
Min cuts needed for Palindrome Partitioning is 3
```

Time Complexity: $O(n^3)$

An optimization to above approach

In above approach, we can calculating minimum cut while finding all palindromic substring. If we finding all palindromic substring 1st and then we calculate minimum cut, time complexity will reduce to $O(n^2)$.
Thanks for **Vivek** for suggesting this optimization.

```
// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>

// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartion(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
       C[i] = Minimum number of cuts needed for palindrome partitioning
                  of substring str[0..i]
       P[i][j] = true if substring str[i..j] is palindrome, else false
       Note that C[i] is 0 if P[0][i] is true */
    int C[n];
    bool P[n][n];

    int i, j, k, L; // different looping variables

    // Every substring of length 1 is a palindrome
    for (i=0; i<n; i++)
    {
        P[i][i] = true;
    }

    /* L is substring length. Build the solution in bottom up manner by
       considering all substrings of length starting from 2 to n. */
    for (L=2; L<=n; L++)
    {
        // For substring of length L, set different possible starting indexes
        for (i=0; i<n-L+1; i++)
        {
            j = i+L-1; // Set ending index

            // If L is 2, then we just need to compare two characters. Else
            // need to check two corner characters and value of P[i+1][j-1]
            if (L == 2)
```

Aman Barnwal

```
                P[i][j] = (str[i] == str[j]);
            else
                P[i][j] = (str[i] == str[j]) && P[i+1][j-1];
        }
 }

 for (i=0; i<n; i++)
 {
  if (P[0][i] == true)
   C[i] = 0;
  else
  {
   C[i] = INT_MAX;
   for(j=0;j<i;j++)
   {
    if(P[j+1][i] == true && 1+C[j]<C[i])
     C[i]=1+C[j];
   }
  }
 }

    // Return the min cut value for complete string. i.e., str[0..n-1]
    return C[n-1];
}

// Driver program to test above function
int main()
{
   char str[] = "ababbbabbababa";
   printf("Min cuts needed for Palindrome Partitioning is %d",
           minPalPartion(str));
   return 0;
}
```

Output:

```
Min cuts needed for Palindrome Partitioning is 3
```

Time Complexity: $O(n^2)$

Aman Barnwal

Dynamic Programming | Set 18 (Partition problem)

Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same.

Examples

```
arr[] = {1, 5, 11, 5}
Output: true
The array can be partitioned as {1, 5, 5} and {11}

arr[] = {1, 5, 3}
Output: false
The array cannot be partitioned into equal sum sets.
```

Following are the two main steps to solve this problem:
1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.

The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Recursive Solution
Following is the recursive property of the second step mentioned above.

```
Let isSubsetSum(arr, n, sum/2) be the function that returns true if
there is a subset of arr[0..n-1] with sum equal to sum/2

The isSubsetSum problem can be divided into two subproblems
 a) isSubsetSum() without considering last element
    (reducing n to n-1)
 b) isSubsetSum considering the last element
    (reducing sum/2 by arr[n-1] and n to n-1)
If any of the above the above subproblems return true, then return true.
isSubsetSum (arr, n, sum/2) = isSubsetSum (arr, n-1, sum/2) ||
                              isSubsetSum (arr, n-1, sum/2 - arr[n-1])
```

C/C++

```
// A  recursive C program for partition problem
#include <stdio.h>

// A utility function that returns true if there is
// a subset of arr[] with sun equal to given sum
bool isSubsetSum (int arr[], int n, int sum)
{
   // Base Cases
   if (sum == 0)
     return true;
   if (n == 0 && sum != 0)
     return false;

   // If last element is greater than sum, then
   // ignore it
   if (arr[n-1] > sum)
     return isSubsetSum (arr, n-1, sum);

   /* else, check if sum can be obtained by any of
      the following
      (a) including the last element
      (b) excluding the last element
   */
   return isSubsetSum (arr, n-1, sum) ||
          isSubsetSum (arr, n-1, sum-arr[n-1]);
}

// Returns true if arr[] can be partitioned in two
//  subsets of equal sum, otherwise false
bool findPartiion (int arr[], int n)
{
    // Calculate sum of the elements in array
    int sum = 0;
    for (int i = 0; i < n; i++)
       sum += arr[i];

    // If sum is odd, there cannot be two subsets
    // with equal sum
    if (sum%2 != 0)
       return false;
```

Aman Barnwal

```
    // Find if there is subset with sum equal to
    // half of total sum
    return isSubsetSum (arr, n, sum/2);
}

// Driver program to test above function
int main()
{
  int arr[] = {3, 1, 5, 9, 12};
  int n = sizeof(arr)/sizeof(arr[0]);
  if (findPartiion(arr, n) == true)
     printf("Can be divided into two subsets "
            "of equal sum");
  else
     printf("Can not be divided into two subsets"
            " of equal sum");
  return 0;
}
```

Java

```
// A recursive Java solution for partition problem
import java.io.*;

class Partition
{
    // A utility function that returns true if there is a
    // subset of arr[] with sun equal to given sum
    static boolean isSubsetSum (int arr[], int n, int sum)
    {
        // Base Cases
        if (sum == 0)
            return true;
        if (n == 0 && sum != 0)
            return false;

        // If last element is greater than sum, then ignore it
        if (arr[n-1] > sum)
            return isSubsetSum (arr, n-1, sum);

        /* else, check if sum can be obtained by any of
           the following
        (a) including the last element
        (b) excluding the last element
        */
        return isSubsetSum (arr, n-1, sum) ||
               isSubsetSum (arr, n-1, sum-arr[n-1]);
    }

    // Returns true if arr[] can be partitioned in two
    // subsets of equal sum, otherwise false
    static boolean findPartition (int arr[], int n)
    {
        // Calculate sum of the elements in array
        int sum = 0;
        for (int i = 0; i < n; i++)
            sum += arr[i];

        // If sum is odd, there cannot be two subsets
        // with equal sum
        if (sum%2 != 0)
            return false;

        // Find if there is subset with sum equal to half
        // of total sum
        return isSubsetSum (arr, n, sum/2);
    }

    /*Driver function to check for above function*/
    public static void main (String[] args)
    {

        int arr[] = {3, 1, 5, 9, 12};
        int n = arr.length;
        if (findPartition(arr, n) == true)
            System.out.println("Can be divided into two "+
                               "subsets of equal sum");
        else
            System.out.println("Can not be divided into " +
```

Aman Barnwal

```
                                "two subsets of equal sum");
    }
}
/* This code is contributed by Devesh Agrawal */
```

```
Can be divided into two subsets of equal sum
```

Time Complexity: O(2^n) In worst case, this solution tries two possibilities (whether to include or exclude) for every element.

Dynamic Programming Solution
The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array part[][] of size (sum/2)*(n+1). And we can construct the solution in bottom up manner such that every filled entry has following property

```
part[i][j] = true if a subset of {arr[0], arr[1], ..arr[j-1]} has sum
             equal to i, otherwise false
```

C/C++

```
// A Dynamic Programming based C program to partition problem
#include <stdio.h>

// Returns true if arr[] can be partitioned in two subsets of
// equal sum, otherwise false
bool findPartiion (int arr[], int n)
{
    int sum = 0;
    int i, j;

    // Caculcate sun of all elements
    for (i = 0; i < n; i++)
      sum += arr[i];

    if (sum%2 != 0)
       return false;

    bool part[sum/2+1][n+1];

    // initialize top row as true
    for (i = 0; i <= n; i++)
      part[0][i] = true;

    // initialize leftmost column, except part[0][0], as 0
    for (i = 1; i <= sum/2; i++)
      part[i][0] = false;

     // Fill the partition table in botton up manner
     for (i = 1; i <= sum/2; i++)
     {
       for (j = 1; j <= n; j++)
       {
         part[i][j] = part[i][j-1];
         if (i >= arr[j-1])
           part[i][j] = part[i][j] || part[i - arr[j-1]][j-1];
       }
     }

    /* // uncomment this part to print table
     for (i = 0; i <= sum/2; i++)
     {
       for (j = 0; j <= n; j++)
          printf ("%4d", part[i][j]);
       printf("\n");
     } */

     return part[sum/2][n];
}

// Driver program to test above funtion
int main()
{
  int arr[] = {3, 1, 1, 2, 2, 1};
  int n = sizeof(arr)/sizeof(arr[0]);
  if (findPartiion(arr, n) == true)
     printf("Can be divided into two subsets of equal sum");
  else
```

Aman Barnwal

```
     printf("Can not be divided into two subsets of equal sum");
  getchar();
  return 0;
}
```

Java

```
// A dynamic programming based Java program for partition problem
import java.io.*;

class Partition {

    // Returns true if arr[] can be partitioned in two subsets of
    // equal sum, otherwise false
    static boolean findPartition (int arr[], int n)
    {
        int sum = 0;
        int i, j;

        // Caculcate sun of all elements
        for (i = 0; i < n; i++)
            sum += arr[i];

        if (sum%2 != 0)
            return false;

        boolean part[][]=new boolean[sum/2+1][n+1];

        // initialize top row as true
        for (i = 0; i <= n; i++)
            part[0][i] = true;

        // initialize leftmost column, except part[0][0], as 0
        for (i = 1; i <= sum/2; i++)
            part[i][0] = false;

        // Fill the partition table in botton up manner
        for (i = 1; i <= sum/2; i++)
        {
            for (j = 1; j <= n; j++)
            {
                part[i][j] = part[i][j-1];
                if (i >= arr[j-1])
                    part[i][j] = part[i][j] ||
                                 part[i - arr[j-1]][j-1];
            }
        }

        /* // uncomment this part to print table
        for (i = 0; i <= sum/2; i++)
        {
            for (j = 0; j <= n; j++)
                printf ("%4d", part[i][j]);
            printf("\n");
        } */

        return part[sum/2][n];
    }

    /*Driver function to check for above function*/
    public static void main (String[] args)
    {
        int arr[] = {3, 1, 1, 2, 2,1};
        int n = arr.length;
        if (findPartition(arr, n) == true)
            System.out.println("Can be divided into two "
                               "subsets of equal sum");
        else
            System.out.println("Can not be divided into"
                           " two subsets of equal sum");

    }
}
/* This code is contributed by Devesh Agrawal */
```

```
Can be divided into two subsets of equal sum
```

Following diagram shows the values in partition table. The diagram is taken form the wiki page of partition problem.

Aman Barnwal

The entry part[i][j] indicates whether there is a subset of {arr[0], arr[1], .. arr[j-1]} that sums to i

	{}	{3}	{3,1}	{3,1,1}	{3,1,1,2}	{3,1,1,2,2}	{3,1,1,2,2,1}
0	True	True	True	True	True	True	True
1	False	False	True	True	True	True	True
2	False	False	False	True	True	True	True
3	False	True	True	True	True	True	True
4	False	False	True	True	True	True	True
5	False	False	False	True	True	True	True

Dynamic Programming table for
arr[] = {3,1,1,2,2,1}

Time Complexity: O(sum*n)
Auxiliary Space: O(sum*n)
Please note that this solution will not be feasible for arrays with big sum.

References:
http://en.wikipedia.org/wiki/Partition_problem

Aman Barnwal

Dynamic Programming | Set 19 (Word Wrap Problem)

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width.

The word processors like MS Word do task of placing line breaks. The idea is to have balanced lines. In other words, not have few lines with lots of extra spaces and some lines with small amount of extra spaces.

