Dynamic programming notes

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

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 don't 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 doesn't have common subproblems

Recursion tree for execution of *fib(5)*

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

```
    return fib(n-1) + fib(n-2);
}
```

```
                                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)
```

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.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

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.

```
/* 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);
   }

   return lookup[n];
}
```

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.

```
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];
}
```

Longest common subsequence

A sequence is a group of letters that are part of the word in the same direction. Longest common subsequence is a group of letters that is common between two words in the same direction.

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.

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”, “AEDFH**R**”), L(“ABCDG**H**”, “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

```
/* 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));
}
```

The idea is to start either at the end or at the beginning:

Starting at the beginning:

1. If the first letter of both the words is the same, then this letter will form part of a long subsequence hence the count is incremented and this is continued until a letter that is not common is found.
2. When a letter that is not common is found, then we will have two choices:
    a. Move to the next letter in the first word and go to step 1.
    b. Move to the next letter in the second word and go to step 1.
    c. Compare the LCS length of both a and b and select the max.

The count value is incremented when letters from either words are found to be same and left to zero if it is not found to be same.

Dynamic programming kicks in and triggers a case of memosiation where the count value of letters of both words are stored and hence need not be calculated again and again as required in variants of step 2.

```
/* 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]; // 0 to m is m+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) // the first row and the first column are all
zeroes
         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]);  //compare with up element and
the left element in the matrix
     }
   }

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

Arrays X= {aba} and Y ={axcd}

```
L[3+1][4+1] = L[4][5] =  0  0  0  0  0
                         0  1  1  1  1
                         0  1  1  1  1
                         0  1  1  1  1
```

Disecting the above code:
L[i][j] = 2 where i = 4 means with the first 4 letters of the first word, the longest common subsequence with the second word is of length 2.

See the diagonal of L[m][n]

Longest increasing subsequence

The objective 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}

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.
How does the recusive approach look like:

```
/* 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;

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

    /* Recursively get all LIS ending with arr[0], arr[1] ... ar[n-2]. If
       arr[i-1] is smaller than arr[n-1], and max ending with arr[n-1] needs
       to be updated, then update it */
    for(int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref); //We find the max_ending for i =1,2,3..n
        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;
}
```

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

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation.

If we take the recursive approach then we may end up with a complexity of O(2^n) as there can be 2^n subsequences in the given sequence. Hence it is pertinent to use dynamic programming and reduce the complexity.

```
/* 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)// have a case where a[j]
is greatest but lis[j] is only 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;
}
```

The above gives a complexity of O(n^2).

How does the above work:

Can we reduce the complexity. Yes it is possible to reduce the complexity to O(n log n). Using the approach below:

Let `S[pos]` be defined as the smallest integer that ends an increasing sequence of length `pos`.
Now iterate through every integer `X` of the input set and do the following:

1. If `X` > last element in `S`, then append `X` to the end of `S`. This essentialy means we have found a new largest `LIS`.
2. Otherwise find the smallest element in `S`, which is `>=` than `X`, and change it to `X`. Because `S`is sorted at any time, the element can be found using binary search in `log(N)`.

Total runtime - `N` integers and a binary search for each of them - N * log(N) = O(N log N)
Now let's do a real example:

Set of integers: `2 6 3 4 1 2 9 5 8`
Steps:

```
0. S = {} - Initialize S to the empty set
1. S = {2} - New largest LIS
2. S = {2, 6} - New largest LIS
3. S = {2, 3} - Changed 6 to 3
4. S = {2, 3, 4} - New largest LIS
5. S = {1, 3, 4} - Changed 2 to 1
6. S = {1, 2, 4} - Changed 3 to 2
7. S = {1, 2, 4, 9} - New largest LIS
8. S = {1, 2, 4, 5} - Changed 9 to 5
9. S = {1, 2, 4, 5, 8} - New largest LIS
```

So the length of the LIS is `5` (the size of S).
To reconstruct the actual `LIS` we will again use a parent array. Let `parent[i]` be the predecessor of element with index `i` in the `LIS` ending at element with index `i`.
To make things simpler, we can keep in the array `S`, not the actual integers, but their indices(positions) in the set. We do not keep `{1, 2, 4, 5, 8}`, but keep `{4, 5, 3, 7, 8}`.
That is input[4] = **1**, input[5] = **2**, input[3] = **4**, input[7] = **5**, input[8] = **8**.
If we update properly the parent array, the actual LIS is:

```
input[S[lastElementOfS]],
input[parent[S[lastElementOfS]]],
input[parent[parent[S[lastElementOfS]]]],
.....................................
```

Now to the important thing - how do we update the parent array? There are two options:

1. If `X` > last element in `S`, then `parent[indexX] = indexLastElement`. This means the parent of the newest element is the last element. We just prepend `X` to the end of `S`.
2. Otherwise find the index of the smallest element in `S`, which is `>=` than `X`, and change it to `X`. Here `parent[indexX] = S[index - 1]`.

Code below:

```
// Binary search
int GetCeilIndex(int A[], int T[], int l, int r, int key) {
   int m;

   while( r - l > 1 ) {
      m = l + (r - l)/2;
      if( A[T[m]] >= key )
         r = m;
      else
         l = m;
   }

   return r;
}

int LongestIncreasingSubsequence(int A[], int size) {
   // Add boundary case, when array size is zero
   // Depend on smart pointers

   int *tailIndices = new int[size];
   int *prevIndices = new int[size];
   int len;

   memset(tailIndices, 0, sizeof(tailIndices[0])*size);
   memset(prevIndices, 0xFF, sizeof(prevIndices[0])*size);

   tailIndices[0] = 0;
   prevIndices[0] = -1;
   len = 1; // it will always point to empty location
   for( int i = 1; i < size; i++ ) {
      if( A[i] < A[tailIndices[0]] ) {
         // new smallest value
         tailIndices[0] = i;
      } else if( A[i] > A[tailIndices[len-1]] ) {
         // A[i] wants to extend largest subsequence
         prevIndices[i] = tailIndices[len-1];
         tailIndices[len++] = i;
      } else {
         // A[i] wants to be a potential condidate of future subsequence
         // It will replace ceil value in tailIndices
        int pos = GetCeilIndex(A, tailIndices, -1, len-1, A[i]);

        prevIndices[i] = tailIndices[pos-1];
        tailIndices[pos] = i;
      }
   }
   cout << "LIS of given input" << endl;
   for( int i = tailIndices[len-1]; i >= 0; i = prevIndices[i] )
      cout << A[i] << "   ";
   cout << endl;

   delete[] tailIndices;
   delete[] prevIndices;

   return len;
}
```

Comment:

Understanding the O(n log n) approach took lot of effort and time, Realizing that the parent/prevIndex stores the index for a larger element/or (pos-1) for a element replaced by binary search was quite a revelation.

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)
```

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.

```
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;
}
```

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

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

This is similar to the longest increasing subsequence wherein instead of computing the length of the sequence at every array index, we compute the sum and retrieve the index which corresponds to the max sum.

```
/* maxSumIS() returns the maximum sum of increasing subsequence in arr[] of
   size n */
int maxSumIS( int arr[], int n )
{
```

```
    int *msis, i, j, max = 0;
    msis = (int*) malloc ( sizeof( int ) * 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];

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

    return max;
}
```

Longest Palindrome 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.

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
```

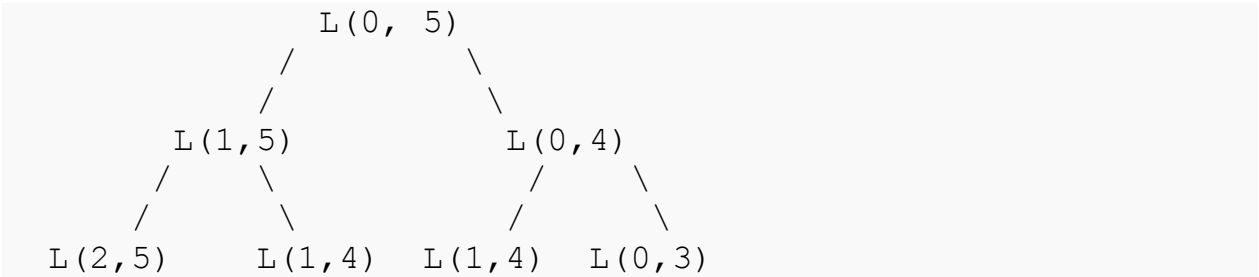
```
Else if (j == i + 1) L(i, j) = 2

// 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.

```
// 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) );
}
```

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



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

```
// 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];
}
```

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.

Longest Palindrome Substring

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)

Dynamic programming solution, $O(N^2)$ time and $O(N^2)$ space:
To improve over the brute force solution from a DP approach, first think how we can avoid unnecessary re-computation in validating palindromes. Consider the case "ababa". If we already knew that "bab" is a palindrome, it is obvious that "ababa" must be a palindrome since the two left and right end letters are the same.

