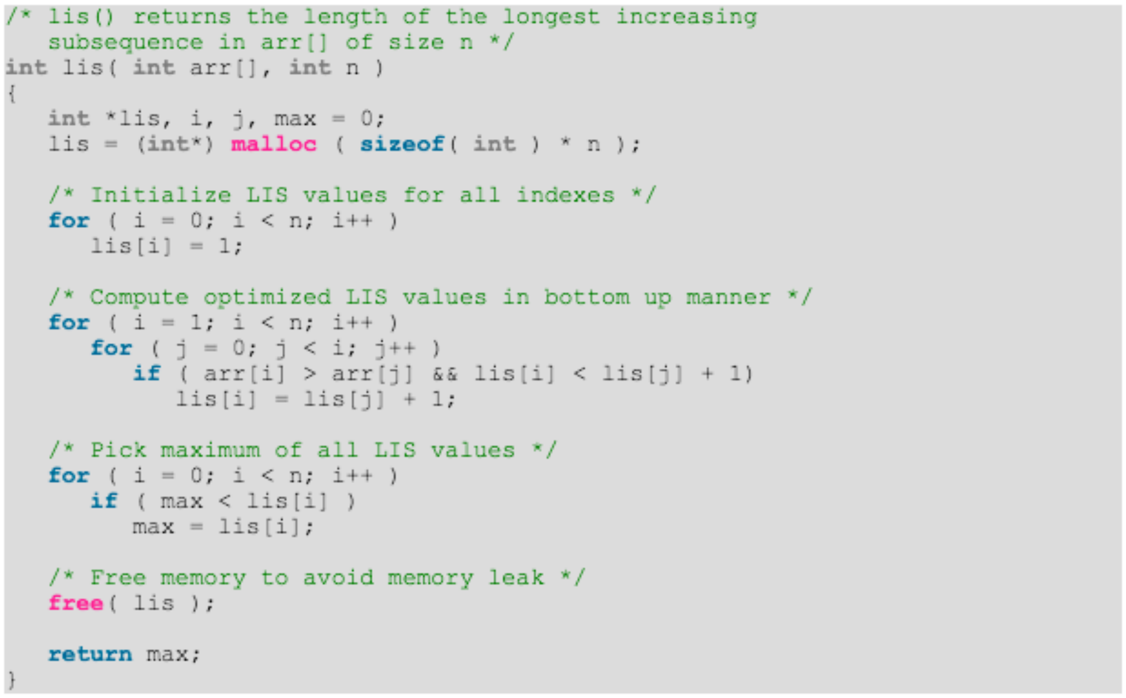
Longest Increasing Subsequence
Ex: {10, 22, 9, 33, 21, 50, 41, 60, 80}
Output: 6, {10, 22, 33, 50, 60, 80}

Let L(i) = length of LIS in till index i such that arr[i] is part of LIS and arr[i] is the last element in LIS.

L(i) = {1 + Max(L(j))} where j < i and arr[j] < arr[i]. If no such j, then L(i) = 1

To get LIS of a given array, need to return max(L(i)) where 0 < i < n.



```
/* 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;
}
```

Time: $O(n^2)$

<u>Longest Common Subsequence</u>

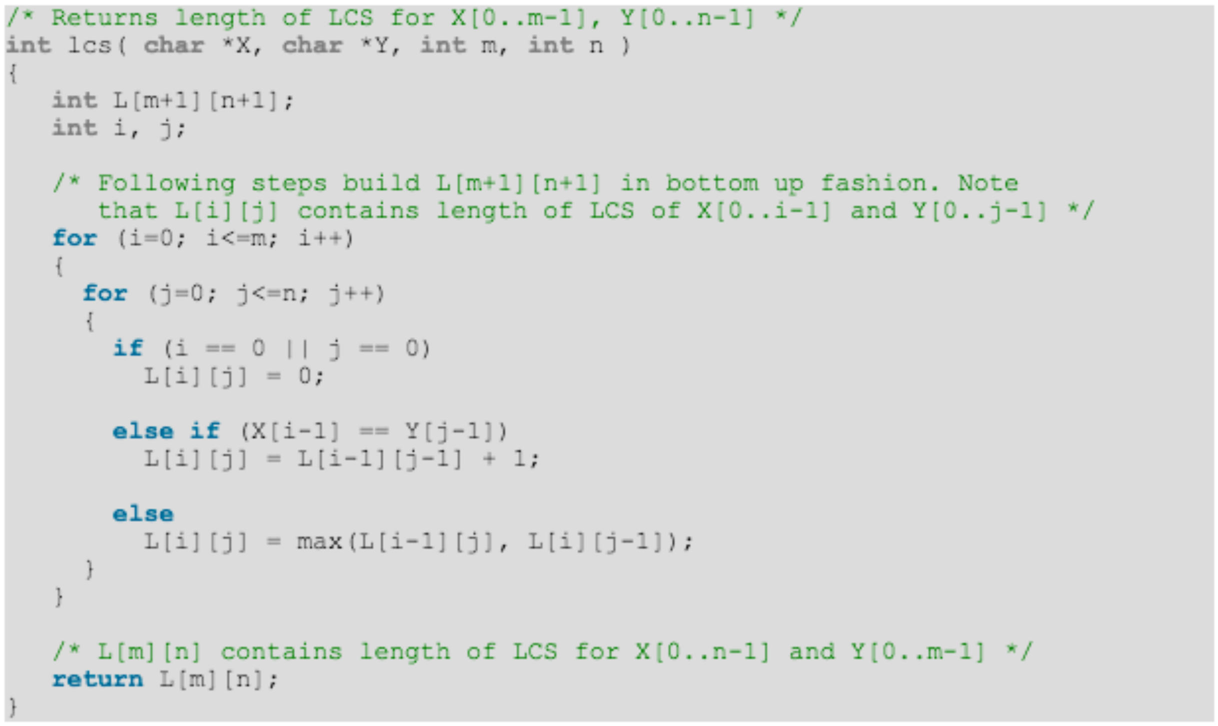
Ex:

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

If last characters of both sequences match (X[m-1] == Y[n-1]):
L(i,j) = L(i-1, j-1) + 1

If last characters of both sequences do not match (X[m-1] != Y[n-1]):
L(i,j) = Max{L(i-1, j), (i, j-1)}

