Count all possible paths from top left to bottom right of a mXn matrix

The problem is to count all the possible paths from top left to bottom right of a mXn matrix with the constraints thatÂ ***from each cell you can either move only to right or down***
We have discussed a solution to print all possible paths, counting all paths is easier. Let NumberOfPaths(m, n) be the count of paths to reach row number m and column number n in the matrix, NumberOfPaths(m, n) can be recursively written as following.

```
#include <iostream>

using namespace std;


// Returns count of possible paths to reach cell at row number m and column

// number n from the topmost leftmost cell (cell at 1, 1)

int  numberOfPaths(int m, int n)

{

   // If either given row number is first or given column number is first

   if (m == 1 || n == 1)

        return 1;


   // If diagonal movements are allowed then the last addition

   // is required.

   return  numberOfPaths(m-1, n) + numberOfPaths(m, n-1);

           // + numberOfPaths(m-1,n-1);

}


int main()

{

        cout << numberOfPaths(3, 3);

        return 0;

}
```

Output:

```
6
```

The time complexity of above recursive solution is exponential. There are many overlapping subproblems. We can draw a recursion tree for numberOfPaths(3, 3) and see many overlapping subproblems. The recursion tree would be similar to Recursion tree for Longest Common Subsequence problem.

So this problem has both properties (seeÂ thisÂ andÂ this) of a dynamic programming problem. Like other typicalÂ Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array count[][] in bottom up manner using the above recursive formula.

```
#include <iostream>

using namespace std;


// Returns count of possible paths to reach cell at row number m and column

// number n from the topmost leftmost cell (cell at 1, 1)

int numberOfPaths(int m, int n)

{

    // Create a 2D table to store results of subproblems

    int count[m][n];


    // Count of paths to reach any cell in first column is 1

    for (int i = 0; i < m; i++)

        count[i][0] = 1;


    // Count of paths to reach any cell in first column is 1

    for (int j = 0; j < n; j++)

        count[0][j] = 1;


    // Calculate count of paths for other cells in bottom-up manner using

    // the recursive solution

    for (int i = 1; i < m; i++)

    {

        for (int j = 1; j < n; j++)


            // By uncommenting the last part the code calculatest he total

            // possible paths if the diagonal Movements are allowed

            count[i][j] = count[i-1][j] + count[i][j-1]; //+ count[i-1][j-1];


    }

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

}
```

```
// Driver program to test above functions

int main()

{

    cout << numberOfPaths(3, 3);

    return 0;

}
```

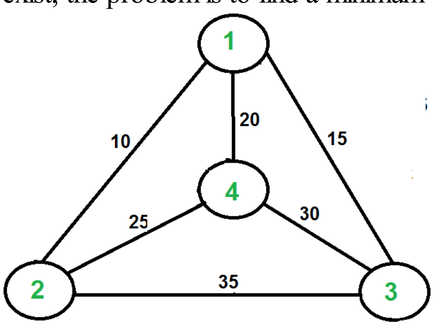
Output:

```
6
```

Time complexity of the above dynamic programming solution is O(mn).
Note the count can also be calculated using the formula (m-1 + n-1)!/(m-1)!(n-1)! as mentioned in the comments of this article.
This article is contributed by **Hariprasad NG**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
++

Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)

Travelling Salesman Problem (TSP): 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.
Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.
The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.
Following are different solutions for the traveling salesman problem.
Naive Solution:
1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.
Time Complexity: Ω(n!)
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.

We will soon be discussing approximate algorithms for travelling salesman problem.

References:

http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf
http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

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

++

Dynamic Programming | Set 36 (Maximum Product Cutting)

Given a rope of length n meters, cut the rope in different parts of integer lengths in a way that maximizes product of lengths of all parts. You must make at least one cut. Assume that the length of rope is more than 2 meters.

Examples:

```
Input: n = 2

Output: 1 (Maximum obtainable product is 1*1)



Input: n = 3

Output: 2 (Maximum obtainable product is 1*2)



Input: n = 4

Output: 4 (Maximum obtainable product is 2*2)



Input: n = 5

Output: 6 (Maximum obtainable product is 2*3)
```

```
Input: n = 10

Output: 36 (Maximum obtainable product is 3*3*4)
```

1) Optimal Substructure:
This problem is similar to [Rod Cutting Problem.](#) We can get the maximum product 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 maxProd(n) be the maximum product for a rope of length n. maxProd(n) can be written as following.
maxProd(n) = max(i*(n-i), maxProdRec(n-i)*i) for all i in {1, 2, 3 .. n}
2) Overlapping Subproblems
Following is simple recursive C++ implementation of the problem. The implementation simply follows the recursive structure mentioned above.

```
// A Naive Recursive method to find maxium product

#include <iostream>

using namespace std;


// Utility function to get the maximum of two and three integers

int max(int a, int b) { return (a > b)? a : b;}

int max(int a, int b, int c) { return max(a, max(b, c));}


// The main function that returns maximum product obtainable

// from a rope of length n

int maxProd(int n)

{

    // Base cases

    if (n == 0 || n == 1) return 0;


    // Make a cut at different places and take the maximum of all

    int max_val = 0;

    for (int i = 1; i < n; i++)

      max_val = max(max_val, i*(n-i), maxProd(n-i)*i);


    // Return the maximum of all values

    return max_val;

}
```

```
/* Driver program to test above functions */
int main()
{
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}
```

Output:

```
Maximum Product is 36
```

Considering the above implementation, following is recursion tree for a Rope of length 5.

```
mP() ---> maxProd()


                            mP(5)
                 /        /      \      \
                /        /        \       \
            mP(4)       mP(3)      mP(2)   mP(1)
           /  |  \       /  \        |
          /   |   \     /    \       |
     mP(3) mP(2) mP(1) mP(2) mP(1) mP(1)
    / \       |          |
   /   \      |          |
 mP(2) mP(1) mP(1)      mP(1)
```

In the above partial recursion tree, mP(3) 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 problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array val[] in bottom up manner.

```
// A Dynamic Programming solution for Max Product Problem
int maxProd(int n)
{
   int val[n+1];
   val[0] = val[1] = 0;
```

```
    // Build the table val[] in bottom up manner and return

    // the last entry from the table

    for (int i = 1; i <= n; i++)

    {

       int max_val = 0;

       for (int j = 1; j <= i/2; j++)

          max_val = max(max_val, (i-j)*j, j*val[i-j]);

       val[i] = max_val;

    }

    return val[n];

}
```