Stated more formally below:

Define P[i, j] ← true **iff** the substring S_i … S_j is a palindrome, otherwise false.

Therefore,

P[i, j] ← (P[i+1, j-1] **and** $S_i = S_j$)

The base cases are:

P[i, i] ← true
P[i, i+1] ← ($S_i = S_{i+1}$)

This yields a straight forward DP solution, which we first initialize the one and two letters palindromes, and work our way up finding all three letters palindromes, and so on…

```
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

//if n = 6, then i = 0 to 4 and j for every i is (k-1)th value away from i
and when k is finally n, i = 0 and j = n-1
            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;

                if( k > maxLength )
                {
                    start = i;
                    maxLength = k;
                }
            }
        }
    }

    printf("Longest palindrome substring is: ");
    printSubStr( str, start, start + maxLength - 1 );

    return maxLength; // return length of LPS
}
```

Example: habbak

Table =
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 1 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1

In the loop k =3
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 1 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1

In the loop k =4
1 0 0 0 0 0
0 1 0 0 1 0
0 0 1 1 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1

In the loop k =5
1 0 0 0 0 0
0 1 0 0 1 0
0 0 1 1 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1

In the loop k =6
1 0 0 0 0 0
0 1 0 0 1 0
0 0 1 1 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1

So finally I = 1 and (1,4) is (i,j) for length k = 4. (1,4) is 1 because (2,3) is 1 and (2,3) = (b,b)

Example 2:

For string “aba”

Initial

1 0 0
0 1 0
0 0 1

Loop k =3

1 0 1
0 1 0
0 0 1

Here I =0, (I,j) = (0,2) and (0,2) = (a,a)

Length of the longest substring without repeating characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substrings without repeating characters for “ABDEFGABEF” are “BDEFGA” and “DEFGAB”, with length 6. For “BBBB” the longest substring is “B”, with length 1. For “GEEKSFORGEEKS”, there are two longest substrings shown in the below diagrams, with length 7.

The desired time complexity is O(n) where n is the length of the string.

Let us talk about the linear time solution now. This solution uses extra space to store the last indexes of already visited characters. The idea is to scan the string from left to right, keep track of the maximum length Non-Repeating Character Substring (NRCS) seen so far. Let the maximum length be max_len. When we traverse the string, we also keep track of length of the current NRCS using cur_len variable. For every new character, we look for it in already processed part of the string (A temp array called visited[] is used for this purpose). If it is not present, then we increase the cur_len by 1. If present, then there are two cases:

a) The previous instance of character is not part of current NRCS (The NRCS which is under process). In this case, we need to simply increase cur_len by 1.
b) If the previous instance is part of the current NRCS, then our current NRCS changes. It becomes the substring staring from the next character of previous instance to currently scanned character. We also need to compare cur_len and max_len, before changing current NRCS (or changing cur_len).

```
int longestUniqueSubsttr(char *str)
{
    int n = strlen(str);
    int cur_len = 1;  // To store the lenght of current substring
    int max_len = 1;  // To store the result
    int prev_index;  // To store the previous index
    int i;
    int *visited = (int *)malloc(sizeof(int)*NO_OF_CHARS);

    /* Initialize the visited array as -1, -1 is used to indicate that
       character has not been visited yet. */
    for (i = 0; i < NO_OF_CHARS;  i++)
        visited[i] = -1;

    /* Mark first character as visited by storing the index of first
       character in visited array. */
    visited[str[0]] = 0;

    /* Start from the second character. First character is already processed
```

```
        (cur_len and max_len are initialized as 1, and visited[str[0]] is set
*/
    for (i = 1; i < n; i++)
    {
        prev_index =  visited[str[i]];

        /* If the currentt character is not present in the already processed
           substring or it is not part of the current NRCS, then do cur_len++
*/
        if (prev_index == -1 || i - cur_len > prev_index)
            cur_len++;

        /* If the current character is present in currently considered NRCS,
           then update NRCS to start from the next character of previous
instance. */
        else
        {
          //Also, when we are changing the NRCS, we should also check whether
            //length of the previous NRCS was greater than max_len or not.*/
            if (cur_len > max_len)
                max_len = cur_len;

 //cur_len is updated here  as the new string starts from the prev_index
            cur_len = i - prev_index;         }

        visited[str[i]] = i; // update the index of current character
    }

    // Compare the length of last NRCS with max_len and update max_len if
needed
    if (cur_len > max_len)
        max_len = cur_len;


    free(visited); // free memory allocated for visited

    return max_len;
}
```

Time Complexity: O(n + d) where n is length of the input string and d is number of characters in input string alphabet. For example, if string consists of lowercase English characters then value of d is 26.
Auxiliary Space: O(d)
Algorithmic Paradigm: Dynamic Programming

As an exercise, try the modified version of the above problem where you need to print the maximum length NRCS also (the above program only prints length of it).