```
The extra spaces includes spaces put at the end of every line except the last one.
The problem is to minimize the following total cost.
 Cost of a line = (Number of extra spaces in the line)^3
 Total Cost = Sum of costs for all lines

For example, consider the following string and line width M = 15
 "Geeks for Geeks presents word wrap problem"

Following is the optimized arrangement of words in 3 lines
Geeks for Geeks
presents word
wrap problem

The total extra spaces in line 1, line 2 and line 3 are 0, 2 and 3 respectively.
So optimal value of total cost is 0 + 2*2 + 3*3 = 13
```

Please note that the total cost function is not sum of extra spaces, but sum of cubes (or square is also used) of extra spaces. The idea behind this cost function is to balance the spaces among lines. For example, consider the following two arrangement of same set of words:

1) There are 3 lines. One line has 3 extra spaces and all other lines have 0 extra spaces. Total extra spaces = 3 + 0 + 0 = 3. Total cost = 3*3*3 + 0*0*0 + 0*0*0 = 27.

2) There are 3 lines. Each of the 3 lines has one extra space. Total extra spaces = 1 + 1 + 1 = 3. Total cost = 1*1*1 + 1*1*1 + 1*1*1 = 3.

Total extra spaces are 3 in both scenarios, but second arrangement should be preferred because extra spaces are balanced in all three lines. The cost function with cubic sum serves the purpose because the value of total cost in second scenario is less.

Method 1 (Greedy Solution)
The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives optimal solution for many cases, but doesnt give optimal solution in all cases. For example, consider the following string aaa bb cc ddddd and line width as 6. Greedy method will produce following output.

```
aaa bb
cc
ddddd
```

Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So total cost is 0 + 64 + 1 = 65.

But the above solution is not the best solution. Following arrangement has more balanced spaces. Therefore less value of total cost function.

```
aaa
bb cc
ddddd
```

Extra spaces in the above 3 lines are 3, 1 and 1 respectively. So total cost is 27 + 1 + 1 = 29.

Despite being sub-optimal in some cases, the greedy approach is used by many word processors like MS Word and OpenOffice.org Writer.

Method 2 (Dynamic Programming)
The following Dynamic approach strictly follows the algorithm given in solution of Cormen book. First we compute costs of all possible lines in a 2D table lc[][]. The value lc[i][j] indicates the cost to put words from i to j in a single line where i and j are indexes of words in the input sequences. If a sequence of words from i to j cannot fit in a single line, then lc[i][j] is considered infinite (to avoid it from being a part of the solution). Once we have the lc[][] table constructed, we can calculate total cost using following recursive formula. In the following formula, C[j] is the optimized total cost for arranging words from 1 to j.

$$c[j] = \begin{cases} 0 & \text{if } j = 0\,, \\ \min\limits_{1 \le i \le j} (c[i-1] + lc[i,j]) & \text{if } j > 0\,. \end{cases}$$

The above recursion has overlapping subproblem property. For example, the solution of subproblem c(2) is used by c(3), C(4) and so on. So Dynamic Programming is used to store the results of subproblems. The array c[] can be computed from left to right, since each value depends only on earlier values.
To print the output, we keep track of what words go on what lines, we can keep a parallel p array that points to where each c value came from.

Aman Bamwal

The last line starts at word p[n] and goes through word n. The previous line starts at word p[p[n]] and goes through word p[n] 1, etc. The function printSolution() uses p[] to print the solution.
In the below program, input is an array l[] that represents lengths of words in a sequence. The value l[i] indicates length of the ith word (i starts from 1) in theinput sequence.

```
// A Dynamic programming solution for Word Wrap Problem
#include <limits.h>
#include <stdio.h>
#define INF INT_MAX

// A utility function to print the solution
int printSolution (int p[], int n);

// l[] represents lengths of different words in input sequence. For example,
// l[] = {3, 2, 2, 5} is for a sentence like "aaa bb cc ddddd".  n is size of
// l[] and M is line width (maximum no. of characters that can fit in a line)
void solveWordWrap (int l[], int n, int M)
{
    // For simplicity, 1 extra space is used in all below arrays

    // extras[i][j] will have number of extra spaces if words from i
    // to j are put in a single line
    int extras[n+1][n+1];

    // lc[i][j] will have cost of a line which has words from
    // i to j
    int lc[n+1][n+1];

    // c[i] will have total cost of optimal arrangement of words
    // from 1 to i
    int c[n+1];

    // p[] is used to print the solution.
    int p[n+1];

    int i, j;

    // calculate extra spaces in a single line.  The value extra[i][j]
    // indicates extra spaces if words from word number i to j are
    // placed in a single line
    for (i = 1; i <= n; i++)
    {
        extras[i][i] = M - l[i-1];
        for (j = i+1; j <= n; j++)
            extras[i][j] = extras[i][j-1] - l[j-1] - 1;
    }

    // Calculate line cost corresponding to the above calculated extra
    // spaces. The value lc[i][j] indicates cost of putting words from
    // word number i to j in a single line
    for (i = 1; i <= n; i++)
    {
        for (j = i; j <= n; j++)
        {
            if (extras[i][j] < 0)
                lc[i][j] = INF;
            else if (j == n && extras[i][j] >= 0)
                lc[i][j] = 0;
            else
                lc[i][j] = extras[i][j]*extras[i][j];
        }
    }

    // Calculate minimum cost and find minimum cost arrangement.
    //  The value c[j] indicates optimized cost to arrange words
    // from word number 1 to j.
    c[0] = 0;
    for (j = 1; j <= n; j++)
    {
        c[j] = INF;
        for (i = 1; i <= j; i++)
        {
            if (c[i-1] != INF && lc[i][j] != INF && (c[i-1] + lc[i][j] < c[j]))
            {
                c[j] = c[i-1] + lc[i][j];
                p[j] = i;
            }
        }
    }
```

Aman Barnwal

```
    printSolution(p, n);
}

int printSolution (int p[], int n)
{
    int k;
    if (p[n] == 1)
        k = 1;
    else
        k = printSolution (p, p[n]-1) + 1;
    printf ("Line number %d: From word no. %d to %d \n", k, p[n], n);
    return k;
}

// Driver program to test above functions
int main()
{
    int l[] = {3, 2, 2, 5};
    int n = sizeof(l)/sizeof(l[0]);
    int M = 6;
    solveWordWrap (l, n, M);
    return 0;
}
```

Output:

```
Line number 1: From word no. 1 to 1
Line number 2: From word no. 2 to 3
Line number 3: From word no. 4 to 4
```

Time Complexity: O(n^2)
Auxiliary Space: O(n^2) The auxiliary space used in the above program cane be optimized to O(n) (See the reference 2 for details)

References:
http://en.wikipedia.org/wiki/Word_wrap

Aman Barnwal

Dynamic Programming | Set 20 (Maximum Length Chain of Pairs)

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if b < c. Chain of pairs can be formed in this fashion. Find the longest chain which can be formed from a given set of pairs. Source: Amazon Interview | Set 2

For example, if the given pairs are {{5, 24}, {39, 60}, {15, 28}, {27, 40}, {50, 90} }, then the longest chain that can be formed is of length 3, and the chain is {{5, 24}, {27, 40}, {50, 90}}

This problem is a variation of standard Longest Increasing Subsequence problem. Following is a simple two step process.
1) Sort given pairs in increasing order of first (or smaller) element.
2) Now run a modified LIS process where we compare the second element of already finalized LIS with the first element of new LIS being constructed.

The following code is a slight modification of method 2 of this post.