*Fill the table row by row.



```
/* 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]);
     }
   }

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

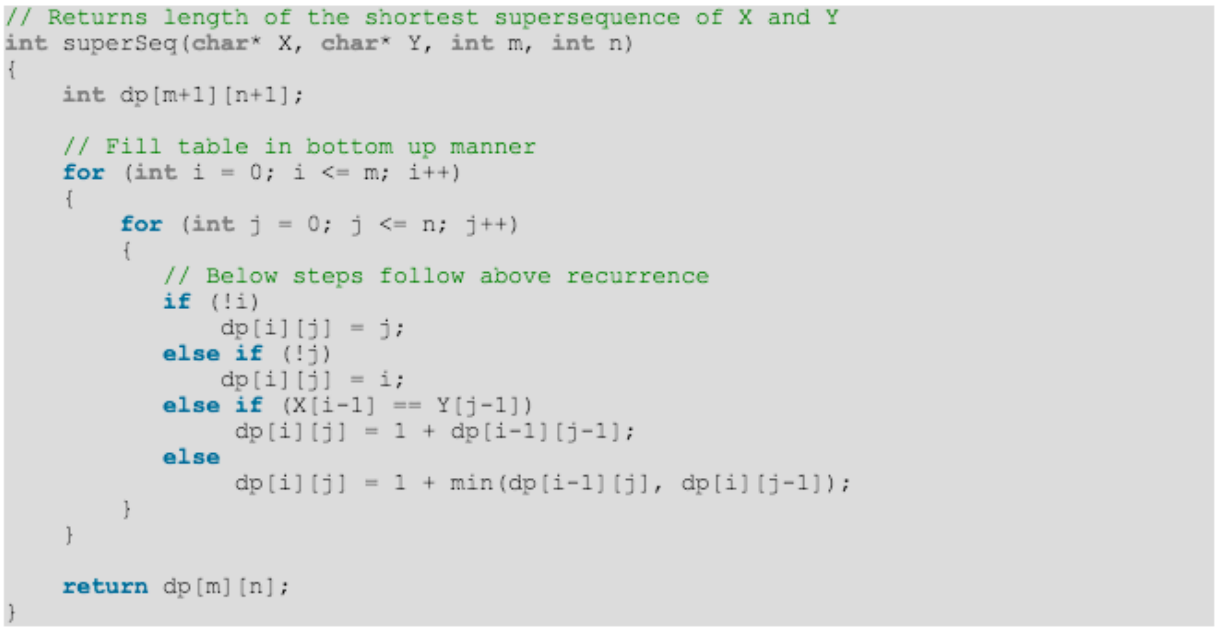
Time: O(mn)

<u>Shortest Common Supersequence</u>

Given two strings, find the shortest string that has both str1 and str2 as subsequences.
Ex:
Input: str1 = “geek”, str2 = “eke”
Output: “geeke”

Input: str = “AGGTAB”, str2 = “GXTXAYB”
Output: “AGXGTXAYB”

1) Find Longest Common Subsequence (lcs) of two given strings. For example, lcs of “geek” and “eke” is “ek”.
2) Insert non-lcs characters (in their original order in strings) to the lcs found above, and return the result. So “ek” becomes “geeke” which is shortest common supersequence.
Let us consider another example, str1 = “AGGTAB” and str2 = “GXTXAYB”. LCS of str1 and str2 is “GTAB”. Once we find LCS, we insert characters of both strings in order and we get “AGXGTXAYB”



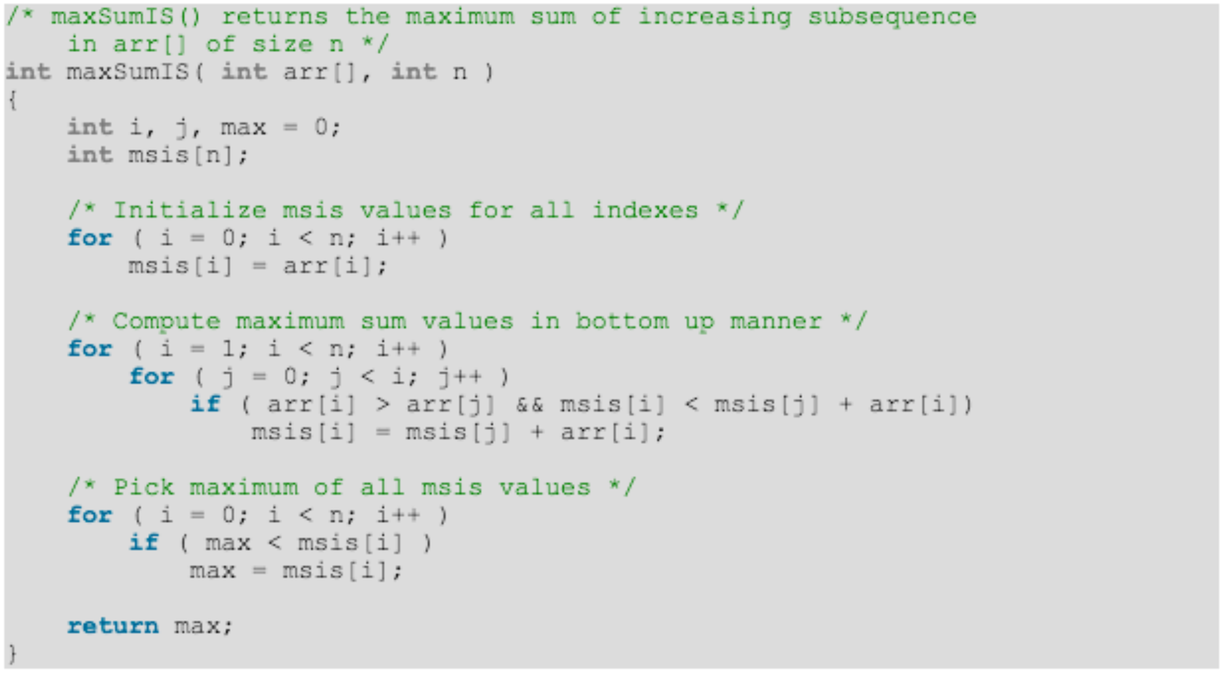
```
// Returns length of the shortest supersequence of X and Y
int superSeq(char* X, char* Y, int m, int n)
{
    int dp[m+1][n+1];

    // Fill table in bottom up manner
    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
           // Below steps follow above recurrence
           if (!i)
               dp[i][j] = j;
           else if (!j)
               dp[i][j] = i;
           else if (X[i-1] == Y[j-1])
                dp[i][j] = 1 + dp[i-1][j-1];
           else
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1]);
        }
    }

    return dp[m][n];
}
```

Time: O(mn)

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 integers 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.

Variation of LIS problem. Just change to use sum instead of length of increasing subsequence.



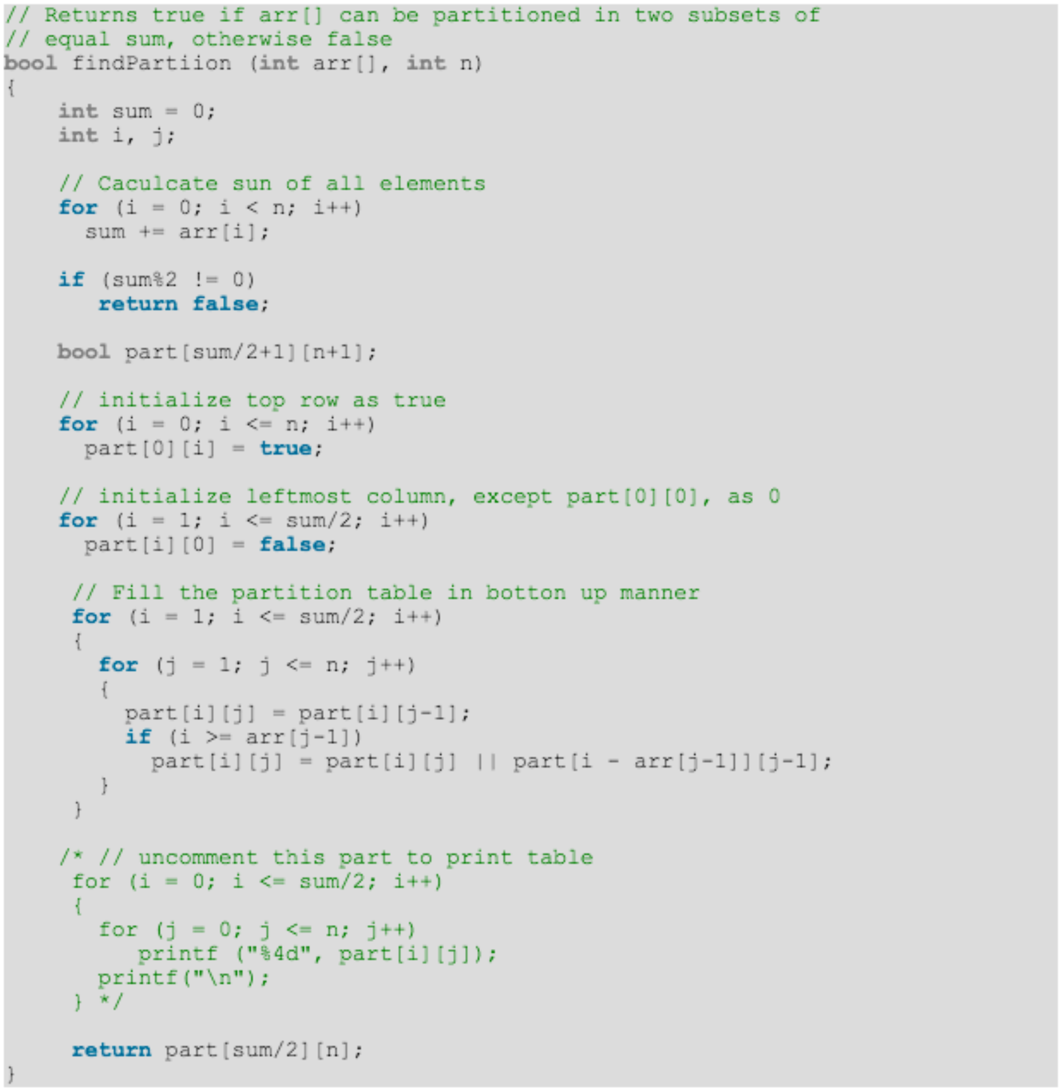
```
/* maxSumIS() returns the maximum sum of increasing subsequence
    in arr[] of size n */
int maxSumIS( int arr[], int n )
{
    int i, j, max = 0;
    int msis[n];

    /* 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;
}
```

Time: O(n^2)

Partition Problem

Determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is the same.
Ex: arr[] = {1, 5, 11, 5}
Output: true. Array can be partitioned as {1, 5, 5} and {11}

Steps:
1) Calculate sum of the array. If sum is odd, there cannot 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.