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, “abalblbbabblalblaba” 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 “albabbbablblababa”. 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

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
```

```
// 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)
                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];
}
```

```
Answer for abcd
i = 0 j =  1 L = 2 C[k+1][j] 0
i = 1 j =  2 L = 2 C[k+1][j] 0
i = 2 j =  3 L = 2 C[k+1][j] 0

i = 0 j =  2 L = 3 C[k+1][j] 1
i = 0 j =  2 L = 3 C[k+1][j] 0
i = 1 j =  3 L = 3 C[k+1][j] 1
i = 1 j =  3 L = 3 C[k+1][j] 0
```

```
i = 0 j =  3 L = 4 C[k+1][j] 2
i = 0 j =  3 L = 4 C[k+1][j] 1
i = 0 j =  3 L = 4 C[k+1][j] 0
Min cuts needed for Palindrome Partitioning is 3
```

Complexity is O(n^3)

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.

Naïve recursive implementation

Following is the recursive formula for isSubsetSum() problem.

```
// 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]);
}
```

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)

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]

```
// Returns true if there is a subset of set[] with sun equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // 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];
}
```

```
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];
```

Set[]= {1,2,3,4,5} , Sum =9

Sum =9, I =0 to 10, j = 0 to 6

	0	1	2	3	4	5
0	T	T	T	T	T	T
1	T	T	T	T	T	T
2	F	T	T	T	T	T
3	F	T	T	T	T	T
4	F	F	T	T	T	T
5	F	F	F	F	T	T
6	F	F	F	T	T	T
7	F					
8	F					
9	F					

Time complexity of the above solution is O(sum*n).

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

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.

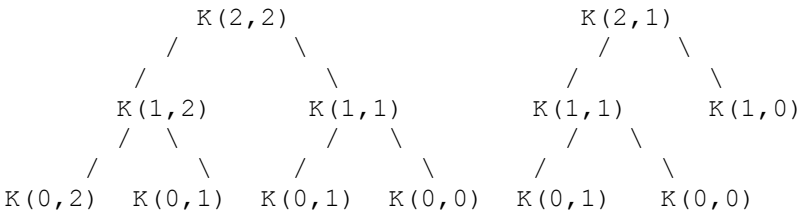
2) Overlapping Subproblems

Recursive approach

```
// 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)
                  );
}
```

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)
                    /            \
                  /                \
```



Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Following is Dynamic Programming based implementation. It constructs a temporary array K[][] in bottom up manner.

```
// 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];  //if weight is greater then exclude
       }
   }

   return K[n][W];
}
```

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

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 doesn't 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.

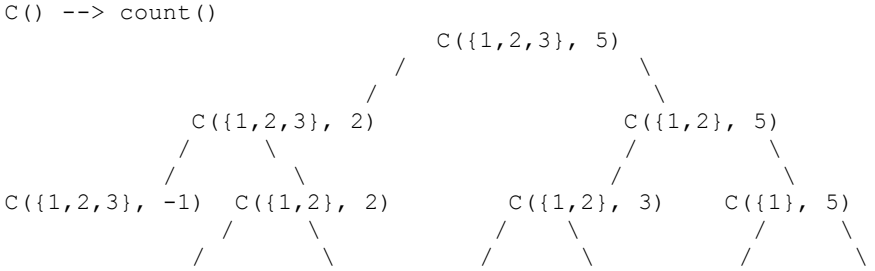
1) Optimal Substructure
To count total number solutions, we can divide all set solutions in two sets.
1) Solutions that do not contain mth coin (or Sm).
2) Solutions that contain at least one Sm.
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[],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

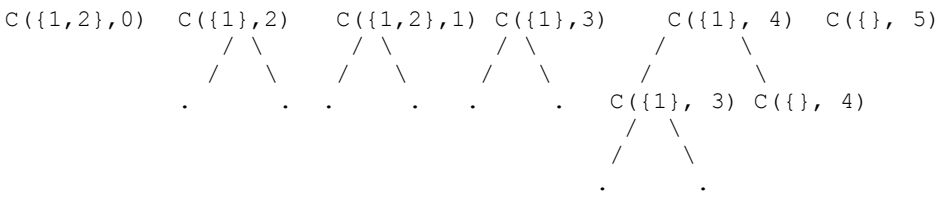
Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

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

```
// 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] );
}
```

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}, 1) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.