```
#include<stdio.h>
#include<stdlib.h>

// Structure for a pair
struct pair
{
  int a;
  int b;
};

// This function assumes that arr[] is sorted in increasing order
// according the first (or smaller) values in pairs.
int maxChainLength( struct pair arr[], int n)
{
   int i, j, max = 0;
   int *mcl = (int*) malloc ( sizeof( int ) * n );

   /* Initialize MCL (max chain length) values for all indexes */
   for ( i = 0; i < n; i++ )
      mcl[i] = 1;

   /* Compute optimized chain length values in bottom up manner */
   for ( i = 1; i < n; i++ )
      for ( j = 0; j < i; j++ )
         if ( arr[i].a > arr[j].b && mcl[i] < mcl[j] + 1)
            mcl[i] = mcl[j] + 1;

   // mcl[i] now stores the maximum chain length ending with pair i

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

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

   return max;
}


/* Driver program to test above function */
int main()
{
    struct pair arr[] = { {5, 24}, {15, 25},
                          {27, 40}, {50, 60} };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of maximum size chain is %d\n",
           maxChainLength( arr, n ));
    return 0;
}
```

Output:

```
Length of maximum size chain is 3
```

Time Complexity: O(n^2) where n is the number of pairs.

The given problem is also a variation of Activity Selection problem and can be solved in (nLogn) time. To solve it as a activity selection problem, consider the first element of a pair as start time in activity selection problem, and the second element of pair as end time. Thanks to Palash for suggesting this approach.

Aman Barnwal

Aman Barnwal

Dynamic Programming | Set 22 (Box Stacking Problem)

You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Source: http://people.csail.mit.edu/bdean/6.046/dp/. The link also has video for explanation of solution.

The Box Stacking problem is a variation of LIS problem. We need to build a maximum height stack.

Following are the key points to note in the problem statement:
1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.
2) We can rotate boxes. For example, if there is a box with dimensions {1x2x3} where 1 is height, 23 is base, then there can be three possibilities, {1x2x3}, {2x1x3} and {3x1x2}.
3) We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.

Following is the **solution** based on DP solution of LIS problem.

1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.

2) Sort the above generated 3n boxes in decreasing order of base area.

3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.
MSH(i) = Maximum possible Stack Height with box i at top of stack
MSH(i) = { Max (MSH(j)) + height(i) } where j < i and width(j) > width(i) and depth(j) > depth(i).
If there is no such j then MSH(i) = height(i)

4) To get overall maximum height, we return max(MSH(i)) where 0 < i < n Following is C++ implementation of the above solution.

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

/* Representation of a box */
struct Box
{
  // h > height, w > width, d > depth
  int h, w, d;  // for simplicity of solution, always keep w <= d
};

// A utility function to get minimum of two intgers
int min (int x, int y)
{ return (x < y)? x : y; }

// A utility function to get maximum of two intgers
int max (int x, int y)
{ return (x > y)? x : y; }

/* Following function is needed for library function qsort(). We
   use qsort() to sort boxes in decreasing order of base area.
   Refer following link for help of qsort() and compare()
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare (const void *a, const void * b)
{
    return ( (*(Box *)b).d * (*(Box *)b).w )
           ( (*(Box *)a).d * (*(Box *)a).w );
}

/* Returns the height of the tallest stack that can be formed with give type of boxes */
int maxStackHeight( Box arr[], int n )
```

Aman Barnwal

```
{
   /* Create an array of all rotations of given boxes
      For example, for a box {1, 2, 3}, we consider three
      instances{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}} */
   Box rot[3*n];
   int index = 0;
   for (int i = 0; i < n; i++)
   {
      // Copy the original box
      rot[index] = arr[i];
      index++;

      // First rotation of box
      rot[index].h = arr[i].w;
      rot[index].d = max(arr[i].h, arr[i].d);
      rot[index].w = min(arr[i].h, arr[i].d);
      index++;

      // Second rotation of box
      rot[index].h = arr[i].d;
      rot[index].d = max(arr[i].h, arr[i].w);
      rot[index].w = min(arr[i].h, arr[i].w);
      index++;
   }

   // Now the number of boxes is 3n
   n = 3*n;

   /* Sort the array rot[] in decreasing order, using library
      function for quick sort */
   qsort (rot, n, sizeof(rot[0]), compare);

   // Uncomment following two lines to print all rotations
   // for (int i = 0; i < n; i++ )
   //    printf("%d x %d x %d\n", rot[i].h, rot[i].w, rot[i].d);

   /* Initialize msh values for all indexes
      msh[i] > Maximum possible Stack Height with box i on top */
   int msh[n];
   for (int i = 0; i < n; i++ )
      msh[i] = rot[i].h;

   /* Compute optimized msh values in bottom up manner */
   for (int i = 1; i < n; i++ )
      for (int j = 0; j < i; j++ )
         if ( rot[i].w < rot[j].w &&
              rot[i].d < rot[j].d &&
              msh[i] < msh[j] + rot[i].h
            )
         {
              msh[i] = msh[j] + rot[i].h;
         }


   /* Pick maximum of all msh values */
   int max = -1;
   for ( int i = 0; i < n; i++ )
      if ( max < msh[i] )
         max = msh[i];

   return max;
}

/* Driver program to test above function */
int main()
{
  Box arr[] = { {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32} };
  int n = sizeof(arr)/sizeof(arr[0]);

  printf("The maximum possible height of stack is %d\n",
         maxStackHeight (arr, n) );

  return 0;
}
```

Output:

```
The maximum possible height of stack is 60
```

In the above program, given input boxes are {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32}. Following are all rotations of the boxes in decreasing order of base area.

Aman Barnwal

```
10 x 12 x 32
12 x 10 x 32
32 x 10 x 12
4 x 6 x 7
4 x 5 x 6
6 x 4 x 7
5 x 4 x 6
7 x 4 x 6
6 x 4 x 5
1 x 2 x 3
2 x 1 x 3
3 x 1 x 2
```

The height 60 is obtained by boxes { {**3**, 1, 2}, {**1**, 2, 3}, {**6**, 4, 5}, {**4**, 5, 6}, {**4**, 6, 7}, {**32**, 10, 12}, {**10**, 12, 32}}

Time Complexity: O(n^2)
Auxiliary Space: O(n)

Aman Barnwal

Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141, ..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0 \text{ and } F_1 = 1.$$

Write a function *int fib(int n)* that returns F_n. For example, if *n* = 0, then *fib()* should return 0. If n = 1, then it should return 1. For n > 1, it should return $F_{n-1} + F_{n-2}$

Following are different methods to get the nth Fibonacci number.

Method 1 (Use recursion)
A simple method that is a direct recusrive implementation mathematical recurance relation given above.

```
#include<stdio.h>
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

Time Complexity: T(n) = T(n-1) + T(n-2) which is exponential.
We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.

```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /      \                /     \
         fib(3)      fib(2)         fib(2)    fib(1)
        /     \        /    \        /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

Extra Space: O(n) if we consider the function call stack size, otherwise O(1).

Method 2 (Use Dynamic Programming)
We can avoid the repeated work done is the method 1 by storing the Fibonacci numbers calculated so far.

```
#include<stdio.h>

int fib(int n)
{
  /* Declare an array to store Fibonacci numbers. */
  int f[n+1];
  int i;

  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }
```

Aman Barnwal

```
  return f[n];
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

Time Complexity: O(n)
Extra Space: O(n)

Method 3 (Space Otimized Method 2)
We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibannaci number in series.

```
#include<stdio.h>
int fib(int n)
{
  int a = 0, b = 1, c, i;
  if( n == 0)
    return a;
  for (i = 2; i <= n; i++)
  {
     c = a + b;
     a = b;
     b = c;
  }
  return b;
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

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

Method 4 (Using power of the matrix {{1,1},{1,0}})
This another O(n) which relies on the fact that if we n times multiply the matrix M = {{1,1},{1,0}} to itself (in other words calculate power(M, n)), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

```
#include <stdio.h>

/* Helper function that multiplies 2 matricies F and M of size 2*2, and
  puts the multiplication result back to F[][] */
void multiply(int F[2][2], int M[2][2]);

/* Helper function that calculates F[][] raise to the power n and puts the
  result in F[][]
  Note that this function is desinged only for fib() and won't work as general
  power function */
void power(int F[2][2], int n);

int fib(int n)
{
  int F[2][2] = {{1,1},{1,0}};
  if (n == 0)
      return 0;
  power(F, n-1);

  return F[0][0];
}

void multiply(int F[2][2], int M[2][2])
{
  int x =  F[0][0]*M[0][0] + F[0][1]*M[1][0];
  int y =  F[0][0]*M[0][1] + F[0][1]*M[1][1];
```

Aman Barnwal

```
  int z =  F[1][0]*M[0][0] + F[1][1]*M[1][0];
  int w =  F[1][0]*M[0][1] + F[1][1]*M[1][1];

  F[0][0] = x;
  F[0][1] = y;
  F[1][0] = z;
  F[1][1] = w;
}

void power(int F[2][2], int n)
{
  int i;
  int M[2][2] = {{1,1},{1,0}};

  // n - 1 times multiply the matrix to {{1,0},{0,1}}
  for (i = 2; i <= n; i++)
      multiply(F, M);
}

/* Driver program to test above function */
int main()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

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

Method 5 (Optimized Method 4)
The method 4 can be optimized to work in O(Logn) time complexity. We can do recursive multiplication to get power(M, n) in the prevous method (Similar to the optimization done in this post)

```
#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);

void power(int F[2][2], int n);

/* function that returns nth Fibonacci number */
int fib(int n)
{
  int F[2][2] = {{1,1},{1,0}};
  if (n == 0)
    return 0;
  power(F, n-1);
  return F[0][0];
}

/* Optimized version of power() in method 4 */
void power(int F[2][2], int n)
{
  if( n == 0 || n == 1)
      return;
  int M[2][2] = {{1,1},{1,0}};

  power(F, n/2);
  multiply(F, F);

  if (n%2 != 0)
     multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
  int x =  F[0][0]*M[0][0] + F[0][1]*M[1][0];
  int y =  F[0][0]*M[0][1] + F[0][1]*M[1][1];
  int z =  F[1][0]*M[0][0] + F[1][1]*M[1][0];
  int w =  F[1][0]*M[0][1] + F[1][1]*M[1][1];

  F[0][0] = x;
  F[0][1] = y;
  F[1][0] = z;
  F[1][1] = w;
}

/* Driver program to test above function */
```

Aman Barnwal

```
int main()
{
  int n = 9;
  printf("%d", fib(9));
  getchar();
  return 0;
}
```

Time Complexity: **O(Logn)**
Extra Space: O(Logn) if we consider the function call stack size, otherwise O(1).

References:
http://en.wikipedia.org/wiki/Fibonacci_number
http://www.ics.uci.edu/~eppstein/161/960109.html

Aman Barnwal

Minimum number of jumps to reach end

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then cannot move through that element.

Example:

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1-> 3 -> 8 ->9)
```

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps eg to 5 or 8 or 9.