```
// 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];
}
```

Time: O(sum * n)
Space: O(sum * n)

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.
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:

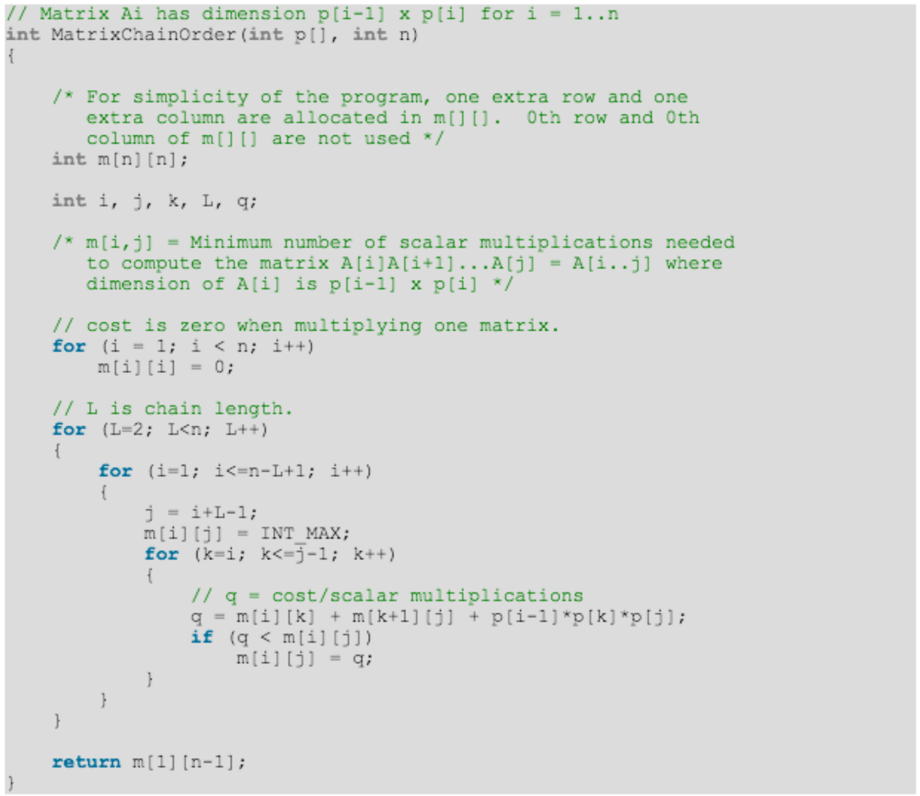
(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Look at substrings for $1 <= i <= j <= n$. Let $C(i,j)$ = min cost of computing $A_i \times A_{i+1} \times \ldots \times A_j$

Try all k for last split, so that last multiplication is:

$(A_i \times \ldots \times A_k)$	x	$(A_{k+1} \times \ldots \times A_j)$
costs $C(i,k)$	$m_{i-1}m_km_j$	costs $C(k+1, j)$

Thus, $C(i,j) = \min_{i <= k <= j-1} \{ C(i,k) + C(k+1,j) + m_{i-1}m_km_j \}$
Base Case: $C(i,i) = 0$
Fill the table by width $s = j - i$



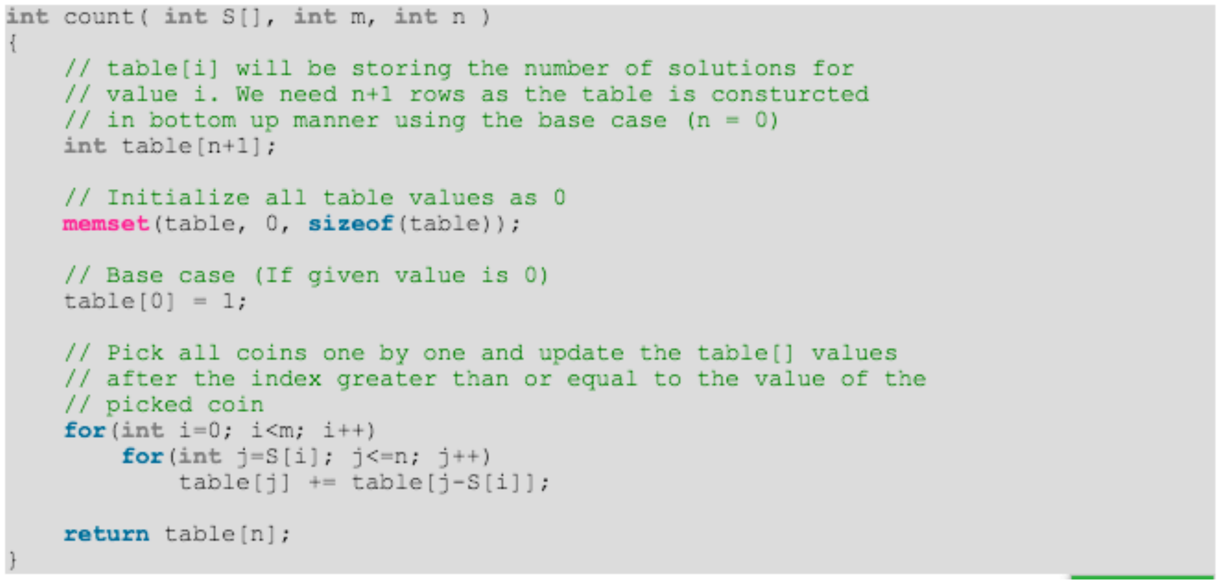
```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* 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
       dimension 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: $O(n^3)$
Space: $O(n^2)$

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.

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.



```
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];
}
```