Time Complexity of the Dynamic Programming solution is O(n^2) and it requires O(n) extra space.
A Tricky Solution:
If we see some examples of this problems, we can easily observe following pattern.
The maximum product can be obtained be repeatedly cutting parts of size 3 while size is greater than 4, keeping the last part as size of 2 or 3 or 4. For example, n = 10, the maximum product is obtained by 3, 3, 4. For n = 11, the maximum product is obtained by 3, 3, 3, 2. Following is C++ implementation of this approach.

```
#include <iostream>

using namespace std;


/* The main function that teturns the max possible product */

int maxProd(int n)

{

   // n equals to 2 or 3 must be handled explicitly

   if (n == 2 || n == 3) return (n-1);


   // Keep removing parts of size 3 while n is greater than 4

   int res = 1;

   while (n > 4)

   {

       n -= 3;

       res *= 3; // Keep multiplying 3 to res

   }

   return (n * res); // The last part multiplied by previous parts
```

```
}


/* Driver program to test above functions */

int main()

{

    cout << "Maximum Product is " << maxProd(10);

    return 0;

}
```

Output:

```
Maximum Product is 36
```

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

Dynamic Programming | Set 35 (Longest Arithmetic Progression)

Given a set of numbers, find the **L**ength of the **L**ongest **A**rithmetic **P**rogression (**LLAP**) in it.
Examples:

```
set[] = {1, 7, 10, 15, 27, 29}

output = 3

The longest arithmetic progression is {1, 15, 29}


set[] = {5, 10, 15, 20, 25, 30}

output = 6

The whole set is in AP
```

For simplicity, we have assumed that the given set is sorted. We can always add a pre-processing step to first sort the set and then apply the below algorithms.
A **simple solution** is to one by one consider every pair as first two elements of AP and check for the remaining elements in sorted set. To consider all pairs as first two elements, we need to run a O(n^2) nested loop. Inside the nested loops, we need a third loop which linearly looks for the more elements in **A**rithmetic **P**rogression (**AP**). This process takes $O(n^3)$ time.
We can solve this problem in $O(n^2)$ time **using Dynamic Programming**. To get idea of the DP solution, let us first discuss solution of following simpler problem.
Given a sorted set, find if there exist three elements in Arithmetic Progression or not
Please note that, the answer is true if there are 3 or more elements in AP, otherwise false.
To find the three elements, we first fix an element as middle element and search for other two (one smaller and one greater). We start from the second element and fix every element as middle element. For an element set[j] to be middle of AP, there must exist

elements ‘set[i]‘ and ‘set[k]‘ such that set[i] + set[k] = 2*set[j] where 0 <= i < j and j < k <=n-1.
How to efficiently find i and k for a given j? We can find i and k in linear time using following simple algorithm.
1) Initialize i as j-1 and k as j+1
2) Do following while i >= 0 and j <= n-1
..........**a)** If set[i] + set[k] is equal to 2*set[j], then we are done.
……..**b)** If set[i] + set[k] > 2*set[j], then decrement i (do iâ€“-).
……..**c)** Else if set[i] + set[k] < 2*set[j], then increment k (do k++).
Following is C++ implementation of the above algorithm for the simpler problem.

```
// The function returns true if there exist three elements in AP

// Assumption: set[0..n-1] is sorted.

// The code strictly implements the algorithm provided in the reference.

bool arithmeticThree(int set[], int n)

{

    // One by fix every element as middle element

    for (int j=1; j<n-1; j++)

    {

        // Initialize i and k for the current j

        int i = j-1, k = j+1;


        // Find if there exist i and k that form AP

        // with j as middle element

        while (i >= 0 && k <= n-1)

        {

            if (set[i] + set[k] == 2*set[j])

                return true;

            (set[i] + set[k] < 2*set[j])? k++ : i–;

        }

    }


    return false;

}
```

See this for a complete running program.
How to extend the above solution for the original problem?
The above function returns a boolean value. The required output of original problem is **L**ength of the **L**ongest **A**rithmetic **P**rogression (**LLAP**) which is an integer value. If the given set has two or more elements, then the value of LLAP is at least 2 (Why?).
The idea is to create a 2D table L[n][n]. An entry L[i][j] in this table stores LLAP with set[i] and set[j] as first two elements of AP

and j > i. The last column of the table is always 2 (Why - see the meaning of L[i][j]). Rest of the table is filled from bottom right to top left. To fill rest of the table, j (second element in AP) is first fixed. i and k are searched for a fixed j. If i and k are found such that i, j, k form an AP, then the value of L[i][j] is set as L[j][k] + 1. Note that the value of L[j][k] must have been filled before as the loop traverses from right to left columns.

Following is C++ implementation of the Dynamic Programming algorithm.

```
// C++ program to find Length of the Longest AP (llap) in a given sorted set.

// The code strictly implements the algorithm provided in the reference.

#include <iostream>

using namespace std;


// Returns length of the longest AP subset in a given set

int lenghtOfLongestAP(int set[], int n)

{

    if (n <= 2)  return n;


    // Create a table and initialize all values as 2. The value of

    // L[i][j] stores LLAP with set[i] and set[j] as first two

    // elements of AP. Only valid entries are the entries where j>i

    int L[n][n];

    int llap = 2;  // Initialize the result


    // Fill entries in last column as 2. There will always be

    // two elements in AP with last number of set as second

    // element in AP

    for (int i = 0; i < n; i++)

        L[i][n-1] = 2;


    // Consider every element as second element of AP

    for (int j=n-2; j>=1; j--)

    {

        // Search for i and k for j

        int i = j-1, k = j+1;

        while (i >= 0 && k <= n-1)

        {
```

```
        if (set[i] + set[k] < 2*set[j])
            k++;

        // Before changing i, set L[i][j] as 2
        else if (set[i] + set[k] > 2*set[j])
        {   L[i][j] = 2, i--;   }

        else
        {
            // Found i and k for j, LLAP with i and j as first two
            // elements is equal to LLAP with j and k as first two
            // elements plus 1. L[j][k] must have been filled
            // before as we run the loop from right side
            L[i][j] = L[j][k] + 1;

            // Update overall LLAP, if needed
            llap = max(llap, L[i][j]);

            // Change i and k to fill more L[i][j] values for
            // current j
            i--; k++;
        }
    }

    // If the loop was stopped due to k becoming more than
    // n-1, set the remaining entties in column j as 2
    while (i >= 0)
    {
        L[i][j] = 2;
        i--;
    }
}
return llap;
```

```
}


/* Drier program to test above function*/

int main()

{

    int set1[] = {1, 7, 10, 13, 14, 19};

    int n1 = sizeof(set1)/sizeof(set1[0]);

    cout <<   lenghtOfLongestAP(set1, n1) << endl;


    int set2[] = {1, 7, 10, 15, 27, 29};

    int n2 = sizeof(set2)/sizeof(set2[0]);

    cout <<   lenghtOfLongestAP(set2, n2) << endl;


    int set3[] = {2, 4, 6, 8, 10};

    int n3 = sizeof(set3)/sizeof(set3[0]);

    cout <<   lenghtOfLongestAP(set3, n3) << endl;


    return 0;

}
```

Output:

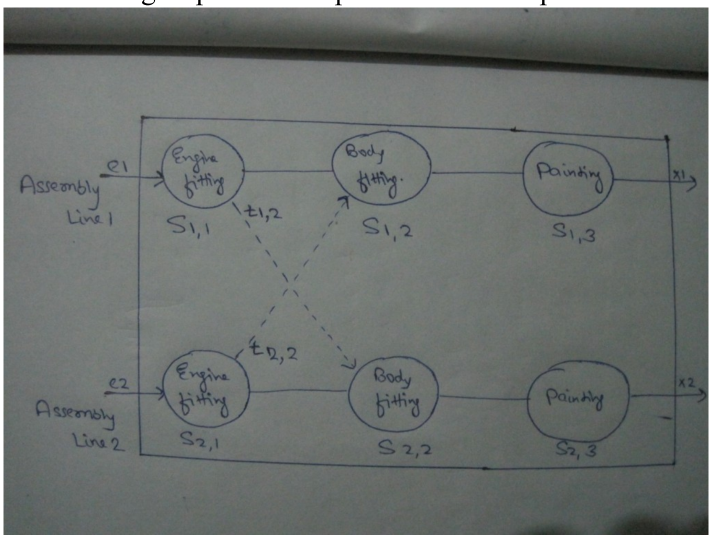
```
4

3

5
```

Time Complexity: $O(n^2)$
Auxiliary Space: $O(n^2)$
References:
http://www.cs.uiuc.edu/~jeffe/pubs/pdf/arith.pdf
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
+++

Dynamic Programming | Set 34 (Assembly Line Scheduling)

A car factory has two assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station j – 1 to station j on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for the two lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

The below figure presents the problem in a clear picture:



The following information can be extracted from the problem statement to make it simpler:

- Two assembly lines, 1 and 2, each with stations from 1 to n.
- A car chassis must pass through all stations from 1 to n in order(in any of the two assembly lines). i.e. it cannot jump from station i to station j if they are not at one move distance.
- The car chassis can move one station forward in the same line, or one station diagonally in the other line. It incurs an extra cost ti, j to move to station j from line i. No cost is incurred for movement in same line.
- The time taken in station j on line i is $a_{i, j}$.
- $S_{i, j}$ represents a station j on line i.

Breaking the problem into smaller sub-problems:

We can easily find the ith factorial if (i-1)th factorial is known. Can we apply the similar funda here?

If the minimum time taken by the chassis to leave station $S_{i, j-1}$ is known, the minimum time taken to leave station $S_{i, j}$ can be calculated quickly by combining $a_{i, j}$ and $t_{i, j}$.

T1(j) indicates the minimum time taken by the car chassis to leave station j on assembly line 1.

T2(j) indicates the minimum time taken by the car chassis to leave station j on assembly line 2.

Base cases:

The entry time e_i comes into picture only when the car chassis enters the car factory.

Time taken to leave first station in line 1 is given by:

T1(1) = Entry time in Line 1 + Time spent in station $S_{1,1}$

T1(1) = $e_1 + a_{1,1}$

Similarly, time taken to leave first station in line 2 is given by:

T2(1) = $e_2 + a_{2,1}$

Recursive Relations:

If we look at the problem statement, it quickly boils down to the below observations:

The car chassis at station $S_{1,j}$ can come either from station $S_{1, j-1}$ or station $S_{2, j-1}$.

Case #1: Its previous station is $S_{1, j-1}$

The minimum time to leave station $S_{1,j}$ is given by:

T1(j) = Minimum time taken to leave station $S_{1, j-1}$ + Time spent in station $S_{1, j}$

$T1(j) = T1(j-1) + a_{1, j}$

Case #2: Its previous station is $S_{2, j-1}$

The minimum time to leave station S1, j is given by:

T1(j) = Minimum time taken to leave station $S_{2, j-1}$ + Extra cost incurred to change the assembly line + Time spent in station $S_{1, j}$

$T1(j) = T2(j-1) + t_{2, j} + a_{1, j}$

The minimum time T1(j) is given by the minimum of the two obtained in cases #1 and #2.

$T1(j) = min((T1(j-1) + a_{1, j}), (T2(j-1) + t_{2, j} + a_{1, j}))$

Similarly the minimum time to reach station S2, j is given by:

$T2(j) = min((T2(j-1) + a_{2, j}), (T1(j-1) + t_{1, j} + a_{2, j}))$

The total minimum time taken by the car chassis to come out of the factory is given by:

Tmin = min(Time taken to leave station $S_{i,n}$ + Time taken to exit the car factory)

$Tmin = min(T1(n) + x_1, T2(n) + x_2)$

Why dynamic programming?

The above recursion exhibits overlapping sub-problems. There are two ways to reach station $S_{1, j}$:

1. From station $S_{1, j-1}$
2. From station $S_{2, j-1}$

So, to find the minimum time to leave station $S_{1, j}$ the minimum time to leave the previous two stations must be calculated(as explained in above recursion).

Similarly, there are two ways to reach station $S_{2, j}$:

1. From station $S_{2, j-1}$
2. From station $S_{1, j-1}$

Please note that the minimum times to leave stations $S_{1, j-1}$ and $S_{2, j-1}$ have already been calculated.

So, we need two tables to store the partial results calculated for each station in an assembly line. The table will be filled in bottom-up fashion.

Note:

In this post, the word â€œleaveâ€ has been used in place of â€œreachâ€ to avoid the confusion. Since the car chassis must spend a fixed time in each station, the word leave suits better.