Method 1 (Naive Recursive Approach)
A naive approach is to start from the first element and recursively call for all the elements reachable from first element. The minimum number of jumps to reach end from first can be calculated using minimum number of jumps needed to reach end from the elements reachable from first.

minJumps(start, end) = Min (minJumps(k, end)) for all k reachable from start

```
#include <stdio.h>
#include <limits.h>

// Returns minimum number of jumps to reach arr[h] from arr[l]
int minJumps(int arr[], int l, int h)
{
   // Base case: when source and destination are same
   if (h == l)
     return 0;

   // When nothing is reachable from the given source
   if (arr[l] == 0)
     return INT_MAX;

   // Traverse through all the points reachable from arr[l]. Recursively
   // get the minimum number of jumps needed to reach arr[h] from these
   // reachable points.
   int min = INT_MAX;
   for (int i = l+1; i <= h && i <= l + arr[l]; i++)
   {
       int jumps = minJumps(arr, i, h);
       if(jumps != INT_MAX && jumps + 1 < min)
           min = jumps + 1;
   }

   return min;
}

// Driver program to test above function
int main()
{
  int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Minimum number of jumps to reach end is %d ", minJumps(arr, 0, n-1));
  return 0;
}
```

If we trace the execution of this method, we can see that there will be overlapping subproblems. For example, minJumps(3, 9) will be called two times as arr[3] is reachable from arr[1] and arr[2]. So this problem has both properties (optimal substructure and overlapping subproblems) of Dynamic Programming.

Method 2 (Dynamic Programming)
In this method, we build a jumps[] array from left to right such that jumps[i] indicates the minimum number of jumps needed to reach arr[i] from arr[0]. Finally, we return jumps[n-1].

```
#include <stdio.h>
#include <limits.h>

int min(int x, int y) { return (x < y)? x: y; }

// Returns minimum number of jumps to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
    int *jumps = new int[n];  // jumps[n-1] will hold the result
    int i, j;
```

Aman Barnwal

```
    if (n == 0 || arr[0] == 0)
        return INT_MAX;

    jumps[0] = 0;

    // Find the minimum number of jumps to reach arr[i]
    // from arr[0], and assign this value to jumps[i]
    for (i = 1; i < n; i++)
    {
        jumps[i] = INT_MAX;
        for (j = 0; j < i; j++)
        {
            if (i <= j + arr[j] && jumps[j] != INT_MAX)
            {
                jumps[i] = min(jumps[i], jumps[j] + 1);
                break;
            }
        }
    }
    return jumps[n-1];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 1, 0, 9};
    int size = sizeof(arr)/sizeof(int);
    printf("Minimum number of jumps to reach end is %d ", minJumps(arr,size));
    return 0;
}
```

Output:

```
Minimum number of jumps to reach end is 3
```

Thanks to paras for suggesting this method.

Time Complexity: O(n^2)

Method 3 (Dynamic Programming)
In this method, we build jumps[] array from right to left such that jumps[i] indicates the minimum number of jumps needed to reach arr[n-1] from arr[i]. Finally, we return arr[0].

```
int minJumps(int arr[], int n)
{
    int *jumps = new int[n];  // jumps[0] will hold the result
    int min;

    // Minimum number of jumps needed to reach last element
    // from last elements itself is always 0
    jumps[n-1] = 0;

    int i, j;

    // Start from the second element, move from right to left
    // and construct the jumps[] array where jumps[i] represents
    // minimum number of jumps needed to reach arr[m-1] from arr[i]
    for (i = n-2; i >=0; i--)
    {
        // If arr[i] is 0 then arr[n-1] can't be reached from here
        if (arr[i] == 0)
            jumps[i] = INT_MAX;

        // If we can direcly reach to the end point from here then
        // jumps[i] is 1
        else if (arr[i] >= n - i - 1)
            jumps[i] = 1;

        // Otherwise, to find out the minimum number of jumps needed
        // to reach arr[n-1], check all the points reachable from here
        // and jumps[] value for those points
        else
        {
            min = INT_MAX;  // initialize min value

            // following loop checks with all reachable points and
```

Aman Barnwal

```
            // takes the minimum
            for (j = i+1; j < n && j <= arr[i] + i; j++)
            {
                if (min > jumps[j])
                    min = jumps[j];
            }

            // Handle overflow
            if (min != INT_MAX)
              jumps[i] = min + 1;
            else
              jumps[i] = min; // or INT_MAX
        }
    }

    return jumps[0];
}
```

Time Complexity: O(n^2) in worst case.

Thanks to Ashish for suggesting this solution.

Aman Barnwal

Maximum size square sub-matrix with all 1s

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.

```
   0  1  1  0  1
   1  1  0  1  0
   0  1  1  1  0
   1  1  1  1  0
   1  1  1  1  1
   0  0  0  0  0
```

The maximum square sub-matrix with all set bits is

```
    1  1  1
    1  1  1
    1  1  1
```

Algorithm:
Let the given binary matrix be M[R][C]. The idea of the algorithm is to construct an auxiliary size matrix S[][] in which each entry S[i][j] represents size of the square sub-matrix with all 1s including M[i][j] where M[i][j] is the rightmost and bottommost entry in sub-matrix.

```
1) Construct a sum matrix S[R][C] for the given M[R][C].
     a) Copy first row and first columns as it is from M[][] to S[][]
     b) For other entries, use following expressions to construct S[][]
         If M[i][j] is 1 then
            S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1
         Else /*If M[i][j] is 0*/
            S[i][j] = 0
2) Find the maximum entry in S[R][C]
3) Using the value and coordinates of maximum entry in S[i], print
   sub-matrix of M[][]
```

For the given M[R][C] in above example, constructed S[R][C] would be:

```
   0  1  1  0  1
   1  1  0  1  0
   0  1  1  1  0
   1  1  2  2  0
   1  2  2  3  1
   0  0  0  0  0
```

The value of maximum entry in above matrix is 3 and coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

```
#include<stdio.h>
#define bool int
#define R 6
#define C 5

void printMaxSubSquare(bool M[R][C])
{
  int i,j;
  int S[R][C];
  int max_of_s, max_i, max_j;

  /* Set first column of S[][]*/
  for(i = 0; i < R; i++)
     S[i][0] = M[i][0];

  /* Set first row of S[][]*/
  for(j = 0; j < C; j++)
     S[0][j] = M[0][j];

  /* Construct other entries of S[][]*/
  for(i = 1; i < R; i++)
  {
    for(j = 1; j < C; j++)
    {
      if(M[i][j] == 1)
        S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1;
      else
        S[i][j] = 0;
    }
  }
```

Aman Barnwal

```
  /* Find the maximum entry, and indexes of maximum entry
     in S[][] */
  max_of_s = S[0][0]; max_i = 0; max_j = 0;
  for(i = 0; i < R; i++)
  {
    for(j = 0; j < C; j++)
    {
      if(max_of_s < S[i][j])
      {
         max_of_s = S[i][j];
         max_i = i;
         max_j = j;
      }
    }
  }

  printf("\n Maximum size sub-matrix is: \n");
  for(i = max_i; i > max_i - max_of_s; i--)
  {
    for(j = max_j; j > max_j - max_of_s; j--)
    {
      printf("%d ", M[i][j]);
    }
    printf("\n");
  }
}

/* UTILITY FUNCTIONS */
/* Function to get minimum of three values */
int min(int a, int b, int c)
{
  int m = a;
  if (m > b)
    m = b;
  if (m > c)
    m = c;
  return m;
}

/* Driver function to test above functions */
int main()
{
  bool M[R][C] =  {{0, 1, 1, 0, 1},
                   {1, 1, 0, 1, 0},
                   {0, 1, 1, 1, 0},
                   {1, 1, 1, 1, 0},
                   {1, 1, 1, 1, 1},
                   {0, 0, 0, 0, 0}};

  printMaxSubSquare(M);
  getchar();
}
```

Aman Barnwal

Time Complexity: O(m*n) where m is number of rows and n is number of columns in the given matrix.
Auxiliary Space: O(m*n) where m is number of rows and n is number of columns in the given matrix.
Algorithmic Paradigm: Dynamic Programming

Ugly Numbers

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15,
shows the first 11 ugly numbers. By convention, 1 is included.
Write a program to find and print the 150th ugly number.

METHOD 1 (Simple)
Thanks to Nedylko Draganov for suggesting this solution.

Algorithm:
Loop for all positive integers until ugly number count is smaller than n, if an integer is ugly than increment ugly number count.

To check if a number is ugly, divide the number by greatest divisible powers of 2, 3 and 5, if the number becomes 1 then it is an ugly number otherwise not.

For example, let us see how to check for 300 is ugly or not. Greatest divisible power of 2 is 4, after dividing 300 by 4 we get 75. Greatest divisible power of 3 is 3, after dividing 75 by 3 we get 25. Greatest divisible power of 5 is 25, after dividing 25 by 25 we get 1. Since we get 1 finally, 300 is ugly number.

Implementation:

```
# include<stdio.h>
# include<stdlib.h>

/*This function divides a by greatest divisible
  power of b*/
int maxDivide(int a, int b)
{
  while (a%b == 0)
   a = a/b;
  return a;
}

/* Function to check if a number is ugly or not */
int isUgly(int no)
{
  no = maxDivide(no, 2);
  no = maxDivide(no, 3);
  no = maxDivide(no, 5);

  return (no == 1)? 1 : 0;
}

/* Function to get the nth ugly number*/
int getNthUglyNo(int n)
{
  int i = 1;
  int count = 1;   /* ugly number count */

  /*Check for all integers untill ugly count
    becomes n*/
  while (n > count)
  {
    i++;
    if (isUgly(i))
      count++;
  }
  return i;
}

/* Driver program to test above functions */
int main()
{
    unsigned no = getNthUglyNo(150);
    printf("150th ugly no. is %d ",  no);
    getchar();
    return 0;
}
```

This method is not time efficient as it checks for all integers until ugly number count becomes n, but space complexity of this method is O(1)

METHOD 2 (Use Dynamic Programming)
Here is a time efficient solution with O(n) extra space. The ugly-number sequence is 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15,

Aman Barnwal

because every number can only be divided by 2, 3, 5, one way to look at the sequence is to split the sequence to three groups as below:
(1) 12, 22, 32, 42, 52,
(2) 13, 23, 33, 43, 53,
(3) 15, 25, 35, 45, 55,

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5,) multiply 2, 3, 5. Then we use similar merge method as merge sort, to get every ugly number from the three subsequence. Every step we choose the smallest one, and move one step after.