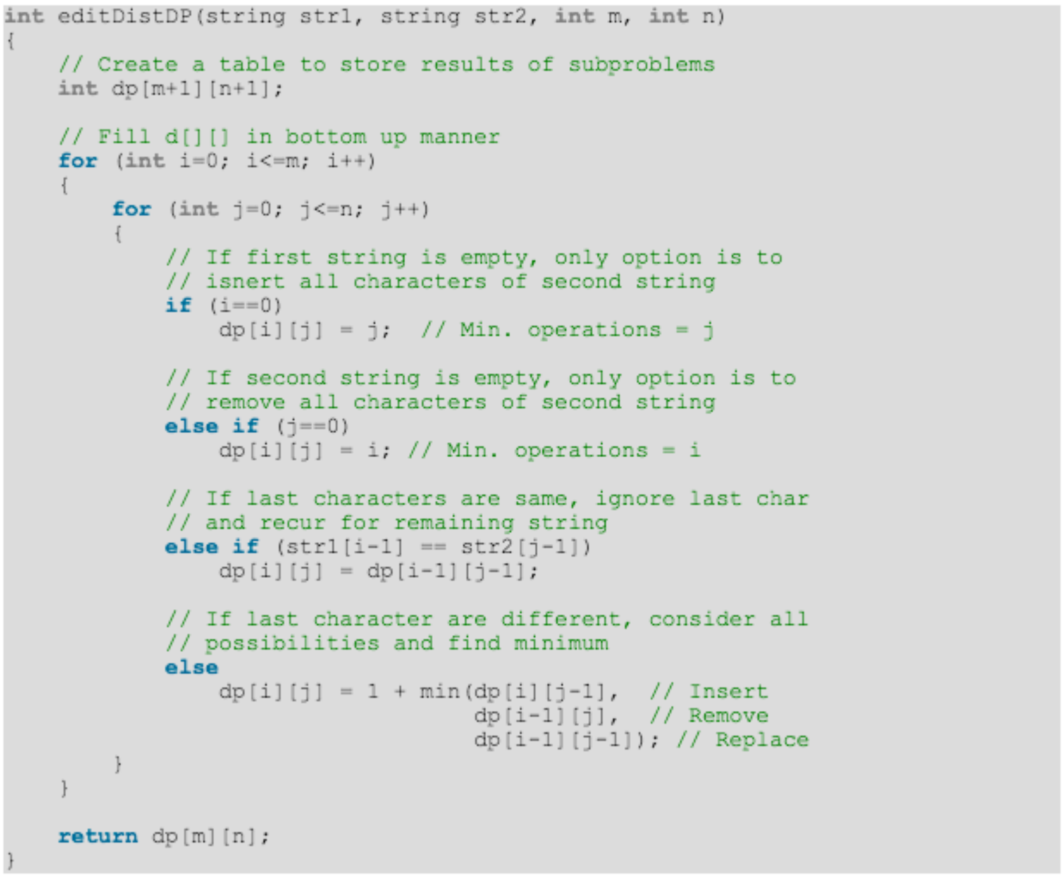
Time: O(mn)

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'.
Insert, Remove, Replace

Subproblems: Length of first string and length of 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, recur for lengths m-1 and n-1.
2. Else, if last characters are not same. Consider all operations on 'str1', consider all three operations on last character of first string, and recursively compute min cost for all three operations and take min of three values.
Insert: Recur for m and n-1
Remove: Recur for m-1 and n
Replace: Recur for m-1 and n-1



```
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];
}
```

Time: O(mn)
Space: O(mn)

Word Wrap

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.

Total cost function is sum of cubes of extra spaces.
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.

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

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

Optimal Binary Search Tree

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

The optimal cost for freq[i…j] can be recursively calculated using the following formula:

$$optCost(i, j) = \sum_{k=i}^{j} freq[k] + \min_{r=i}^{j}[optCost(i, r-1) + optCost(r+1, j)]$$

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.

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 2 pieces of lengths 2 and 6).

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

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 cutRod(n) be the required (best possible price) value for a rod of length n.
cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

```
/* 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: $O(n^2)$

You are given a wooden stick of length X with m markings on it at arbitrary places (integral), and the markings suggest where the cuts are to be made accordingly. For chopping a L-length stick into two pieces, the carpenter charges L dollars (does not matter whether the two pieces are of equal length or not, i.e, the chopping cost is independent of where the chopping point is).
Design a DP algorithm that calculates the minimum overall cost. **Time: $O(n^3)$**
Cost[i,j] = optimal cost of chopping the portion of the original stick occupied by piece i through piece j (with all intermediate pieces included).
Cost [i,j] = min $_{i \le k \le j}$ { Cost(i,k) + Cost(k+1,j) } + length of stick covering piece i though piece j

```
for $i = n$ to 1 do
    for $j = i$ to $n$ do
        for $k = i$ to $j$ do
            Cost $(i,j)$ = min { Cost $(i,j)$ , Cost$(i,k)$ + Cost$(k+1,j)$ + length$_{i\ to\ j}$}
```

<u>Bellman Ford</u>

Algorithm:

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.

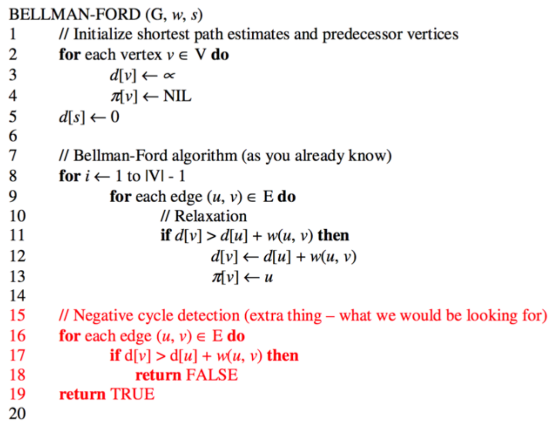
1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

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 doesn't 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

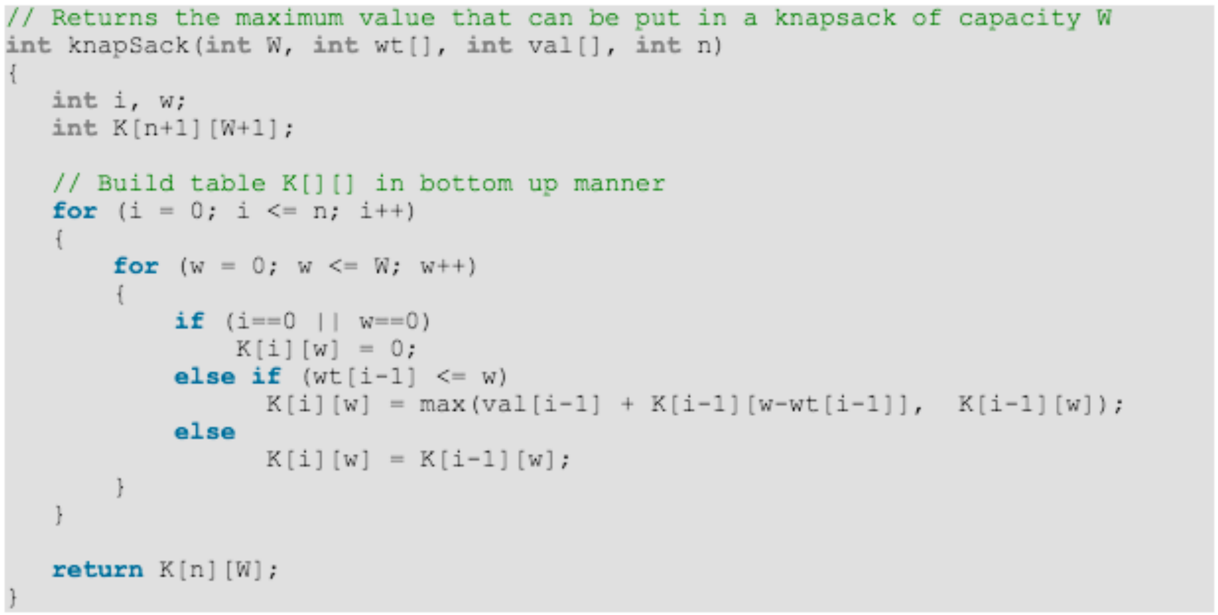
Time: O(V * E)



```
BELLMAN-FORD (G, w, s)
    // Initialize shortest path estimates and predecessor vertices
    for each vertex v ∈ V do
        d[v] ← ∝
        π[v] ← NIL
    d[s] ← 0

    // Bellman-Ford algorithm (as you already know)
    for i ← 1 to |V| - 1
        for each edge (u, v) ∈ E do
            // Relaxation
            if d[v] > d[u] + w(u, v) then
                d[v] ← d[u] + w(u, v)
                π[v] ← u

    // Negative cycle detection (extra thing – what we would be looking for)
    for each edge (u, v) ∈ E do
        if d[v] > d[u] + w(u, v) then
            return FALSE
    return TRUE
```

0-1 Knapsack
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 don't pick it (0-1 property).

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.