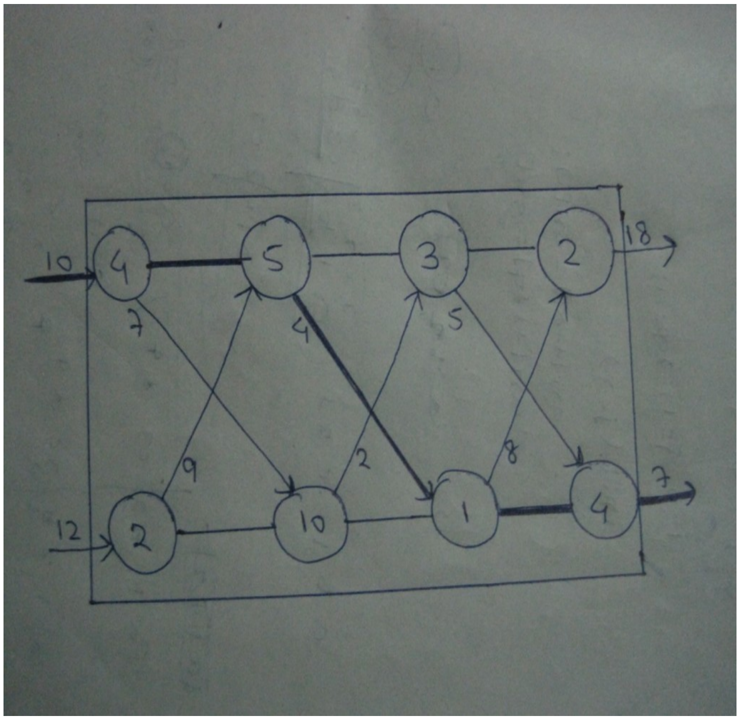
Implementation:

```
// A C program to find minimum possible time by the car chassis to complete

#include <stdio.h>

#define NUM_LINE 2

#define NUM_STATION 4


// Utility function to find minimum of two numbers

int min(int a, int b) { return a < b ? a : b; }


int carAssembly(int a[][NUM_STATION], int t[][NUM_STATION], int *e, int *x)

{
```

```
    int T1[NUM_STATION], T2[NUM_STATION], i;

    T1[0] = e[0] + a[0][0]; // time taken to leave first station in line 1
    T2[0] = e[1] + a[1][0]; // time taken to leave first station in line 2

    // Fill tables T1[] and T2[] using the above given recursive relations
    for (i = 1; i < NUM_STATION; ++i)
    {
        T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i]);
        T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i]);
    }

    // Consider exit times and retutn minimum
    return min(T1[NUM_STATION-1] + x[0], T2[NUM_STATION-1] + x[1]);
}

int main()
{
    int a[][NUM_STATION] = {{4, 5, 3, 2},
                {2, 10, 1, 4}};
    int t[][NUM_STATION] = {{0, 7, 4, 5},
                {0, 9, 2, 8}};
    int e[] = {10, 12}, x[] = {18, 7};

    printf("%d", carAssembly(a, t, e, x));

    return 0;
}
```

Output:

```
35
```



The bold line shows the path covered by the car chassis for given input values.

Exercise:

Extend the above algorithm to print the path covered by the car chassis in the factory.

References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest]

This article is compiled by **[Aashish Barnwal]**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

++

Dynamic Programming | Set 33 (Find if a string is interleaved of two other strings)

Given three strings A, B and C. Write a function that checks whether C is an interleaving of A and B. C is said to be interleaving A and B, if it contains all characters of A and B and order of all characters in individual strings is preserved.

We have discussed a simple solution of this problem [here]. The simple solution doesn’t work if strings A and B have some common characters. For example A = â€œXXYâ€, string B = â€œXXZâ€ and string C = â€œXXZXXXYâ€. To handle all cases, two possibilities need to be considered.

a) If first character of C matches with first character of A, we move one character ahead in A and C and recursively check.

b) If first character of C matches with first character of B, we move one character ahead in B and C and recursively check.

If any of the above two cases is true, we return true, else false. Following is simple recursive implementation of this approach (Thanks to [Frederic]for suggesting this)

```
// A simple recursive function to check whether C is an interleaving of A and B

bool isInterleaved(char *A, char *B, char *C)

{

    // Base Case: If all strings are empty

    if (!(*A || *B || *C))

        return true;
```

```
    // If C is empty and any of the two strings is not empty
    if (*C == '\0')
        return false;

    // If any of the above mentioned two possibilities is true,
    // then return true, otherwise false
    return ( (*C == *A) && isInterleaved(A+1, B, C+1))
           || ((*C == *B) && isInterleaved(A, B+1, C+1));
}
```

Dynamic Programming

The worst case time complexity of recursive solution is $O(2^n)$. The above recursive solution certainly has many overlapping subproblems. For example, if wee consider A = “XXX”, B = “XXX” and C = “XXXXXX” and draw recursion tree, there will be many overlapping subproblems.
Therefore, like other typical Dynamic Programming problems, we can solve it by creating a table and store results of subproblems in bottom up manner. Thanks to Abhinav Ramana for suggesting this method and implementation.

```
// A Dynamic Programming based program to check whether a string C is
// an interleaving of two other strings A and B.
#include <iostream>
#include <string.h>
using namespace std;

// The main function that returns true if C is
// an interleaving of A and B, otherwise false.
bool isInterleaved(char* A, char* B, char* C)
{
    // Find lengths of the two strings
    int M = strlen(A), N = strlen(B);

    // Let us create a 2D table to store solutions of
    // subproblems.  C[i][j] will be true if C[0..i+j-1]
    // is an interleaving of A[0..i-1] and B[0..j-1].
    bool IL[M+1][N+1];
```

```
    memset(IL, 0, sizeof(IL)); // Initialize all values as false.

    // C can be an interleaving of A and B only of sum
    // of lengths of A & B is equal to length of C.
    if ((M+N) != strlen(C))
       return false;

    // Process all characters of A and B
    for (int i=0; i<=M; ++i)
    {
        for (int j=0; j<=N; ++j)
        {
            // two empty strings have an empty string
            // as interleaving
            if (i==0 && j==0)
                IL[i][j] = true;

            // A is empty
            else if (i==0 && B[j-1]==C[j-1])
                IL[i][j] = IL[i][j-1];

            // B is empty
            else if (j==0 && A[i-1]==C[i-1])
                IL[i][j] = IL[i-1][j];

            // Current character of C matches with current character of A,
            // but doesn't match with current character of B
            else if(A[i-1]==C[i+j-1] && B[j-1]!=C[i+j-1])
                IL[i][j] = IL[i-1][j];

            // Current character of C matches with current character of B,
            // but doesn't match with current character of A
            else if (A[i-1]!=C[i+j-1] && B[j-1]==C[i+j-1])
```

```
                IL[i][j] = IL[i][j-1];


            // Current character of C matches with that of both A and B
            else if (A[i-1]==C[i+j-1] && B[j-1]==C[i+j-1])
                IL[i][j]=(IL[i-1][j] || IL[i][j-1]) ;
        }
    }

    return IL[M][N];
}

// A function to run test cases
void test(char *A, char *B, char *C)
{
    if (isInterleaved(A, B, C))
        cout << C <<" is interleaved of " << A <<" and " << B << endl;
    else
        cout << C <<" is not interleaved of " << A <<" and " << B << endl;
}



// Driver program to test above functions
int main()
{
    test("XXY", "XXZ", "XXZXXXY");
    test("XY" ,"WZ" ,"WZXY");
    test ("XY", "X", "XXY");
    test ("YX", "X", "XXY");
    test ("XXY", "XXZ", "XXXXZY");
    return 0;
}
```

Output:

```
XXZXXXY is not interleaved of XXY and XXZ

WZXY is interleaved of XY and WZ

XXY is interleaved of XY and X

XXY is not interleaved of YX and X

XXXXZY is interleaved of XXY and XXZ
```

See this for more test cases.
Time Complexity: O(MN)
Auxiliary Space: O(MN)
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
++

Dynamic Programming | Set 32 (Word Break Problem)