Algorithm:

```
1 Declare an array for ugly numbers:  ugly[150]
2 Initialize first ugly no:  ugly[0] = 1
3 Initialize three array index variables i2, i3, i5 to point to
   1st element of the ugly array:
        i2 = i3 = i5 =0;
4 Initialize 3 choices for the next ugly no:
         next_mulitple_of_2 = ugly[i2]*2;
         next_mulitple_of_3 = ugly[i3]*3
         next_mulitple_of_5 = ugly[i5]*5;
5 Now go in a loop to fill all ugly numbers till 150:
For (i = 1; i < 150; i++ )
{
    /* These small steps are not optimized for good
      readability. Will optimize them in C program */
    next_ugly_no  = Min(next_mulitple_of_2,
                                next_mulitple_of_3,
                                next_mulitple_of_5);
    if  (next_ugly_no  == next_mulitple_of_2)
    {
        i2 = i2 + 1;
        next_mulitple_of_2 = ugly[i2]*2;
    }
    if  (next_ugly_no  == next_mulitple_of_3)
    {
        i3 = i3 + 1;
        next_mulitple_of_3 = ugly[i3]*3;
     }
     if  (next_ugly_no  == next_mulitple_of_5)
     {
        i5 = i5 + 1;
        next_mulitple_of_5 = ugly[i5]*5;
     }
     ugly[i] =  next_ugly_no
}/* end of for loop */
6.return next_ugly_no
```

Example:
Let us see how it works

```
initialize
   ugly[] =  | 1 |
   i2 =  i3 = i5 = 0;

First iteration
   ugly[1] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
           = Min(2, 3, 5)
           = 2
   ugly[] =  | 1 | 2 |
   i2 = 1,  i3 = i5 = 0  (i2 got incremented )

Second iteration
    ugly[2] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
            = Min(4, 3, 5)
            = 3
    ugly[] =  | 1 | 2 | 3 |
    i2 = 1,  i3 =  1, i5 = 0  (i3 got incremented )

Third iteration
    ugly[3] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
            = Min(4, 6, 5)
            = 4
    ugly[] =  | 1 | 2 | 3 |  4 |
    i2 = 2,  i3 =  1, i5 = 0  (i2 got incremented )

Fourth iteration
    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
            = Min(6, 6, 5)
            = 5
    ugly[] =  | 1 | 2 | 3 |  4 | 5 |
```

Aman Barnwal

```
    i2 = 2,  i3 =  1, i5 = 1  (i5 got incremented )

Fifth iteration
    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(6, 6, 10)
              = 6
    ugly[] =  | 1 | 2 | 3 |  4 | 5 | 6 |
    i2 = 3,  i3 =  2, i5 = 1  (i2 and i3 got incremented )

Will continue same way till I < 150
```

Program:

```
# include<stdio.h>
# include<stdlib.h>
# define bool int

/* Function to find minimum of 3 numbers */
unsigned min(unsigned , unsigned , unsigned );

/* Function to get the nth ugly number*/
unsigned getNthUglyNo(unsigned n)
{
    unsigned *ugly =
              (unsigned *)(malloc (sizeof(unsigned)*n));
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned i;
    unsigned next_multiple_of_2 = 2;
    unsigned next_multiple_of_3 = 3;
    unsigned next_multiple_of_5 = 5;
    unsigned next_ugly_no = 1;
    *(ugly+0) = 1;

    for(i=1; i<n; i++)
    {
       next_ugly_no = min(next_multiple_of_2,
                          next_multiple_of_3,
                          next_multiple_of_5);
       *(ugly+i) = next_ugly_no;
       if(next_ugly_no == next_multiple_of_2)
       {
           i2 = i2+1;
           next_multiple_of_2 = *(ugly+i2)*2;
       }
       if(next_ugly_no == next_multiple_of_3)
       {
           i3 = i3+1;
           next_multiple_of_3 = *(ugly+i3)*3;
       }
       if(next_ugly_no == next_multiple_of_5)
       {
           i5 = i5+1;
           next_multiple_of_5 = *(ugly+i5)*5;
       }
    } /*End of for loop (i=1; i<n; i++) */
    return next_ugly_no;
}

/* Function to find minimum of 3 numbers */
unsigned min(unsigned a, unsigned b, unsigned c)
{
    if(a <= b)
    {
      if(a <= c)
        return a;
      else
        return c;
    }
    if(b <= c)
      return b;
    else
      return c;
}

/* Driver program to test above functions */
int main()
{
    unsigned no = getNthUglyNo(150);
    printf("%dth ugly no. is %d ", 150, no);
    getchar();
    return 0;
```

Aman Barnwal

```
}
```

Algorithmic Paradigm: Dynamic Programming
Time Complexity: O(n)
Storage Complexity: O(n)

Aman Barnwal

Largest Sum Contiguous Subarray

Write an efficient C program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

Kadanes Algorithm:

```
Initialize:
    max_so_far = 0
    max_ending_here = 0

Loop for each element of the array
  (a) max_ending_here = max_ending_here + a[i]
  (b) if(max_ending_here < 0)
            max_ending_here = 0
  (c) if(max_so_far < max_ending_here)
            max_so_far = max_ending_here
return max_so_far
```

Explanation:
Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

```
    Lets take the example:
    {-2, -3, 4, -1, -2, 1, 5, -3}

    max_so_far = max_ending_here = 0

    for i=0,  a[0] =  -2
    max_ending_here = max_ending_here + (-2)
    Set max_ending_here = 0 because max_ending_here < 0

    for i=1,  a[1] =  -3
    max_ending_here = max_ending_here + (-3)
    Set max_ending_here = 0 because max_ending_here < 0

    for i=2,  a[2] =  4
    max_ending_here = max_ending_here + (4)
    max_ending_here = 4
    max_so_far is updated to 4 because max_ending_here greater
    than max_so_far which was 0 till now

    for i=3,  a[3] =  -1
    max_ending_here = max_ending_here + (-1)
    max_ending_here = 3

    for i=4,  a[4] =  -2
    max_ending_here = max_ending_here + (-2)
    max_ending_here = 1

    for i=5,  a[5] =  1
    max_ending_here = max_ending_here + (1)
    max_ending_here = 2

    for i=6,  a[6] =  5
    max_ending_here = max_ending_here + (5)
    max_ending_here = 7
    max_so_far is updated to 7 because max_ending_here is
    greater than max_so_far

    for i=7,  a[7] =  -3
    max_ending_here = max_ending_here + (-3)
    max_ending_here = 4
```

Program:

C++

```
// C++ program to print largest contiguous array sum
#include<iostream>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = 0, max_ending_here = 0;

    for (int i = 0; i < size; i++)
    {
```

Aman Barnwal

```
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    cout << "Maximum contiguous sum is \n" << max_sum;
    return 0;
}
```

Python

```
# Python program to find maximum contiguous subarray

# Function to find the maximum contiguous subarray
def maxSubArraySum(a,size):

    max_so_far = 0
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_ending_here < 0:
            max_ending_here = 0

        if (max_so_far < max_ending_here):
            max_so_far = max_ending_here

    return max_so_far

# Driver function to check the above function
a = [-2, -3, 4, -1, -2, 1, 5, -3]
print"Maximum contiguous sum is", maxSubArraySum(a,len(a))

#This code is contributed by _Devesh Agrawal_
```

```
Maximum contiguous sum is 7
```

Notes:
Algorithm doesn't work for all negative numbers. It simply returns 0 if all numbers are negative. For handling this we can add an extra phase before actual implementation. The phase will look if all numbers are negative, if they are it will return maximum of them (or smallest in terms of absolute value). There may be other ways to handle it though.

Above program can be optimized further, if we compare max_so_far with max_ending_here only if max_ending_here is greater than 0.

C++

```
int maxSubArraySum(int a[], int size)
{
   int max_so_far = 0, max_ending_here = 0;
   for (int i = 0; i < size; i++)
   {
       max_ending_here = max_ending_here + a[i];
       if (max_ending_here < 0)
           max_ending_here = 0;

       /* Do not compare for all elements. Compare only
          when  max_ending_here > 0 */
       else if (max_so_far < max_ending_here)
           max_so_far = max_ending_here;
   }
   return max_so_far;
}
```

Python

Aman Barnwal

```
def maxSubArraySum(a,size):

    max_so_far = 0
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_ending_here < 0:
            max_ending_here = 0

        # Do not compare for all elements. Compare only
        # when  max_ending_here > 0
        elif (max_so_far < max_ending_here):
            max_so_far = max_ending_here

    return max_so_far
```

Time Complexity: O(n)
Algorithmic Paradigm: Dynamic Programming

Following is another simple implementation suggested by **Mohit Kumar**. The implementation handles the case when all numbers in array are negative.

Aman Barnwal

Longest Palindromic Substring | Set 1

Given a string, find the longest substring which is palindrome. For example, if the given string is forgeeksskeegfor, the output should be geeksskeeg.

Method 1 (Brute Force)
The simple approach is to check each substring whether the substring is a palindrome or not. We can run three loops, the outer two loops pick all substrings one by one by fixing the corner characters, the inner loop checks whether the picked substring is palindrome or not.