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

```
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)

    //Here n is the value and m is the number of coins
    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];
}
```

Time Complexity: O(mn)

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

```
// 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 )
{
    // 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++) //( m = number of coins)
        for(int j=S[i]; j<=n; j++) //(n = amount)
            table[j]= table[j] + table[j-S[i]];

    return table[n];
}
```

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.

```
i = 0,  s[i] = 1, j = 1, table[j] = 1
i = 0,  s[i] = 1, j = 2, table[j] = 1
i = 0,  s[i] = 1, j = 3, table[j] = 1
i = 0,  s[i] = 1, j = 4, table[j] = 1
table[j] = {1,1,1,1,1}

i = 1,  s[i] = 2, j = 2, table[j] = 2
i = 1,  s[i] = 2, j = 3, table[j] = 2
i = 1,  s[i] = 2, j = 4, table[j] = 3
table[j] = {1,1,2,2,3}

i = 2,  s[i] = 3, j = 3, table[j] = 3
i = 2,  s[i] = 3, j = 4, table[j] = 4
table[j] = {1,1,2,3,4}
```

Matrix Chain Multiplication

Given a sequence of matrices, find the most efficient way to multiply these matrices together.

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 = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500
operations
    A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000
operations.
```

Clearly the first method is the more efficient.

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

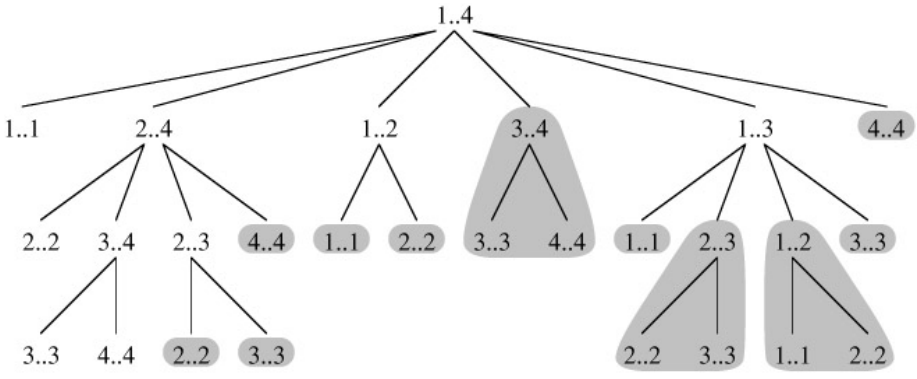
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then 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)

Following is a recursive implementation that simply follows the above optimal substructure property.

```
// 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)
            min = count;
    }

    // Return minimum count
    return min;
}
```

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.



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

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

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

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

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] and 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.

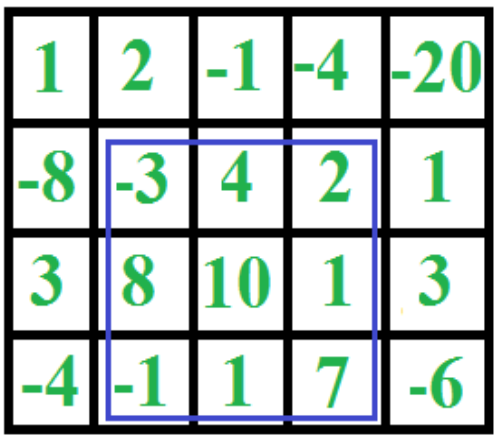
```
void printMaxSubSquare(bool M[R][C])
{
```

```
  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;
    }
  }

  /* 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");
  }
}
```

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.

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.



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).

Kadane's 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 Kadane's 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.

```
// The main function that finds maximum sum rectangle in M[][]
void findMaxSum(int M[][COL])
{
    // Variables to store the final output
    int maxSum = 0, finalLeft, finalRigh
t, 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);
}
```

Time Complexity: O(n^3)

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.

Naïve recursive function

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(price, n-i-1)) for all i in {0, 1 .. n-1}

The implementation simply follows the recursive structure mentioned above.