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words. See following examples for more details.
This is a famous Google interview question, also being asked by many other companies now a days.

```
Consider the following dictionary

{ i, like, sam, sung, samsung, mobile, ice,

  cream, icecream, man, go, mango}


Input:  ilike

Output: Yes

The string can be segmented as "i like".


Input:  ilikesamsung

Output: Yes

The string can be segmented as "i like samsung" or "i like sam sung".
```

Recursive implementation:
The idea is simple, we consider each prefix and search it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix). If the recursive call for suffix returns true, we return true, otherwise we try next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.
We strongly recommend to see **substr** function which is used extensively in following implementations.

```
// A recursive program to test whether a given string can be segmented into

// space separated words in dictionary

#include <iostream>
```

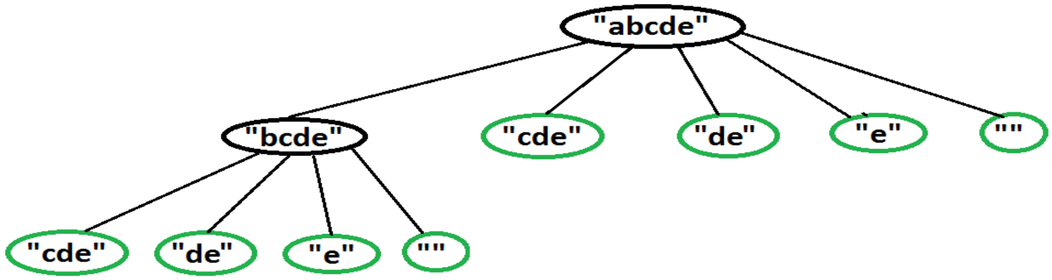
```
using namespace std;

/* A utility function to check whether a word is present in dictionary or not.
  An array of strings is used for dictionary.  Using array of strings for
  dictionary is definitely not a good idea. We have used for simplicity of
  the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile","samsung","sam","sung","man","mango",
                           "icecream","and","go","i","like","ice","cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
           return true;
    return false;
}

// returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
    int size = str.size();

    // Base case
    if (size == 0)  return true;

    // Try all prefixes of lengths from 1 to size
    for (int i=1; i<=size; i++)
    {
        // The parameter for dictionaryContains is str.substr(0, i)
        // str.substr(0, i) which is prefix (of input string) of
        // length 'i'. We first check whether current prefix is in
        // dictionary. Then we recursively check for remaining string
```

```
        // str.substr(i, size-i) which is suffix of length size-i
        if (dictionaryContains( str.substr(0, i) ) &&
            wordBreak( str.substr(i, size-i) ))
            return true;
    }

    // If we have tried all prefixes and none of them worked
    return false;
}

// Driver program to test above functions
int main()
{
    wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
    wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("")? cout <<"Yes\n": cout << "No\n";
    wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
    return 0;
}
```

Output:

```
Yes
Yes
Yes
Yes
Yes
No
```

Dynamic Programming
Why Dynamic Programming? The above problem exhibits overlapping sub-problems. For example, see the following partial recursion tree for string “abcde” in worst case.



Partial recursion tree for input string "abcde". The subproblems encircled with green color are overlapping subproblems

```
// A Dynamic Programming based program to test whether a given string can
// be segmented into space separated words in dictionary
#include <iostream>
#include <string.h>
using namespace std;

/* A utility function to check whether a word is present in dictionary or not.
  An array of strings is used for dictionary.  Using array of strings for
  dictionary is definitely not a good idea. We have used for simplicity of
  the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile","samsung","sam","sung","man","mango",
                           "icecream","and","go","i","like","ice","cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
           return true;
    return false;
}

// Returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
    int size = str.size();
```

```
    if (size == 0)   return true;

    // Create the DP table to store results of subroblems. The value wb[i]
    // will be true if str[0..i-1] can be segmented into dictionary words,
    // otherwise false.
    bool wb[size+1];
    memset(wb, 0, sizeof(wb)); // Initialize all values as false.

    for (int i=1; i<=size; i++)
    {
        // if wb[i] is false, then check if current prefix can make it true.
        // Current prefix is "str.substr(0, i)"
        if (wb[i] == false && dictionaryContains( str.substr(0, i) ))
            wb[i] = true;

        // wb[i] is true, then check for all substrings starting from
        // (i+1)th character and store their results.
        if (wb[i] == true)
        {
            // If we reached the last prefix
            if (i == size)
                return true;

            for (int j = i+1; j <= size; j++)
            {
                // Update wb[j] if it is false and can be updated
                // Note the parameter passed to dictionaryContains() is
                // substring starting from index 'i' and length 'j-i'
                if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))
                    wb[j] = true;

                // If we reached the last character
                if (j == size && wb[j] == true)
```

```
                    return true;
                }
            }
        }

        /* Uncomment these lines to print DP table "wb[]"
         for (int i = 1; i <= size; i++)
            cout << " " << wb[i]; */

        // If we have tried all prefixes and none of them worked
        return false;
    }

    // Driver program to test above functions
    int main()
    {
        wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
        wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";
        wordBreak("")? cout <<"Yes\n": cout << "No\n";
        wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
        wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
        wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
        return 0;
    }
```

Output:

```
Yes
Yes
Yes
Yes
Yes
No
```

Exercise:
The above solutions only finds out whether a given string can be segmented or not. Extend the above Dynamic Programming solution to print all possible partitions of input string. See extended recursive solution for reference.
Examples:

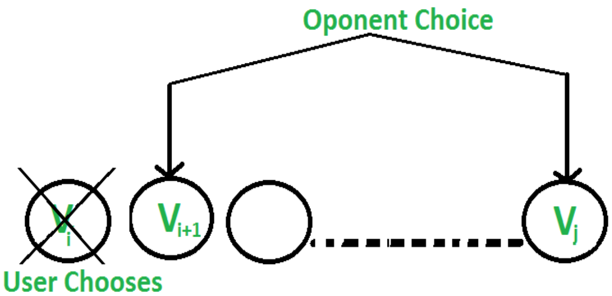
```
Input: ilikeicecreamandmango

Output:

i like ice cream and man go

i like ice cream and mango

i like icecream and man go

i like icecream and mango


Input: ilikesamsungmobile

Output:

i like sam sung mobile

i like samsung mobile
```

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

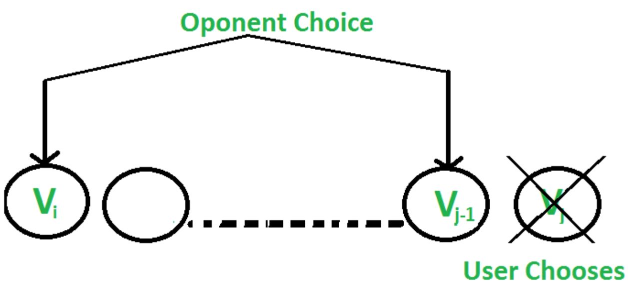
Dynamic Programming | Set 31 (Optimal Strategy for a Game)

Problem statement: Consider a row of n coins of values v1 . . . vn, where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.
Note: The opponent is as clever as the user.
Let us understand the problem with few examples:
Â Â Â Â **1.** 5, 3, 7, 10 : The user collects maximum value as 15(10 + 5)
Â Â Â Â **2.** 8, 15, 3, 7 : The user collects maximum value as 22(7 + 15)
Does choosing the best at each move give an optimal solution?
No. In the second example, this is how the game can finish:
1.
…….User chooses 8.
…….Opponent chooses 15.
…….User chooses 7.
…….Opponent chooses 3.
Total value collected by user is 15(8 + 7)
2.
…….User chooses 7.
…….Opponent chooses 8.
…….User chooses 15.
…….Opponent chooses 3.
Total value collected by user is 22(7 + 15)
So if the user follows the second game state, maximum value can be collected although the first move is not the best.