```
// 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];
}
```

Time: O(nW)

(Unlimited supply of each object)

$T(b) = \max_i \{ T(b - w_i) + v_i : 1 <= i <= n, w_i <= b \}$

```
for b = 0 -> B
    T(b) = 0
    for i = 1 -> n
        if (w_i <= b & T(b) < T(b - w_i) + v_i)
            then T(b) = T(b - w_i) + v_i
return (T(b))
```

Time: O(nB)

(Prefix of objects & prefix of capacity)

if $w_i <= b$: then don't include that object, so T(i,b) = T(i-1,b)

or include object and weight goes down and value goes up: $T(i,b) = v_i + T(i-1,b-w_i)$

Therefore, $T(i,b) = \max \{T(i-1,b), v_i + T(i-1,b-w_i)\}$

```
for b = 0 to B, T(0,b) = 0
for i = 1 to n:
    for b = 0 to B:
       if (w_i > b): T(i,b) = T(i-1,b)
       else : T(i,b) = max {T(i-1,b), v_i + T(i-1,b-w_i)}
return (T(n,b))
```

Minimum Number of Coins That Make a Given Value

Given a value V, if we want to make change for V cents, and we have infinite supply of each of C = {C1, C2, .. , Cm} valued coins, what is the minimum number of coins to make change?

Input: coins[] = {25, 10, 5}, V = 30
Output: Minimum 2 coins required
We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, V = 11
Output: Minimum 2 coins required
We can use one coin of 6 cents and 1 coin of 5 cents

Let C[p] be the min number of coins needed to make change for p cents.
Let x be the value of the first coin used in the optimal solution.
Then C[p] = 1 + C[p – x].

We don't know x so we will try all possible x and take the minimum.
d is the array of denomination values, k is the number of denominations, and n is the amount for which change is to be made.

$$C[p] = \begin{cases} \min_{i:d_i \le p}\{C[p-d_i]+1\} & \textbf{if } p > 0 \\ 0 & \textbf{if } p = 0 \end{cases}$$

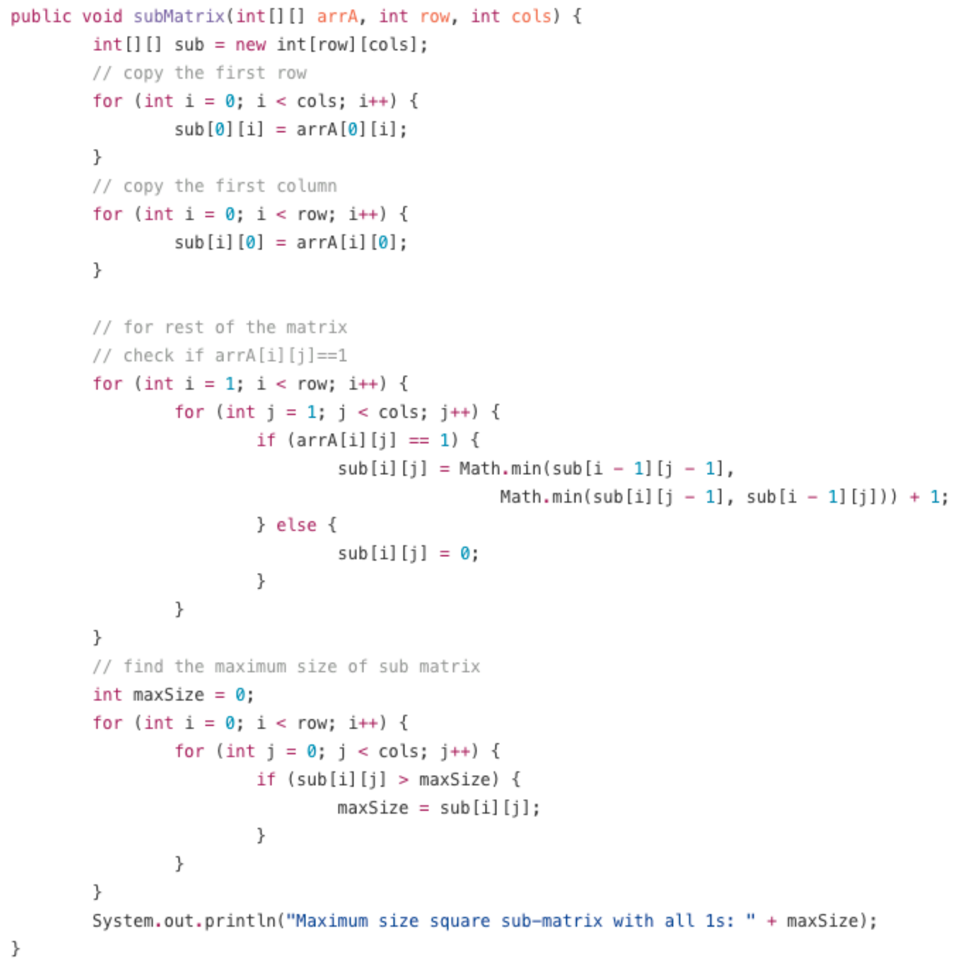
```
CHANGE$(d, k, n)$
1  $C[0] \leftarrow 0$
2  for $p \leftarrow 1$ to $n$
3          $min \leftarrow \infty$
4          for $i \leftarrow 1$ to $k$
5              if $d[i] \le p$ then
6                  if $1 + C[p - d[i]] < min$ then
7                      $min \leftarrow 1 + C[p - d[i]]$
8                      $coin \leftarrow i$
9          $C[p] \leftarrow min$
10         $S[p] \leftarrow coin$
11 return $C$ and $S$
```

Time: O(nk)
Space: O(n)

Maximum Size Square Sub-Matrix With All 1’s
Given a binary matrix, find out the maximum size square sub-matrix with all 1’s.

Algorithm:

- Create an auxiliary array of the same size as given input array. Fill the auxiliary array with maximum size square sub-matrix with all 1’s possible with respect to the particular cell.
- Once the auxiliary is filled, scan it and find out the max entry in it, this will be the size of max size square sub-matrix with all 1’s in the given matrix.
- Filling the aux array:
    - Copy the first row and first column from the given array to the aux array
    - For filling rest of cells, check if particular cell’s value is 0 in given array. If yes, then put 0 in aux array
    - Check if particular cell’s value is 1 in given array. If yes, then put min(value in left cell, top cell and left-top diagonal cell) + 1 in aux cell



```java
public void subMatrix(int[][] arrA, int row, int cols) {
        int[][] sub = new int[row][cols];
        // copy the first row
        for (int i = 0; i < cols; i++) {
                sub[0][i] = arrA[0][i];
        }
        // copy the first column
        for (int i = 0; i < row; i++) {
                sub[i][0] = arrA[i][0];
        }