```
/* 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;
}
```

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

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

                                cR(4)
                      /         /       \      \
                     /         /         \       \
                 cR(3)       cR(2)      cR(1)    cR(0)
               /   |   \       /   \        |
```

```
            /    |    \      /     \       |
      cR(2) cR(1) cR(0) cR(1) cR(0) cR(0)
     / \          |           |
    /   \         |           |
 cR(1) cR(0) cR(0)         cR(0)
```

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 andthis) 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.

```
/* 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];
}
```

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

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

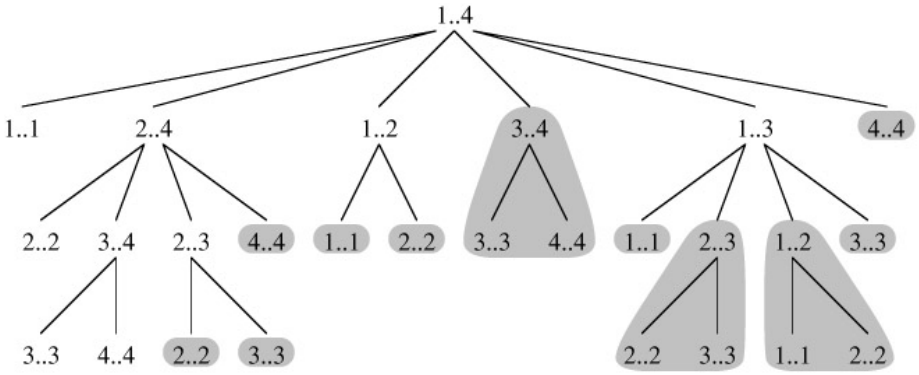
```
Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
There can be following possible BSTs
    10                12                 20          10
20
      \             /    \              /             \
/
      12          10     20           12                20
10
        \                            /                 /
\
         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
```

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// 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
    return min + fsum;
}
```

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].



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'.

```
/* 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];
}
```

Maximum length of 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.

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.

```
// 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;
}
```

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.

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.

Edit distance

Given two strings of size m, n and set of operations replace (R), insert (I) and delete (D) all at equal cost. Find minimum number of edits (operations) required to convert one string into another.

Box Stacking Problem

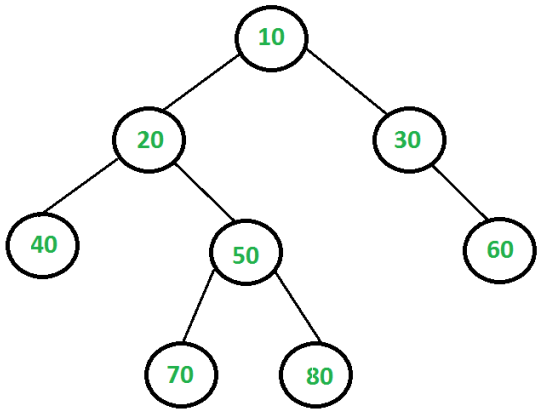
You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can

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.

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, 2×3 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.

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.



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.

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// 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);
}
```

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.

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.

```
// 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);

    // Caculate 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);

    // Return the maximum of two sizes
    root->liss = max(liss_incl, liss_excl);

    return root->liss;
}
```

Time Complexity: O(n) where n is the number of nodes in given Binary tree.

Bellman-Ford Algorithm

Floyd Warshall Algorithm

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.

When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesn't 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 doesn't 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}}
```

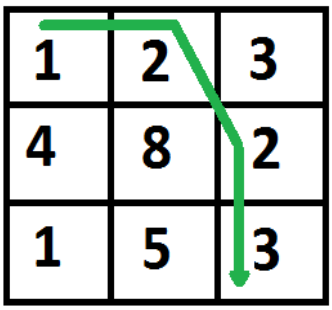
Minimum Cost Path

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).



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]

So the recursive implementation is as follows

```
/* 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) );
}
```

e. Time complexity of this naive recursive solution is exponential and it is terribly slow.

```
mC refers to minCost()
                                  mC(2, 2)
                          /           |           \
                         /            |            \
                mC(1, 1)            mC(1, 2)             mC(2, 1)
             /     |     \       /     |     \           /     |     \
            /      |      \     /      |      \         /      |      \
       mC(0,0) mC(0,1) mC(1,0) mC(0,1) mC(0,2) mC(1,1) mC(1,0) mC(1,1) mC(2,0)
```

Recomputations of same subproblems can be avoided by constructing a temporary array tc[][] in bottom up manner.