There are two choices:
1. The user chooses the ith coin with value Vi: The opponent either chooses (i+1)th coin or jth coin. The opponent intends to choose the coin which leaves the user with minimum value.
i.e. The user can collect the value Vi + min(F(i+2, j), F(i+1, j-1))



2. The user chooses the jth coin with value Vj: The opponent either chooses ith coin or (j-1)th coin. The opponent intends to choose the coin which leaves the user with minimum value.
i.e. The user can collect the value Vj + min(F(i+1, j-1), F(i, j-2))



Following is recursive solution that is based on above two choices. We take the maximum of two choices.

```
F(i, j)  represents the maximum value the user can collect from
         i'th coin to j'th coin.

    F(i, j)  = Max(Vi + min(F(i+2, j), F(i+1, j-1) ),
                   Vj + min(F(i+1, j-1), F(i, j-2) ))
Base Cases
    F(i, j)  = Vi           If j == i
    F(i, j)  = max(Vi, Vj)  If j == i+1
```

Why Dynamic Programming?
The above relation exhibits overlapping sub-problems. In the above relation, F(i+1, j-1) is calculated twice.

```
// C program to find out maximum value from a given sequence of coins
#include <stdio.h>
#include <limits.h>

// Utility functions to get maximum and minimum of two intgers
int max(int a, int b)  {    return a > b ? a : b;  }
```

```
int min(int a, int b)  {    return a < b ? a : b;  }


// Returns optimal value possible that a player can collect from

// an array of coins of size n. Note than n must be even

int optimalStrategyOfGame(int* arr, int n)

{

    // Create a table to store solutions of subproblems

    int table[n][n], gap, i, j, x, y, z;


    // Fill table using above recursive formula. Note that the table

    // is filled in diagonal fashion (similar to http://goo.gl/PQqoS),

    // from diagonal elements to table[0][n-1] which is the result.

    for (gap = 0; gap < n; ++gap)

    {

        for (i = 0, j = gap; j < n; ++i, ++j)

        {

            // Here x is value of F(i+2, j), y is F(i+1, j-1) and

            // z is F(i, j-2) in above recursive formula

            x = ((i+2) <= j) ? table[i+2][j] : 0;

            y = ((i+1) <= (j-1)) ? table[i+1][j-1] : 0;

            z = (i <= (j-2))? table[i][j-2]: 0;


            table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z));

        }

    }


    return table[0][n-1];

}


// Driver program to test above function

int main()

{

    int arr1[] = {8, 15, 3, 7};
```

```
    int n = sizeof(arr1)/sizeof(arr1[0]);
    printf("%d\n", optimalStrategyOfGame(arr1, n));

    int arr2[] = {2, 2, 2, 2};
    n = sizeof(arr2)/sizeof(arr2[0]);
    printf("%d\n", optimalStrategyOfGame(arr2, n));

    int arr3[] = {20, 30, 2, 2, 2, 10};
    n = sizeof(arr3)/sizeof(arr3[0]);
    printf("%d\n", optimalStrategyOfGame(arr3, n));

    return 0;
}
```

Output:

```
22
4
42
```

Exercise
Your thoughts on the strategy when the user wishes to only win instead of winning with the maximum value. Like above problem, number of coins is even.
Can Greedy approach work quite well and give an optimal solution? Will your answer change if number of coins is odd?
This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
++

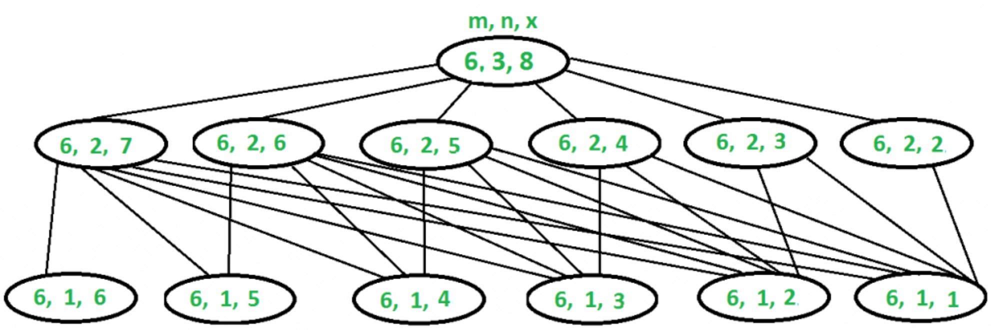
Dynamic Programming | Set 30 (Dice Throw)

Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown.
The **Naive approach** is to find all the possible combinations of values from n dice and keep on counting the results that sum to X.
This problem can be efficiently solved using **Dynamic Programming (DP)**.

```
Let the function to find X from n dice is: Sum(m, n, X)

The function can be represented as:

Sum(m, n, X) = Finding Sum (X - 1) from (n - 1) dice plus 1 from nth dice

               + Finding Sum (X - 2) from (n - 1) dice plus 2 from nth dice
```

```
               + Finding Sum (X - 3) from (n - 1) dice plus 3 from nth dice

                  ..................................................

                  ..................................................

                  ..................................................

              + Finding Sum (X - m) from (n - 1) dice plus m from nth dice


So we can recursively write Sum(m, n, x) as following

Sum(m, n, X) = Sum(m, n - 1, X - 1) +

               Sum(m, n - 1, X - 2) +

               .................... +

               Sum(m, n - 1, X - m)
```

Why DP approach?
The above problem exhibits overlapping subproblems. See the below diagram. Also, see [this](#)recursive implementation. Let there be 3 dice, each with 6 faces and we need to find the number of ways to get sum 8:




```
Sum(6, 3, 8) = Sum(6, 2, 7) + Sum(6, 2, 6) + Sum(6, 2, 5) +

               Sum(6, 2, 4) + Sum(6, 2, 3) + Sum(6, 2, 2)


To evaluate Sum(6, 3, 8), we need to evaluate Sum(6, 2, 7) which can

recursively written as following:

Sum(6, 2, 7) = Sum(6, 1, 6) + Sum(6, 1, 5) + Sum(6, 1, 4) +

               Sum(6, 1, 3) + Sum(6, 1, 2) + Sum(6, 1, 1)


We also need to evaluate Sum(6, 2, 6) which can recursively written

as following:

Sum(6, 2, 6) = Sum(6, 1, 5) + Sum(6, 1, 4) + Sum(6, 1, 3) +

               Sum(6, 1, 2) + Sum(6, 1, 1)

.............................................

.............................................
```

```
Sum(6, 2, 2) = Sum(6, 1, 1)
```