        // for rest of the matrix
        // check if arrA[i][j]==1
        for (int i = 1; i < row; i++) {
                for (int j = 1; j < cols; j++) {
                        if (arrA[i][j] == 1) {
                                sub[i][j] = Math.min(sub[i - 1][j - 1],
                                                Math.min(sub[i][j - 1], sub[i - 1][j])) + 1;
                        } else {
                                sub[i][j] = 0;
                        }
                }
        }
        // find the maximum size of sub matrix
        int maxSize = 0;
        for (int i = 0; i < row; i++) {
                for (int j = 0; j < cols; j++) {
                        if (sub[i][j] > maxSize) {
                                maxSize = sub[i][j];
                        }
                }
        }
        System.out.println("Maximum size square sub-matrix with all 1s: " + maxSize);
}
```

Time: O(row * col)
Space: O(row * col)

Box Stacking

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.

Algorithm:

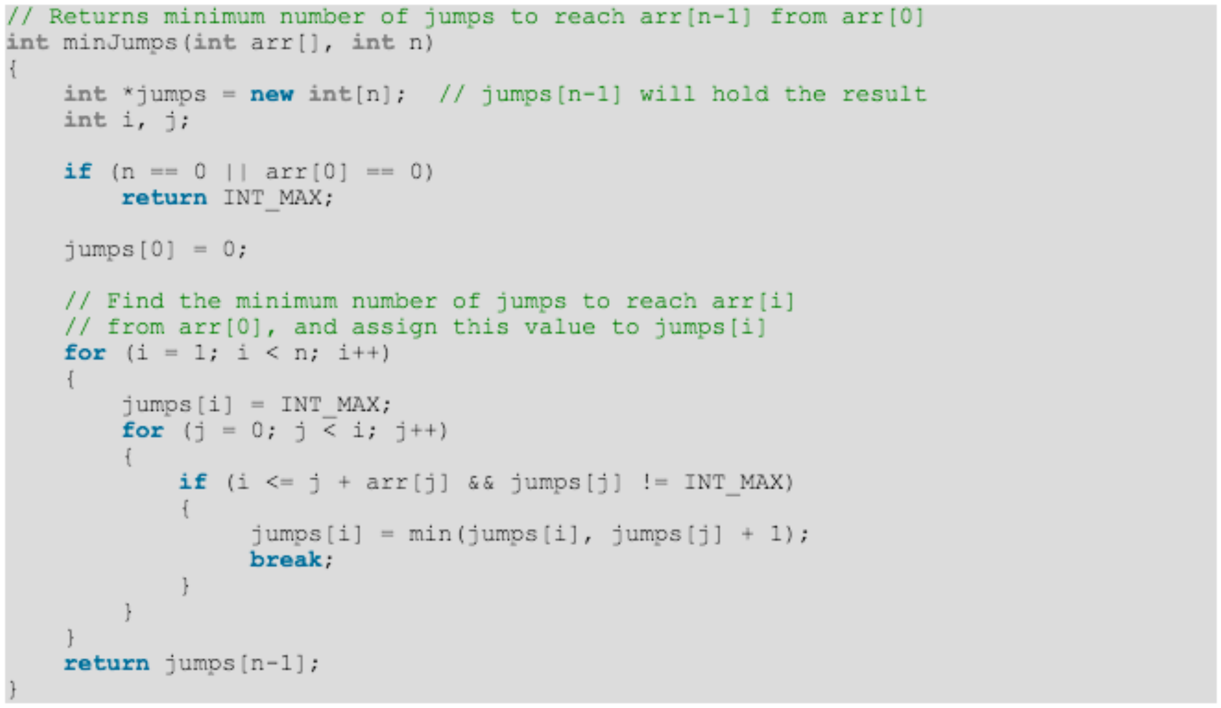
1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.
2) Sort the above generated 3n boxes in decreasing order of base area.
3) After sorting the boxes, the problem is 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

Time: $O(n^2)$
Space: O(n)

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.

Build a jumps[] array from left to right such that jumps[i] indicates the min number of jumps needed to reach arr[i] from arr[0]. Finally, return jumps[n-1].



```
// 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;

    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];
}
```

Time: O(n^2)

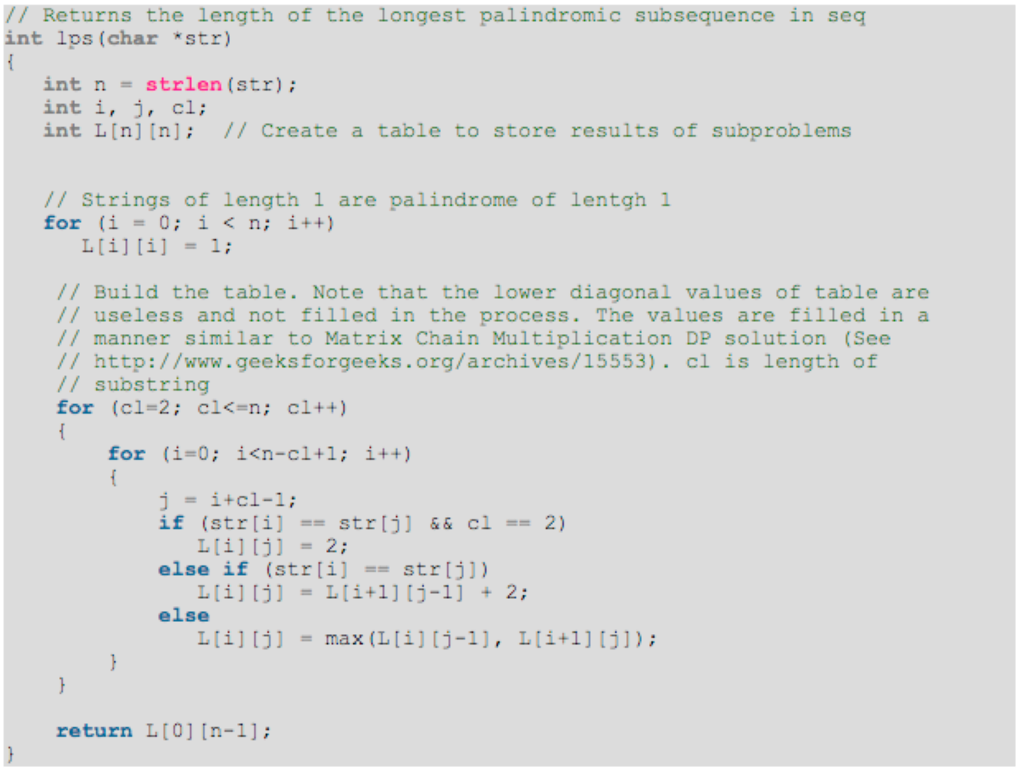
Longest Palindromic Subsequence
Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is “BBABCBCAB”, then the output should be 7 as “BABCBAB” is the longest palindromic subsequence in it. “BBBBB” and “BBCBB” are also palindromic subsequences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

Algorithm:
Given Sequence A[0…n-1]
LPS[0…n-1] : longest palindromic subsequence of the given sequence

Check the first and last characters of the sequence. There are 2 possibilities: either both characters are same or different.

- If both characters are same, add 2 to the result and remove both the characters and solve problem for remaining subsequence.
- If both characters are different, then solve the problem twice, ignore first character (keep last character) and solve remaining subsequence and then ignore last character (keep first character) and solve it and pick the best out of 2 results.

```
if (input[i] == input[j])
    T[i][j] = T[i+1][j-1] + 2        // look diagonally across
else
    T[i][j] = max(T[i+1][j], T[i][j-1])
```



```
// 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: O(n^2)

<u>**Longest Palindromic Substring**</u>

(continuous)

Given a string, find the longest substring which is palindrome. For example, if the given string is "forgeeksskeegfor", the output should be "geeksskeeg".

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.

```
// 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;

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

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

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

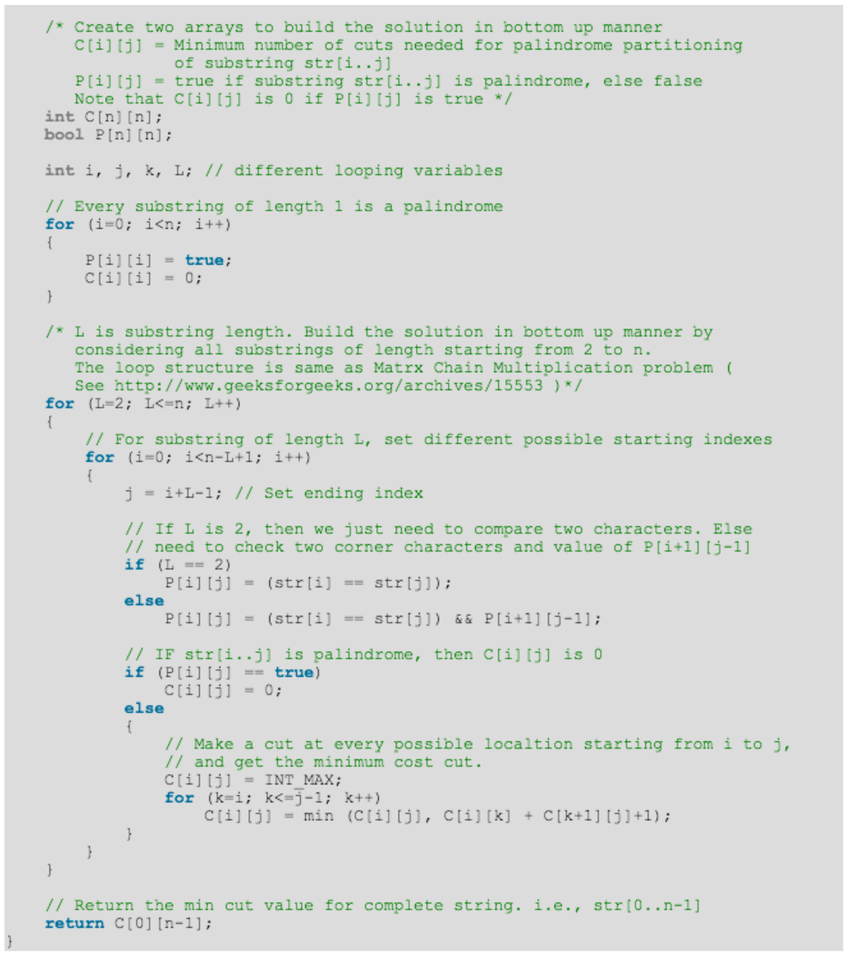
Time: O(n^2)

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.

Algorithm:

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.

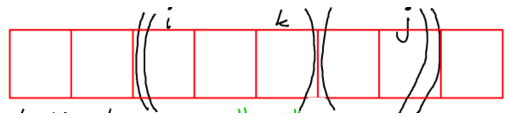