```
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];
}
```

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

Minimum number of jumps to reach end

Given an array all of whose elements are positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7).

More examples:

- [**3**, 2, **7**, 1]: return 10
- [**6**, 2, 1, **4**]: return 10
- [**8**, 9, **5**, 1]: return 13
- [29, **77**, 16]: return 77
- [**29**, 44, **16**]: return 45

This problem can be solved by dynamic programming.

Let's say we have an array of integers:

```
i[1], i[2], i[3], ..., i[n], i[n+1], i[n+2]
```

We partition the array into two parts: the first part containing first n integers, and the second part is the last two integers:

```
{i[1], i[2], i[3], ..., i[n]}, {i[n+1], i[n+2]}
```

Let's denote `M_SUM(n)` as the max sum of the first n integers per your requirement.
There will be two cases:

1. if `i[n]` is not counted into `M_SUM(n)`, then `M_SUM(n+2) = M_SUM(n) + MAX(i[n+1], i[n+2])`
2. if `i[n]` is counted into `M_SUM(n)`, then `M_SUM(n+2) = M_SUM(n) + i[n+2]`

then, `M_SUM(n+2)`, the value we are seeking for, will be the larger value of the above two.
Then we can have a very naive pseudocode as below:

```
function M_SUM(n)
   return MAX(M_SUM(n, true), M_SUM(n, false))

function M_SUM(n, flag)
   if n == 0 then return 0
   else if n == 1
      return flag ? i[0] : 0
   } else {
      if flag then
         return MAX(
                M_SUM(n-2, true) + i[n-1],
                M_SUM(n-2, false) + MAX(i[n-1],i[n-2]))
      else
         return MAX(M_SUM(n-2, false) + i[n-2], M_SUM(n-2, true))
```

```
    }
```
"flag" means "allow using the last integer"

This algorithm has an exponential time complexity.

Dynamical programming techniques can be employed to eliminate the unnecessary recomputation of M_SUM.

Storing each `M_SUM(n, flag)` into a n*2 matrix. In the recursion part, if such a value does not present in the matrix, compute it. Otherwise, just fetch the value from the matrix. This will reduce the time complexity into linear.
The algorithm will have O(n) time complexity, and O(n) space complexity.

Construction of Longest Monotonically Increasing Subsequence (NlogN)

```
int GetCeilIndex(int A[], int T[], int l, int r, int key) {
   int m;

   while( r - l > 1 ) {
      m = l + (r - l)/2;
      if( A[T[m]] >= key )
         r = m;
      else
         l = m;
   }

   return r;
}

int LongestIncreasingSubsequence(int A[], int size) {
   // Add boundary case, when array size is zero
   // Depend on smart pointers

   int *tailIndices = new int[size];
   int *prevIndices = new int[size];
   int len;

   memset(tailIndices, 0, sizeof(tailIndices[0])*size);
   memset(prevIndices, 0xFF, sizeof(prevIndices[0])*size);

   tailIndices[0] = 0;
   prevIndices[0] = -1;
   len = 1; // it will always point to empty location
   for( int i = 1; i < size; i++ ) {
      if( A[i] < A[tailIndices[0]] ) {
         // new smallest value
         tailIndices[0] = i;
      } else if( A[i] > A[tailIndices[len-1]] ) {
         // A[i] wants to extend largest subsequence
```

```
            prevIndices[i] = tailIndices[len-1];
            tailIndices[len++] = i;
        } else {
            // A[i] wants to be a potential condidate of future subsequence
            // It will replace ceil value in tailIndices
           int pos = GetCeilIndex(A, tailIndices, -1, len-1, A[i]);

           prevIndices[i] = tailIndices[pos-1];
           tailIndices[pos] = i;
        }
    }
    cout << "LIS of given input" << endl;
    for( int i = tailIndices[len-1]; i >= 0; i = prevIndices[i] )
       cout << A[i] << "   ";
    cout << endl;

    delete[] tailIndices;
    delete[] prevIndices;

    return len;
}

int main() {
    int A[] = { 2, 5, 3, 7, 11, 8, 10, 13, 6 };
    int size = sizeof(A)/sizeof(A[0]);

    printf("LIS size %d\n", LongestIncreasingSubsequence(A, size));

    return 0;
}
```

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible.

http://stackoverflow.com/questions/4212050/minimum-difference-between-sum-of-two-subsets

If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.

For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).
Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.

The following solution tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining

elements (other subset). We consider both possibilities for every element. When the size of current set becomes n/2, we check whether this solutions is better than the best solution available so far. If it is, then we update the best solution.