Please take a closer look at the above recursion. The sub-problems in RED are solved first time and sub-problems in BLUE are solved again (exhibit overlapping sub-problems). Hence, storing the results of the solved sub-problems saves time.
Following is C++ implementation of Dynamic Programming approach.

```
// C++ program to find number of ways to get sum 'x' with 'n'

// dice where every dice has 'm' faces

#include <iostream>

#include <string.h>

using namespace std;


// The main function that returns number of ways to get sum 'x'

// with 'n' dice and 'm' with m faces.

int findWays(int m, int n, int x)

{

    // Create a table to store results of subproblems.  One extra

    // row and column are used for simpilicity (Number of dice

    // is directly used as row index and sum is directly used

    // as column index).  The entries in 0th row and 0th column

    // are never used.

    int table[n + 1][x + 1];

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


    // Table entries for only one dice

    for (int j = 1; j <= m && j <= x; j++)

        table[1][j] = 1;


    // Fill rest of the entries in table using recursive relation

    // i: number of dice, j: sum

    for (int i = 2; i <= n; i++)

        for (int j = 1; j <= x; j++)

            for (int k = 1; k <= m && k < j; k++)

                table[i][j] += table[i-1][j-k];
```

```
    /* Uncomment these lines to see content of table
    for (int i = 0; i <= n; i++)
    {
      for (int j = 0; j <= x; j++)
        cout << table[i][j] << " ";
      cout << endl;
    } */
    return table[n][x];
}

// Driver program to test above functions
int main()
{
    cout << findWays(4, 2, 1) << endl;
    cout << findWays(2, 2, 3) << endl;
    cout << findWays(6, 3, 8) << endl;
    cout << findWays(4, 2, 5) << endl;
    cout << findWays(4, 3, 5) << endl;

    return 0;
}
```

Output:

```
0
2
21
4
6
```

Time Complexity: O(m * n * x) where m is number of faces, n is number of dice and x is given sum.
We can add following two conditions at the beginning of findWays() to improve performance of program for extreme cases (x is too high or x is too low)

```
    // When x is so high that sum can not go beyond x even when we
```

```
    // get maximum value in every dice throw.
    if (m*n <= x)
        return (m*n == x);

    // When x is too low
    if (n >= x)
        return (n == x);
```

With above conditions added, time complexity becomes O(1) when x >= m*n or when x <= n.
Exercise:
Extend the above algorithm to find the probability to get Sum > X.
This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
++

Dynamic Programming | Set 29 (Longest Common Substring)

Given two strings 'X' and 'Y', find the length of the longest common substring. For example, if the given strings are "GeeksforGeeks" and "GeeksQuiz", the output should be 5 as longest common substring is "Geeks"
Let m and n be the lengths of first and second strings respectively.
A **simple solution** is to one by one consider all substrings of first string and for every substring check if it is a substring in second string. Keep track of the maximum length substring. There will be O(m^2) substrings and we can find whether a string is subsring on another string in O(n) time (See this). So overall time complexity of this method would be $O(n * m^2)$
Dynamic Programming can be used to find the longest common substring in O(m*n) time. The idea is to find length of the longest common suffix for all substrings of both strings and store these lengths in a table.

```
The longest common suffix has following optimal substructure property
   LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1 if X[m-1] = Y[n-1]
                        0  Otherwise (if X[m-1] != Y[n-1])


The maximum length Longest Common Suffix is the longest common substring.
   LCSubStr(X, Y, m, n)  = Max(LCSuff(X, Y, i, j)) where 1 <= i <= m
                                                     and 1 <= j <= n
```

Following is C++ implementation of the above solution.

```
/* Dynamic Programming solution to find length of the longest common substring */
#include<iostream>
#include<string.h>
```

```
using namespace std;


// A utility function to find maximum of two integers

int max(int a, int b)

{   return (a > b)? a : b; }


/* Returns length of longest common substring of X[0..m-1] and Y[0..n-1] */

int LCSubStr(char *X, char *Y, int m, int n)

{

    // Create a table to store lengths of longest common suffixes of

    // substrings.   Notethat LCSuff[i][j] contains length of longest

    // common suffix of X[0..i-1] and Y[0..j-1]. The first row and

    // first column entries have no logical meaning, they are used only

    // for simplicity of program

    int LCSuff[m+1][n+1];

    int result = 0;  // To store length of the longest common substring


    /* Following steps build LCSuff[m+1][n+1] in bottom up fashion. */

    for (int i=0; i<=m; i++)

    {

        for (int j=0; j<=n; j++)

        {

            if (i == 0 || j == 0)

                LCSuff[i][j] = 0;


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

            {

                LCSuff[i][j] = LCSuff[i-1][j-1] + 1;

                result = max(result, LCSuff[i][j]);

            }

            else LCSuff[i][j] = 0;

        }

    }
```

```
    return result;
}

/* Driver program to test above function */
int main()
{
    char X[] = "OldSite:GeeksforGeeks.org";
    char Y[] = "NewSite:GeeksQuiz.com";

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

    cout << "Length of Longest Common Substring is " << LCSubStr(X, Y, m, n);
    return 0;
}
```

Output:

```
Length of Longest Common Substring is 10
```

Time Complexity: O(m*n)
Auxiliary Space: O(m*n)
References: http://en.wikipedia.org/wiki/Longest_common_substring_problem
The longest substring can also be solved in O(n+m) time using Suffix Tree. We will be covering Suffix Tree based solution in a separate post.
Exercise: The above solution prints only length of the longest common substring. Extend the solution to print the substring also.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
++

Dynamic Programming | Set 28 (Minimum insertions to form a palindrome)

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.
Before we go further, let us understand with few examples:
ab: Number of insertions required is 1. **b**ab
aa: Number of insertions required is 0. aa
abcd: Number of insertions required is 3. **dcb**abcd
abcda: Number of insertions required is 2. a**dc**bcda which is same as number of insertions in the substring bcd(Why?).
abcde: Number of insertions required is 4. **edcb**abcde
Let the input string be *str[lâ€¦â€¦h]*. The problem can be broken down into three parts:
1. Find the minimum number of insertions in the substring str[l+1,â€¦â€¦.h].
2. Find the minimum number of insertions in the substring str[lâ€¦â€¦.h-1].