```
    /* Create two arrays to build the solution in bottom up manner
       C[i][j] = Minimum number of cuts needed for palindrome partitioning
                 of substring str[i..j]
       P[i][j] = true if substring str[i..j] is palindrome, else false
       Note that C[i][j] is 0 if P[i][j] is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

Time: O(n^3)

```
if isPalindrome(i,j)
    T[i][j] = 0
else
    T[i][j] = 1 + min{T[i][k] + T[k+1][j]} where k = i … j-1
```

Counting Boolean Parenthesizations



T(i,j) = # of ways to parenthesize symbols i…j such that this subexpression evaluates to true
F(i,j) = # of ways to parenthesize symbols i…j such that this subexpression evaluates to false

Base Case: T(i,i) , F(i,i)

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ Total(i,k) * Total(k+1,j) - F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * F(k+1,j) + F(i,k) * T(k+1) & \text{If operator[k] is '\^{}'} \end{cases}$$
$$\mathrm{Total(i,\ j) = T(i,\ j) + F(i,\ j)}$$

$$F(i,j) = \sum_{k=i}^{j-1} \begin{cases} Total(i,k) * Total(k+1,j) - T(i,k) * T(k+1,j) & \text{If operator[k] is '\&'} \\ F(i,k) * F(k+1,j) & \text{If operator[k] is '|'} \\ T(i,k) * T(k+1,j) + F(i,k) * F(k+1) & \text{If operator[k] is '\^{}'} \end{cases}$$
$$\mathrm{Total(i,\ j) = T(i,\ j) + F(i,\ j)}$$

For XOR should be: (mistake above)
T(i,j) = T(i,k) * F(k+1,j) + F(i,k) * T(k+1,j)
F(i,j) = T(i,k) * T(k+1,j) + F(i,k) * F(k+1,j)

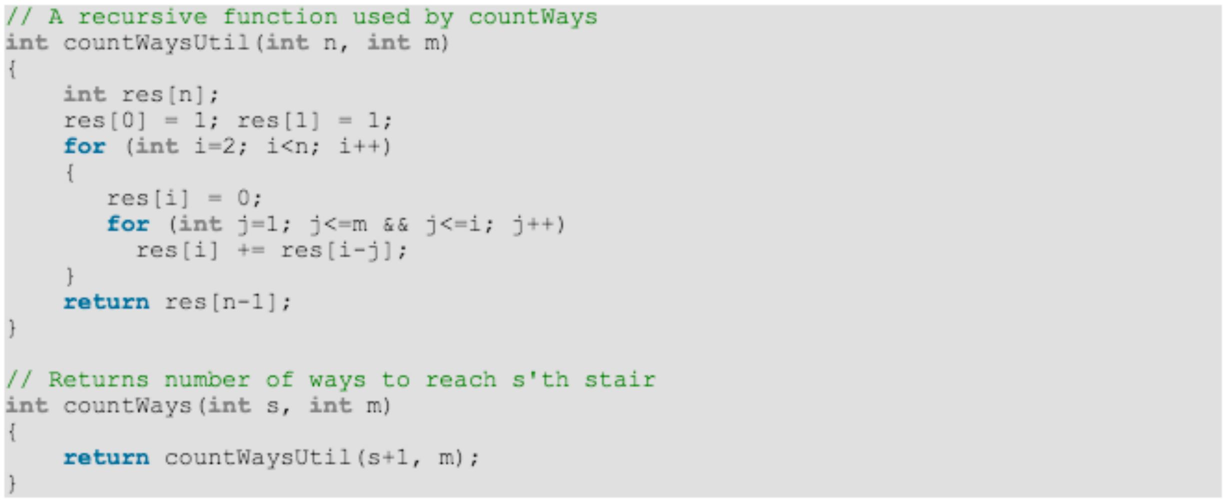
Time: $O(n^3)$

Count Ways to Reach the nth Stair

There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways, the person can reach the top.

```
Input: n = 4

Output: 5

(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)
```

Similar to Fibonacci



```
// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    int res[n];
    res[0] = 1; res[1] = 1;
    for (int i=2; i<n; i++)
    {
       res[i] = 0;
       for (int j=1; j<=m && j<=i; j++)
         res[i] += res[i-j];
    }
    return res[n-1];
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}
```

Time: O(mn)

<u>Find the minimum cost to reach destination using a train</u>

There are N stations on route of a train. The train goes from station 0 to N-1. The ticket cost for all pair of stations (i, j) is given where j is greater than i. Find the minimum cost to reach the destination.

```
// This function returns the smallest possible cost to
// reach station N-1 from station 0.
int minCost(int cost[][N])
{
    // dist[i] stores minimum cost to reach station i
    // from station 0.
    int dist[N];
    for (int i=0; i<N; i++)
       dist[i] = INF;
    dist[0] = 0;

    // Go through every station and check if using it
    // as an intermediate station gives better path
    for (int i=0; i<N; i++)
       for (int j=i+1; j<N; j++)
          if (dist[j] > dist[i] + cost[i][j])
             dist[j] = dist[i] + cost[i][j];

    return dist[N-1];
}
```

Optimal Strategy Game Pick

T[i][j].first = max(T[i+1][j].second + val[i], T[i][j-1].second + val[j]

T[i][j].second = T[i+1][j].first or T[i][j-1].first

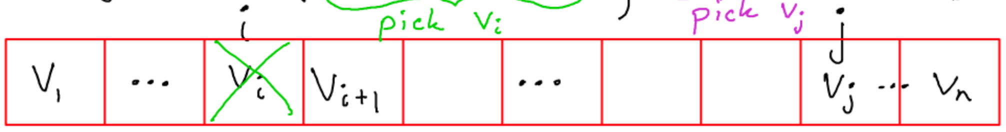
Input: row of $\underset{\text{even}}{\boxed{n}}$ coins of values $v_1 \cdots v_n$

Goal: maximize value of coins selected. $O(n^2)$

$V(i,j)$: max value we can <u>definitely</u> win if it's our turn and only coins $v_i \cdots v_j$ remain.