Time complexity: O (n^3)
Auxiliary complexity: O (1)

Method 2 (Dynamic Programming)
The time complexity can be reduced by storing results of subproblems. The idea is similar to this post. We maintain a boolean table[n][n] that is filled in bottom up manner. The value of table[i][j] is true, if the substring is palindrome, otherwise false. To calculate table[i][j], we first check the value of table[i+1][j-1], if the value is true and str[i] is same as str[j], then we make table[i][j] true. Otherwise, the value of table[i][j] is made false.

```
// A dynamic programming solution for longest palindr.
// This code is adopted from following link
// http://www.leetcode.com/2011/11/longest-palindromic-substring-part-i.html

#include <stdio.h>
#include <string.h>

// A utility function to print a substring str[low..high]
void printSubStr( char* str, int low, int high )
{
    for( int i = low; i <= high; ++i )
        printf("%c", str[i]);
}

// This function prints the longest palindrome substring
// of str[].
// It also returns the length of the longest palindrome
int longestPalSubstr( char *str )
{
    int n = strlen( str ); // get length of input string

    // table[i][j] will be false if substring str[i..j]
    // is not palindrome.
    // Else table[i][j] will be true
    bool table[n][n];
    memset(table, 0, sizeof(table));

    // All substrings of length 1 are palindromes
    int maxLength = 1;
    for (int i = 0; i < n; ++i)
        table[i][i] = true;

    // check for sub-string of length 2.
    int start = 0;
    for (int i = 0; i < n-1; ++i)
    {
        if (str[i] == str[i+1])
        {
            table[i][i+1] = true;
            start = i;
            maxLength = 2;
        }
    }

    // Check for lengths greater than 2. k is length
    // of substring
    for (int k = 3; k <= n; ++k)
    {
        // Fix the starting index
        for (int i = 0; i < n-k+1 ; ++i)
        {
            // Get the ending index of substring from
            // starting index i and length k
            int j = i + k - 1;

            // checking for sub-string from ith index to
            // jth index iff str[i+1] to str[j-1] is a
            // palindrome
            if (table[i+1][j-1] && str[i] == str[j])
            {
                table[i][j] = true;
```

Aman Barnwal

```
                if (k > maxLength)
                {
                    start = i;
                    maxLength = k;
                }
            }
        }
    }

    printf("Longest palindrome substring is: ");
    printSubStr( str, start, start + maxLength - 1 );

    return maxLength; // return length of LPS
}

// Driver program to test above functions
int main()
{
    char str[] = "forgeeksskeegfor";
    printf("\nLength is: %d\n", longestPalSubstr( str ) );
    return 0;
}
```

Output:

```
Longest palindrome substring is: geeksskeeg
Length is: 10
```

Time complexity: O (n^2)
Auxiliary Space: O (n^2)

We will soon be adding more optimized methods as separate posts.

Aman Barnwal

Dynamic Programming | Set 23 (BellmanFord Algorithm)

Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed Dijkstras algorithm for this problem. Dijksras algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). *Dijkstra doesnt work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

Algorithm
Following are the detailed steps.

Input: Graph and a source vertex *src*
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

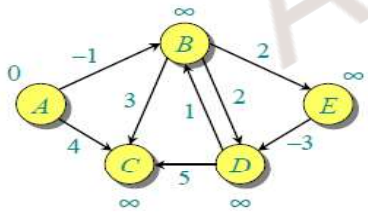
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
..**a)** Do following for each edge u-v
If dist[v] > dist[u] + weight of edge uv, then update dist[v]
.dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
If dist[v] > dist[u] + weight of edge uv, then Graph contains negative weight cycle
The idea of step 3 is, step 2 guarantees shortest distances if graph doesnt contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

How does this work? Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-nost 2 edges, and so on. After the ith iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| 1 edges in any simple path, that is why the outer loop runs |v| 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges (Proof is simple, you can refer this or MIT Video Lecture)

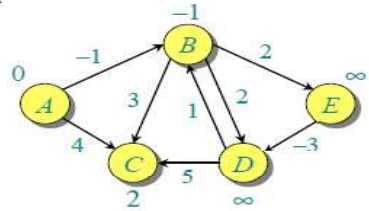
Example
Let us understand the algorithm with following example graph. The images are taken from this source.

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*



A	B	C	D	E
0	∞	∞	∞	∞

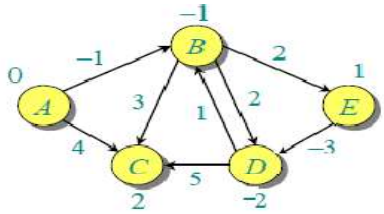
Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).

Aman Bhanwal



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations dont update the distances.

Implementation:

C++

```
// A C / C++ program for Bellman-Ford's single source
// shortest path algorithm.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
         (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
       (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm.  The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;
    dist[src] = 0;
```

Aman Barnwal

```
    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles.  The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle.  If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }

    printArr(dist, V);

    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5;  // Number of vertices in graph
    int E = 8;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;
```

Aman Barnwal

```
    BellmanFord(graph, 0);

    return 0;
}
```

Java

```
// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0;
        }
    };

    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // The main function that finds shortest distances from src
    // to all other vertices using Bellman-Ford algorithm.  The
    // function also detects negative weight cycle
    void BellmanFord(Graph graph,int src)
    {
        int V = graph.V, E = graph.E;
        int dist[] = new int[V];

        // Step 1: Initialize distances from src to all other
        // vertices as INFINITE
        for (int i=0; i<V; ++i)
            dist[i] = Integer.MAX_VALUE;
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple
        // shortest path from src to any other vertex can
        // have at-most |V| - 1 edges
        for (int i=1; i<V; ++i)
        {
            for (int j=0; j<E; ++j)
            {
                int u = graph.edge[j].src;
                int v = graph.edge[j].dest;
                int weight = graph.edge[j].weight;
                if (dist[u]!=Integer.MAX_VALUE &&
                    dist[u]+weight<dist[v])
                    dist[v]=dist[u]+weight;
            }
        }

        // Step 3: check for negative-weight cycles.  The above
        // step guarantees shortest distances if graph doesn't
        // contain negative weight cycle. If we get a shorter
        //  path, then there is a cycle.
        for (int j=0; j<E; ++j)
        {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u]!=Integer.MAX_VALUE &&
                dist[u]+weight<dist[v])
              System.out.println("Graph contains negative weight cycle");
```

Aman Barnwal

```
        }
        printArr(dist, V);
    }

    // A utility function used to print the solution
    void printArr(int dist[], int V)
    {
        System.out.println("Vertex   Distance from Source");
        for (int i=0; i<V; ++i)
            System.out.println(i+"\t\t"+dist[i]);
    }

    // Driver method to test above function
    public static void main(String[] args)
    {
        int V = 5;  // Number of vertices in graph
        int E = 8;  // Number of edges in graph

        Graph graph = new Graph(V, E);

        // add edge 0-1 (or A-B in above figure)
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = -1;

        // add edge 0-2 (or A-C in above figure)
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 4;

        // add edge 1-2 (or B-C in above figure)
        graph.edge[2].src = 1;
        graph.edge[2].dest = 2;
        graph.edge[2].weight = 3;

        // add edge 1-3 (or B-D in above figure)
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 2;

        // add edge 1-4 (or A-E in above figure)
        graph.edge[4].src = 1;
        graph.edge[4].dest = 4;
        graph.edge[4].weight = 2;

        // add edge 3-2 (or D-C in above figure)
        graph.edge[5].src = 3;
        graph.edge[5].dest = 2;
        graph.edge[5].weight = 5;

        // add edge 3-1 (or D-B in above figure)
        graph.edge[6].src = 3;
        graph.edge[6].dest = 1;
        graph.edge[6].weight = 1;

        // add edge 4-3 (or E-D in above figure)
        graph.edge[7].src = 4;
        graph.edge[7].dest = 3;
        graph.edge[7].weight = -3;

        graph.BellmanFord(graph, 0);
    }
}
// Contributed by Aakash Hasija
```

```
Vertex   Distance from Source
0                0
1                -1
2                2
3                -2
4                1
```

Notes

1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.

2) Bellman-Ford works better (better than Dijksras) for distributed systems. Unlike Dijksras where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

Arnan Barnwal

Exercise

1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.

2) Can we use Dijksras algorithm for shortest paths for graphs with negative weights one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijksras algorithm for the modified graph. Will this algorithm work?

References:

http://www.youtube.com/watch?v=Ttezuzs39nk
http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf

Aman Barnwal

Dynamic Programming | Set 24 (Optimal Binary Search Tree)

Given a sorted array *keys[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches to *keys[i]*. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

Example 1

```
Input:  keys[] = {10, 12}, freq[] = {34, 50}
There can be following two possible BSTs
        10                       12
          \                     /
           12                 10
          I                    II
Frequency of searches of 10 and 12 are 34 and 50 respectively.
The cost of tree I is 34*1 + 50*2 = 134
The cost of tree II is 50*1 + 34*2 = 118
```

Example 2

```
Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
There can be following possible BSTs
    10               12                 20         10              20
      \             /    \              /             \            /
      12          10     20           12               20         10
        \                             /                /          \
         20                         10               12            12
     I              II              III             IV            V
Among all possible BSTs, cost of the fifth BST is minimum.
Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142
```

1) Optimal Substructure:
The optimal cost for freq[i..j] can be recursively calculated using following formula.

$$optCost(i,j) = \sum_{k=i}^{j} freq[k] + \min_{r=i}^{j}[optCost(i,r-1) + optCost(r+1,j)]$$

We need to calculate ***optCost(0, n-1)*** to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make *rth* node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.
We add sum of frequencies from i to j (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A naive recursive implementation of optimal binary search tree problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal binary search tree
int optCost(int freq[], int i, int j)
{
   // Base cases
   if (j < i)      // If there are no elements in this subarray
     return 0;
   if (j == i)     // If there is one element in this subarray
     return freq[i];

   // Get sum of freq[i], freq[i+1], ... freq[j]
   int fsum = sum(freq, i, j);