```
// function that tries every possible solution by calling itself recursively
void TOWUtil(int* arr, int n, bool* curr_elements, int
no_of_selected_elements,
             bool* soln, int* min_diff, int sum, int curr_sum, int
curr_position)
{
    // checks whether the it is going out of bound
    if (curr_position == n)
        return;

    // checks that the numbers of elements left are not less than the
    // number of elements required to form the solution
    if ((n/2 - no_of_selected_elements) > (n - curr_position))
        return;

    // consider the cases when current element is not included in the
solution
    TOWUtil(arr, n, curr_elements, no_of_selected_elements,
              soln, min_diff, sum, curr_sum, curr_position+1);

    // add the current element to the solution
    no_of_selected_elements++;
    curr_sum = curr_sum + arr[curr_position];
    curr_elements[curr_position] = true;

    // checks if a solution is formed
    if (no_of_selected_elements == n/2)
    {
        // checks if the solution formed is better than the best solution so
far
        if (abs(sum/2 - curr_sum) < *min_diff)
        {
            *min_diff = abs(sum/2 - curr_sum);
            for (int i = 0; i<n; i++)
                soln[i] = curr_elements[i];
        }
    }
    else
    {
        // consider the cases where current element is included in the
solution
        TOWUtil(arr, n, curr_elements, no_of_selected_elements, soln,
                  min_diff, sum, curr_sum, curr_position+1);
    }

    // removes current element before returning to the caller of this
function
    curr_elements[curr_position] = false;
}

// main function that generate an arr
```

```
void tugOfWar(int *arr, int n)
{
    // the boolen array that contains the inclusion and exclusion of an
element
    // in current set. The number excluded automatically form the other set
    bool* curr_elements = new bool[n];

    // The inclusion/exclusion array for final solution
    bool* soln = new bool[n];

    int min_diff = INT_MAX;

    int sum = 0;
    for (int i=0; i<n; i++)
    {
        sum += arr[i];
        curr_elements[i] =  soln[i] = false;
    }

    // Find the solution using recursive function TOWUtil()
    TOWUtil(arr, n, curr_elements, 0, soln, &min_diff, sum, 0, 0);

    // Print the solution
    cout << "The first subset is: ";
    for (int i=0; i<n; i++)
    {
        if (soln[i] == true)
            cout << arr[i] << " ";
    }
    cout << "\nThe second subset is: ";
    for (int i=0; i<n; i++)
    {
        if (soln[i] == false)
            cout << arr[i] << " ";
    }
}
```

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.

You are given many slabs each with a length and a breadth. A slab i can be put on slab j if both dimensions of i are less than that of j. In this similar manner, you can keep on putting slabs on each other. Find the maximum stack possible which you can create out of the given slabs.

Here is my solution:--
1) first rearrange the dimensions of slabs i.e. put bigger dimension in y and smaller dimenson in x (rotate the slab)
2) then arrange all slabs in increasing order of x dimension
3) and then find the longest increasing sub-sequence based on y dimension.....

Ex:- (2,5),(5,1),(1,3),(1,2),(6,1)

Step-1=> (2,5),(1,5),(1,3),(1,2),(1,6)

Step-2=> (1,2),(1,3),(1,5),(1,6),(2,5)

Step-3=> LIS=4 { (1,2),(1,3),(1,5),(1,6) OR (1,2),(1,3),(1,5),(2,5) }

You are given pairs of numbers. In a pair the first number is smaller with respect to the second number. Suppose you have two sets (a, b) and (c, d), the second set can follow the first set if b<c. So you can form a long chain in the similar fashion. Find the longest chain which can be formed.

Let the pair be (xi,yi) as its given xi<yi, instead of soring wrt xi, soring wrt yi will be the ideal and the final sequnce should be x1<y1<x2<y2...xn<yn where (xi yi) is a pair So algo could be
1. Sort pairs wrt yi in ascending oeder.
2. add first pair (x1,y1) to sequence.
3. Start with i=1, and yi<x(i+1) then add x(i+1) y(i+1) to sequnce and increment i;
4. repeat step 3 till i=1;
in simple, we just eliminating the pairs that doesn't fit into the sequence...

Find the maximum contiguous subsequence in a bar chart.

I was able to find a solution in O(N^3) which is basically the brute force method The idea is that given a bar chart with peaks and valley's, you should find the subsequence X which gives you the maximum area underneath the chart. For the life of me I couldn't think of the answer at the time, , I had a hard time visualizing it, however it's a classical computer science problem called "the maximum sum contiguous subsequence problem

http://www.careercup.com/page?pid=dynamic-programming-interview-questions

Partition problem

http://en.wikipedia.org/wiki/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.
```