$V(i,i)$ $V(i,i+1)$ $V(i,i+2)$ $V(i,i+3)$

$$V(i,j) = \max\left\{ \underbrace{\min\left\{ \begin{array}{l} V(i+1,j-1), \\ V(i+2,j) \end{array} \right\} + V_i}_{\text{pick } v_i}, \underbrace{\min\left\{ \begin{array}{l} V(i,j-2), \\ V(i+1,j-1) \end{array} \right\} + V_j}_{\text{pick } v_j} \right\}$$

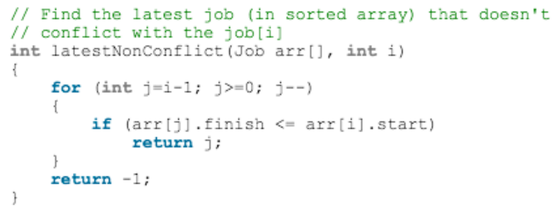


Weighted Job Scheduling

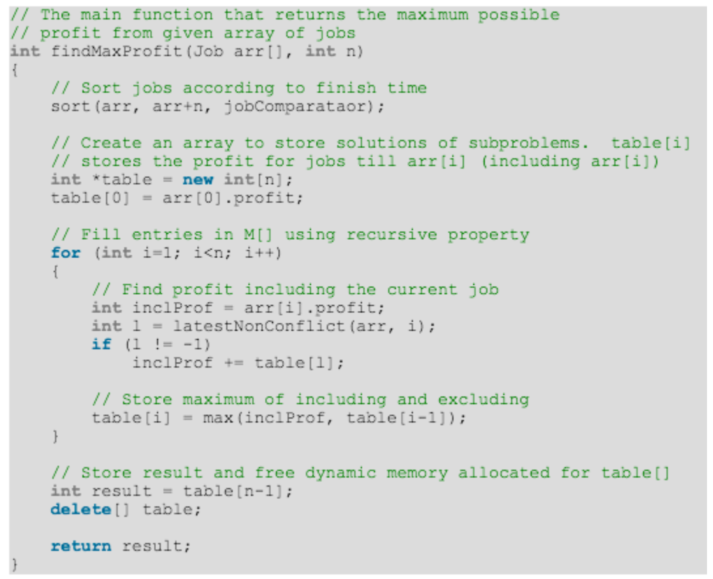
Given N jobs where every job is represented by following 3 elements in it: start time, finish time, profit or value associated.

```
Input: Number of Jobs n = 4

       Job Details {Start Time, Finish Time, Profit}

       Job 1:  {1, 2, 50}

       Job 2:  {3, 5, 20}

       Job 3:  {6, 19, 100}

       Job 4:  {2, 100, 200}

Output: The maximum profit is 250.

We can get the maximum profit by scheduling jobs 1 and 4.

Note that there is longer schedules possible Jobs 1, 2 and 3

but the profit with this schedule is 20+50+100 which is less than 250.
```

```
if (doNotOverlap(i,j))
    T[i][j] = max(T[i], T[j] + profit[i])
```



```
// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]
int latestNonConflict(Job arr[], int i)
{
    for (int j=i-1; j>=0; j--)
    {
        if (arr[j].finish <= arr[i].start)
            return j;
    }
    return -1;
}
```

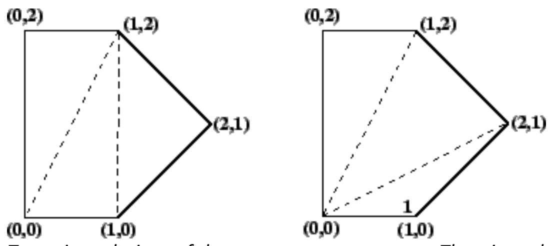


```
// The main function that returns the maximum possible
// profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, jobComparataor);

    // Create an array to store solutions of subproblems.  table[i]
    // stores the profit for jobs till arr[i] (including arr[i])
    int *table = new int[n];
    table[0] = arr[0].profit;

    // Fill entries in M[] using recursive property
    for (int i=1; i<n; i++)
    {
        // Find profit including the current job
        int inclProf = arr[i].profit;
        int l = latestNonConflict(arr, i);
        if (l != -1)
            inclProf += table[l];

        // Store maximum of including and excluding
        table[i] = max(inclProf, table[i-1]);
    }

    // Store result and free dynamic memory allocated for table[]
    int result = table[n-1];
    delete[] table;

    return result;
}
```

Time: O(n^2)

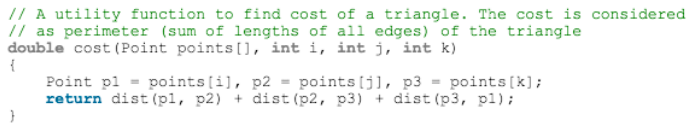
Minimum Cost Polygon Triangulation

A triangulation of a convex polygon is formed by drawing diagonals between non-adjacent vertices (corners) such that the diagonals never intersect. The problem is to find the cost of triangulation with the minimum cost. The cost of a triangulation is sum of the weights of its component triangles. Weight of each triangle is its perimeter (sum of lengths of all sides)

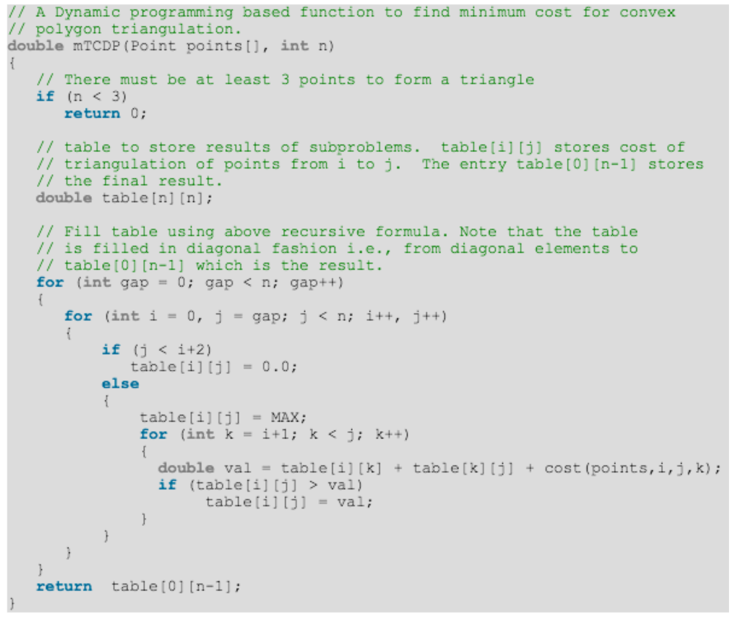


Two triangulations of the same convex pentagon. The triangulation on the left has a cost of 8 + 2√2 + 2√5 (approximately 15.30), the one on the right has a cost of 4 + 2√2 + 4√5 (approximately 15.77).

```
Let Minimum Cost of triangulation of vertices from i to j be minCost(i, j)
If j <= i + 2 Then
  minCost(i, j) = 0
Else
  minCost(i, j) = Min { minCost(i, k) + minCost(k, j) + cost(i, k, j) }
                  Here k varies from 'i+1' to 'j-1'

Cost of a triangle formed by edges (i, j), (j, k) and (k, j) is
  cost(i, j, k)  = dist(i, j) + dist(j, k) + dist(k, j)
```



```
// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}
```



```
// A Dynamic programming based function to find minimum cost for convex
// polygon triangulation.
double mTCDP(Point points[], int n)
{
   // There must be at least 3 points to form a triangle
   if (n < 3)
      return 0;

   // table to store results of subproblems.  table[i][j] stores cost of
   // triangulation of points from i to j.  The entry table[0][n-1] stores
   // the final result.
   double table[n][n];

   // Fill table using above recursive formula. Note that the table
   // is filled in diagonal fashion i.e., from diagonal elements to
   // table[0][n-1] which is the result.
   for (int gap = 0; gap < n; gap++)
   {
      for (int i = 0, j = gap; j < n; i++, j++)
      {
          if (j < i+2)
             table[i][j] = 0.0;
          else
          {
              table[i][j] = MAX;
              for (int k = i+1; k < j; k++)
              {
                double val = table[i][k] + table[k][j] + cost(points,i,j,k);
                if (table[i][j] > val)
                     table[i][j] = val;
              }
          }
      }
   }
   return  table[0][n-1];
}
```

Time: $O(n^3)$

Traveling Salesman Problem

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?

To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*.

We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},
 C(S, i) = dist(1, i)
Else if size of S is greater than 2.
 C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don't have nth in them.

Using the above recurrence relation, we can write dynamic programming based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

References:

http://www.geeksforgeeks.org/fundamentals-of-algorithms/#DynamicProgramming

https://people.cs.clemson.edu/~bcdean/dp_practice/

https://www.youtube.com/playlist?list=PLrmLmBdmIlpsHaNTPP_jHHDx_os9ItYXr

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/