   // Initialize minimum value
   int min = INT_MAX;

   // One by one consider all elements as root and recursively find cost
   // of the BST, compare the cost with min and update min if needed
   for (int r = i; r <= j; ++r)
   {
       int cost = optCost(freq, i, r-1) + optCost(freq, r+1, j);
       if (cost < min)
          min = cost;
   }

   // Return minimum value
```

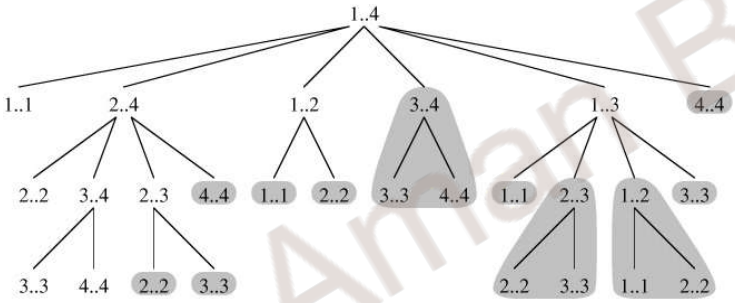
Aman Barnwal

```
   return min + fsum;
}

// The main function that calculates minimum cost of a Binary Search Tree.
// It mainly uses optCost() to find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
     // Here array keys[] is assumed to be sorted in increasing order.
     // If keys[] is not sorted, then add code to sort keys, and rearrange
     // freq[] accordingly.
     return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

```
Cost of Optimal BST is 142
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for freq[1..4].



Aman Barnwal

Since same suproblems are called again, this problem has Overlapping Subprolems property. So optimal BST problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array cost[][] in bottom up manner.

Dynamic Programming Solution

Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values. So how to fill the 2D array in such manner> The idea used in the implementation is same as Matrix Chain Multiplication problem, we use a variable L for chain length and increment L, one by one. We calculate column number j using the values of i and L.

```
// Dynamic Programming code for Optimal Binary Search Tree Problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates minimum cost of
   a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
    /* Create an auxiliary 2D matrix to store results of subproblems */
    int cost[n][n];

    /* cost[i][j] = Optimal cost of binary search tree that can be
       formed from keys[i] to keys[j].
```

```
       cost[0][n-1] will store the resultant cost */

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ... .
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[][]
        for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and chain length L
            int j = i+L-1;
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
               // c = cost when keys[r] becomes root of this subtree
               int c = ((r > i)? cost[i][r-1]:0) +
                       ((r < j)? cost[r+1][j]:0) +
                       sum(freq, i, j);
               if (c < cost[i][j])
                  cost[i][j] = c;
            }
        }
    }
    return cost[0][n-1];
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

```
Cost of Optimal BST is 142
```

Notes

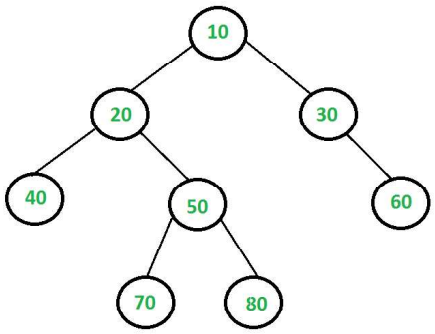
1) The time complexity of the above solution is O(n^4). The time complexity can be easily reduced to O(n^3) by pre-calculating sum of frequencies instead of calling sum() again and again.

2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen r in the innermost loop.

Aman Barnwal

Dynamic Programming | Set 26 (Largest Independent Set Problem)

Given a Binary Tree, find size of the **L**argest **I**ndependent **S**et(LIS) in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.
For example, consider the following binary tree. The largest independent set(LIS) is {10, 40, 60, 70, 80} and size of the LIS is 5.



A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

1) Optimal Substructure:
Let LISS(X) indicates size of largest independent set of a tree with root X.

```
LISS(X) = MAX { (1 + sum of LISS for all grandchildren of X),
                (sum of LISS for all children of X) }
```

The idea is simple, there are two possibilities for every node X, either X is a member of the set or not a member. If X is a member, then the value of LISS(X) is 1 plus LISS of all grandchildren. If X is not a member, then the value is sum of LISS of all children.

2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A naive recursive implementation of Largest Independent Set problem
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    struct node *left, *right;
};

// The function returns size of the largest independent set in a given
// binary tree
int LISS(struct node *root)
{
    if (root == NULL)
       return 0;

    // Caculate size excluding the current node
    int size_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int size_incl = 1;
    if (root->left)
       size_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
       size_incl += LISS(root->right->left) + LISS(root->right->right);

    // Return the maximum of two sizes
    return max(size_incl, size_excl);
}


// A utility function to create a node
struct node* newNode( int data )
```

Aman Bansal

```
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root         = newNode(20);
    root->left                = newNode(8);
    root->left->left          = newNode(4);
    root->left->right         = newNode(12);
    root->left->right->left   = newNode(10);
    root->left->right->right  = newNode(14);
    root->right               = newNode(22);
    root->right->right        = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}
```

Output:

```
Size of the Largest Independent Set is 5
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, LISS of node with value 50 is evaluated for node with values 10 and 20 as 50 is grandchild of 10 and child of 20. Since same suproblems are called again, this problem has Overlapping Subprolems property. So LISS problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field liss is added to tree nodes. The initial value of liss is set as 0 for all nodes. The recursive function LISS() calculates liss for a node only if it is not already set.

```
/* Dynamic programming based program for Largest Independent Set problem */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    int liss;
    struct node *left, *right;
};

// A memoization function returns size of the largest independent set in
//  a given binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    if (root->liss)
        return root->liss;

    if (root->left == NULL && root->right == NULL)
        return (root->liss = 1);

    // Calculate size excluding the current node
    int liss_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int liss_incl = 1;
    if (root->left)
        liss_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        liss_incl += LISS(root->right->left) + LISS(root->right->right);

    // Maximum of two sizes is LISS, store it for future uses.
    root->liss = max(liss_incl, liss_excl);
```

Aman Barnwal

```
    return root->liss;
}

// A utility function to create a node
struct node* newNode(int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->liss = 0;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root         = newNode(20);
    root->left                = newNode(8);
    root->left->left          = newNode(4);
    root->left->right         = newNode(12);
    root->left->right->left   = newNode(10);
    root->left->right->right  = newNode(14);
    root->right               = newNode(22);
    root->right->right        = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}
```

Output

```
Size of the Largest Independent Set is 5
```

Time Complexity: O(n) where n is the number of nodes in given Binary tree.

Following extensions to above solution can be tried as an exercise.
1) Extend the above solution for n-ary tree.

2) The above solution modifies the given tree structure by adding an additional field liss to tree nodes. Extend the solution so that it doesnt modify the tree structure.

3) The above solution only returns size of LIS, it doesnt print elements of LIS. Extend the solution to print all nodes that are part of LIS.

Aman Barnwal

Dynamic Programming | Set 25 (Subset Sum Problem)

Given a set of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.

```
Examples: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output:  True  //There is a subset (4, 5) with sum 9.
```

Let isSubSetSum(int set[], int n, int sum) be the function to find whether there is a subset of set[] with sum equal to *sum*. n is the number of elements in set[].

The isSubsetSum problem can be divided into two subproblems
a) Include the last element, recur for n = n-1, sum = sum set[n-1]
b) Exclude the last element, recur for n = n-1.
If any of the above the above subproblems return true, then return true.

Following is the recursive formula for isSubsetSum() problem.

```
isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) ||
                           isSubsetSum(arr, n-1, sum-set[n-1])
Base Cases:
isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0
isSubsetSum(set, n, sum) = true, if sum == 0
```

Following is naive recursive implementation that simply follows the recursive structure mentioned above.

```
// A recursive solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sun equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
   // Base Cases
   if (sum == 0)
     return true;
   if (n == 0 && sum != 0)
     return false;

   // If last element is greater than sum, then ignore it
   if (set[n-1] > sum)
     return isSubsetSum(set, n-1, sum);

   /* else, check if sum can be obtained by any of the following
      (a) including the last element
      (b) excluding the last element   */
   return isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1]);
}

// Driver program to test above function
int main()
{
  int set[] = {3, 34, 4, 12, 5, 2};
  int sum = 9;
  int n = sizeof(set)/sizeof(set[0]);
  if (isSubsetSum(set, n, sum) == true)
     printf("Found a subset with given sum");
  else
     printf("No subset with given sum");
  return 0;
}
```

Output:

```
 Found a subset with given sum
```

The above solution may try all subsets of given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).

We can solve the problem in Pseudo-polynomial time using Dynamic programming. We create a boolean 2D table subset[][] and fill it in bottom up manner. The value of subset[i][j] will be true if there is a subset of set[0..j-1] with sum equal to i., otherwise false. Finally, we return subset[sum][n]

```
// A Dynamic Programming solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sun equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
```

Aman Barnwal

```
    // The value of subset[i][j] will be true if there is a subset of set[0..j-1]
    //  with sum equal to i
    bool subset[sum+1][n+1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
      subset[0][i] = true;

    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++)
      subset[i][0] = false;

     // Fill the subset table in botton up manner
     for (int i = 1; i <= sum; i++)
     {
       for (int j = 1; j <= n; j++)
       {
         subset[i][j] = subset[i][j-1];
         if (i >= set[j-1])
           subset[i][j] = subset[i][j] || subset[i - set[j-1]][j-1];
       }
     }

    /* // uncomment this code to print table
     for (int i = 0; i <= sum; i++)
     {
       for (int j = 0; j <= n; j++)
          printf ("%4d", subset[i][j]);
       printf("\n");
     } */

     return subset[sum][n];
}

// Driver program to test above function
int main()
{
  int set[] = {3, 34, 4, 12, 5, 2};
  int sum = 9;
  int n = sizeof(set)/sizeof(set[0]);
  if (isSubsetSum(set, n, sum) == true)
     printf("Found a subset with given sum");
  else
     printf("No subset with given sum");
  return 0;
}
```

Output:

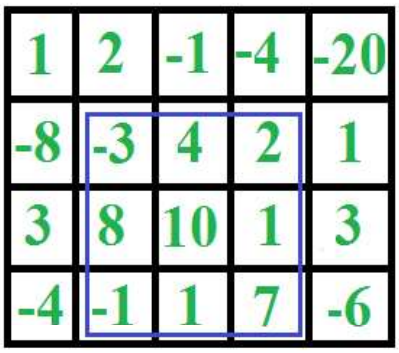
```
Found a subset with given sum
```

Time complexity of the above solution is O(sum*n).

Aman Barnwal

Dynamic Programming | Set 27 (Maximum sum rectangle in a 2D matrix)

Given a 2D array, find the maximum sum subarray in it. For example, in the following 2D array, the maximum sum subarray is highlighted with blue rectangle and sum of this subarray is 29.



This problem is mainly an extension of Largest Sum Contiguous Subarray for 1D array.

The **naive solution** for this problem is to check every possible rectangle in given 2D array. This solution requires 4 nested loops and time complexity of this solution would be O(n^4).

Kadanes algorithm for 1D array can be used to reduce the time complexity to O(n^3). The idea is to fix the left and right columns one by one and find the maximum sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have maximum sum) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sun of elements in every row from left to right and store these sums in an array say temp[]. So temp[i] indicates sum of elements from left to right in row i. If we apply Kadanes 1D algorithm on temp[], and get the maximum sum subarray of temp, this maximum sum would be the maximum possible sum with left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far.

```
// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
#define ROW 4
#define COL 5

// Implementation of Kadane's algorithm for 1D array. The function returns the
// maximum sum and stores starting and ending indexes of the maximum sum subarray
// at addresses pointed by start and finish pointers respectively.
int kadane(int* arr, int* start, int* finish, int n)
{
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i)
    {
        sum += arr[i];
        if (sum < 0)
        {
            sum = 0;
            local_start = i+1;
        }
        else if (sum > maxSum)
        {
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }

     // There is at-least one non-negative number
    if (*finish != -1)
        return maxSum;

    // Special Case: When all numbers in arr[] are negative
    maxSum = arr[0];
    *start = *finish = 0;
```

Aman Barnwal

```
    // Find the maximum element in array
    for (i = 1; i < n; i++)
    {
        if (arr[i] > maxSum)
        {
            maxSum = arr[i];
            *start = *finish = i;
        }
    }
    return maxSum;
}

// The main function that finds maximum sum rectangle in M[][]
void findMaxSum(int M[][COL])
{
    // Variables to store the final output
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left)
    {
        // Initialize all elements of temp as 0
        memset(temp, 0, sizeof(temp));

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right)
        {
            // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += M[i][right];

            // Find the maximum sum subarray in temp[]. The kadane() function
            // also sets values of start and finish.  So 'sum' is sum of
            // rectangle between (start, left) and (finish, right) which is the
            //  maximum sum with boundary columns strictly as left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then update
            // maxSum and other output values
            if (sum > maxSum)
            {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }

    // Print final values
    printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
    printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
    printf("Max sum is: %d\n", maxSum);
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{1, 2, -1, -4, -20},
                       {-8, -3, 4, 2, 1},
                       {3, 8, 10, 1, 3},
                       {-4, -1, 1, 7, -6}
                      };

    findMaxSum(M);

    return 0;
}
```

Output:

```
(Top, Left) (1, 1)
(Bottom, Right) (3, 3)
Max sum is: 29
```

Time Complexity: O(n^3)

Aman Barnwal

Aman Barnwal

Count number of binary strings without consecutive 1s

Given a positive integer N, count all possible distinct binary strings of length N such that there are no consecutive 1s.

Examples:

```
Input:  N = 2
Output: 3
// The 3 strings are 00, 01, 10

Input: N = 3
Output: 5
// The 5 strings are 000, 001, 010, 100, 101
```

This problem can be solved using Dynamic Programming. Let a[i] be the number of binary strings of length i which do not contain any two consecutive 1s and which end in 0. Similarly, let b[i] be the number of such strings which end in 1. We can append either 0 or 1 to a string ending in 0, but we can only append 0 to a string ending in 1. This yields the recurrence relation:

```
a[i] = a[i - 1] + b[i - 1]
b[i] = a[i - 1]
```

The base cases of above recurrence are a[1] = b[1] = 1. The total number of strings of length i is just a[i] + b[i].

Following is C++ implementation of above solution. In the following implementation, indexes start from 0. So a[i] represents the number of binary strings for input length i+1. Similarly, b[i] represents binary strings for input length i+1.

```
// C++ program to count all distinct binary strings
// without two consecutive 1's
#include <iostream>
using namespace std;

int countStrings(int n)
{
    int a[n], b[n];
    a[0] = b[0] = 1;
    for (int i = 1; i < n; i++)
    {
        a[i] = a[i-1] + b[i-1];
        b[i] = a[i-1];
    }
    return a[n-1] + b[n-1];
}


// Driver program to test above functions
int main()
{
    cout << countStrings(3) << endl;
    return 0;
}
```

Output:

```
5
```

Source:
courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf

Aman Barnwal

Dynamic Programming | Set 37 (Boolean Parenthesization Problem)

Given a boolean expression with following symbols.

```
Symbols
    'T' ---> true
    'F' ---> false
```

And following operators filled between symbols

```
Operators
    &   ---> boolean AND
    |   ---> boolean OR
    ^   ---> boolean XOR
```

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Let the input be in form of two arrays one contains the symbols (T and F) in order and other contains operators (&, | and ^}

Examples:

```
Input: symbol[]    = {T, F, T}
       operator[]  = {^, &}
Output: 2
The given expression is "T ^ F & T", it evaluates true
in two ways "((T ^ F) & T)" and "(T ^ (F & T))"

Input: symbol[]    = {T, F, F}
       operator[]  = {^, |}
Output: 2
The given expression is "T ^ F | F", it evaluates true
in two ways "( (T ^ F) | F )" and "( T ^ (F | F) )".

Input: symbol[]    = {T, T, F, T}
       operator[]  = {|, &, ^}
Output: 4
The given expression is "T | T & F ^ T", it evaluates true
in 4 ways ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T)
and (T|((T&F)^T)).
```

Solution:
Let **T(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to true.

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ Total(i,k) * Total(k+1,j) - F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * F(k+1,j) + F(i,k) * T(k+1) & \text{If operator[k] is '\^'} \end{cases}$$

$$\text{Total(i, j)} = \text{T(i, j)} + \text{F(i, j)}$$

Let **F(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to false.

$$F(i,j) = \sum_{k=i}^{j-1} \begin{cases} Total(i,k) * Total(k+1,j) - T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * T(k+1,j) + F(i,k) * F(k+1) & \text{If operator[k] is '\^'} \end{cases}$$

$$\text{Total(i, j)} = \text{T(i, j)} + \text{F(i, j)}$$

Base Cases:

```
T(i, i) = 1 if symbol[i] = 'T'
T(i, i) = 0 if symbol[i] = 'F'

F(i, i) = 1 if symbol[i] = 'F'
F(i, i) = 0 if symbol[i] = 'T'
```

If we draw recursion tree of above recursive solution, we can observe that it many overlapping subproblems. Like other dynamic programming problems, it can be solved by filling a table in bottom up manner. Following is C++ implementation of dynamic programming solution.

```
#include<iostream>
#include<cstring>
using namespace std;

// Returns count of all possible parenthesizations that lead to
// result true for a boolean expression with symbols like true
// and false and operators like &, | and ^ filled between symbols
```

Aman Barnwal

```
int countParenth(char symb[], char oper[], int n)
{
    int F[n][n], T[n][n];

    // Fill diaginal entries first
    // All diagonal entries in T[i][i] are 1 if symbol[i]
    // is T (true).  Similarly, all F[i][i] entries are 1 if
    // symbol[i] is F (False)
    for (int i = 0; i < n; i++)
    {
        F[i][i] = (symb[i] == 'F')? 1: 0;
        T[i][i] = (symb[i] == 'T')? 1: 0;
    }

    // Now fill T[i][i+1], T[i][i+2], T[i][i+3]... in order
    // And F[i][i+1], F[i][i+2], F[i][i+3]... in order
    for (int gap=1; gap<n; ++gap)
    {
        for (int i=0, j=gap; j<n; ++i, ++j)
        {
            T[i][j] = F[i][j] = 0;
            for (int g=0; g<gap; g++)
            {
                // Find place of parenthesization using current value
                // of gap
                int k = i + g;

                // Store Total[i][k] and Total[k+1][j]
                int tik = T[i][k] + F[i][k];
                int tkj = T[k+1][j] + F[k+1][j];

                // Follow the recursive formulas according to the current
                // operator
                if (oper[k] == '&')
                {
                    T[i][j] += T[i][k]*T[k+1][j];
                    F[i][j] += (tik*tkj - T[i][k]*T[k+1][j]);
                }
                if (oper[k] == '|')
                {
                    F[i][j] += F[i][k]*F[k+1][j];
                    T[i][j] += (tik*tkj - F[i][k]*F[k+1][j]);
                }
                if (oper[k] == '^')
                {
                    T[i][j] += F[i][k]*T[k+1][j] + T[i][k]*F[k+1][j];
                    F[i][j] += T[i][k]*T[k+1][j] + F[i][k]*F[k+1][j];
                }
            }
        }
    }
    return T[0][n-1];
}

// Driver program to test above function
int main()
{
    char symbols[] = "TTFT";
    char operators[] = "|&^";
    int n = strlen(symbols);

    // There are 4 ways
    // ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T))
    cout << countParenth(symbols, operators, n);
    return 0;
}
```

Output:

```
4
```

Time Complexity: $O(n^3)$
Auxiliary Space: $O(n^2)$

References:
http://people.cs.clemson.edu/~bcdean/dp_practice/dp_9.swf

Aman Barnwal