3. Find the minimum number of insertions in the substring str[l+1â€¦â€¦h-1].

Recursive Solution

The minimum number of insertions in the string str[lâ€¦..h] can be given as:
minInsertions(str[l+1â€¦..h-1]) if str[l] is equal to str[h]
min(minInsertions(str[lâ€¦..h-1]), minInsertions(str[l+1â€¦..h])) + 1 otherwise

```
// A Naive recursive program to find minimum number insertions

// needed to make a string palindrome

#include <stdio.h>

#include <limits.h>

#include <string.h>


// A utility function to find minimum of two numbers

int min(int a, int b)

{  return a < b ? a : b; }


// Recursive function to find minimum number of insersions

int findMinInsertions(char str[], int l, int h)

{

    // Base Cases

    if (l > h) return INT_MAX;

    if (l == h) return 0;

    if (l == h - 1) return (str[l] == str[h])? 0 : 1;


    // Check if the first and last characters are same. On the basis of the

    // comparison result, decide which subrpoblem(s) to call

    return (str[l] == str[h])? findMinInsertions(str, l + 1, h - 1):

                               (min(findMinInsertions(str, l, h - 1),

                                   findMinInsertions(str, l + 1, h)) + 1);

}


// Driver program to test above functions

int main()

{

    char str[] = "geeks";
```

```
    printf("%d", findMinInsertions(str, 0, strlen(str)-1));

    return 0;

}
```

Output:

```
3
```

Dynamic Programming based Solution
If we observe the above approach carefully, we can find that it exhibits overlapping subproblems.
Suppose we want to find the minimum number of insertions in string “abcde”:

```
                      abcde

                /       |       \

               /        |         \

           bcde          abcd       bcd  <- case 3 is discarded as str[l] != str[h]

       /   |   \       /   |   \

      /    |    \     /    |    \

     cde   bcd  cd   bcd abc bc

   / | \  / | \ /|\ / | \

de cd d cd bc câ€¦â€¦â€¦â€¦â€¦â€¦â€¦.
```

The substrings in bold show that the recursion to be terminated and the recursion tree cannot originate from there. Substring in the same color indicates overlapping subproblems.
How to reuse solutions of subproblems?
We can create a table to store results of subproblems so that they can be used directly if same subproblem is encountered again.
The below table represents the stored values for the string abcde.

```
a b c d e

----------

0 1 2 3 4

0 0 1 2 3

0 0 0 1 2

0 0 0 0 1

0 0 0 0 0
```

How to fill the table?
The table should be filled in diagonal fashion. For the string abcde, 0â€¦.4, the following should be order in which the table is filled:

```
Gap = 1:

(0, 1) (1, 2) (2, 3) (3, 4)
```

```
Gap = 2:

(0, 2) (1, 3) (2, 4)


Gap = 3:

(0, 3) (1, 4)


Gap = 4:

(0, 4)

// A Dynamic Programming based program to find minimum number
// insertions needed to make a string palindrome
#include <stdio.h>
#include <string.h>

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

// A DP function to find minimum number of insersions
int findMinInsertionsDP(char str[], int n)
{
    // Create a table of size n*n. table[i][j] will store
    // minumum number of insertions needed to convert str[i..j]
    // to a palindrome.
    int table[n][n], l, h, gap;

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

    // Fill the table
    for (gap = 1; gap < n; ++gap)
        for (l = 0, h = gap; h < n; ++l, ++h)
            table[l][h] = (str[l] == str[h])? table[l+1][h-1] :
```

```
                    (min(table[l][h-1], table[l+1][h]) + 1);


    // Return minimum number of insertions for str[0..n-1]
    return table[0][n-1];
}


// Driver program to test above function.
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsDP(str, strlen(str)));
    return 0;
}
```

Output:

```
3
```

Time complexity: O(N^2)
Auxiliary Space: O(N^2)
Another Dynamic Programming Solution (Variation of [Longest Common Subsequence Problem)]()
The problem of finding minimum insertions can also be solved using Longest Common Subsequence (LCS) Problem. If we find out LCS of string and its reverse, we know how many maximum characters can form a palindrome. We need insert remaining characters. Following are the steps.
1) Find the length of LCS of input string and its reverse. Let the length be 'l'.
2) The minimum number insertions needed is length of input string minus 'l'.

```
// An LCS based program to find minimum number insertions needed to
// make a string palindrome
#include<stdio.h>
#include <string.h>


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


/* Returns length of LCS for X[0..m-1], Y[0..n-1].
```

```
   See http://goo.gl/bHQVP for details of this function */
int lcs( char *X, char *Y, int m, int n )
{
   int L[n+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];
}

// LCS based function to find minimum number of insersions
int findMinInsertionsLCS(char str[], int n)
{
   // Creata another string to store reverse of 'str'
   char rev[n+1];
   strcpy(rev, str);
```

```
    strrev(rev);

    // The output is length of string minus length of lcs of
    // str and it reverse
    return (n - lcs(str, rev, n, n));
}

// Driver program to test above functions
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsLCS(str, strlen(str)));
    return 0;
}
```

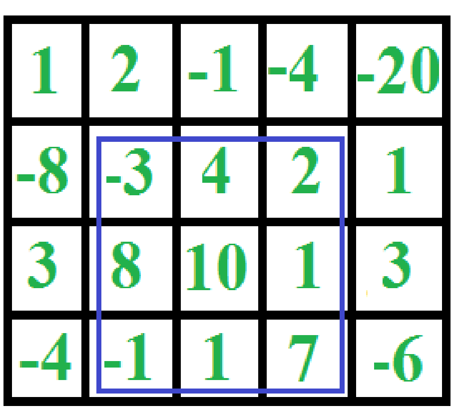
Output:

```
3
```

Time complexity of this method is also O(n^2) and this method also requires O(n^2) extra space.
This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
++

Dynamic Programming | Set 27 (Maximum sum rectangle in a 2D matrix)

Given a 2D array, find the maximum sum subarray in it. For example, in the following 2D array, the maximum sum subarray is highlighted with blue rectangle and sum of this subarray is 29.



This problem is mainly an extension of Largest Sum Contiguous Subarray for 1D array.
The **naive solution** for this problem is to check every possible rectangle in given 2D array. This solution requires 4 nested loops and time complexity of this solution would be O(n^4).

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.

```
// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
#define ROW 4
#define COL 5

// Implementation of Kadane's algorithm for 1D array. The function returns the
// maximum sum and stores starting and ending indexes of the maximum sum subarray
// at addresses pointed by start and finish pointers respectively.
int kadane(int* arr, int* start, int* finish, int n)
{
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i)
    {
        sum += arr[i];
        if (sum < 0)
        {
            sum = 0;
            local_start = i+1;
```

```
        }
        else if (sum > maxSum)
        {
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }

     // There is at-least one non-negative number
    if (*finish != -1)
        return maxSum;

    // Special Case: When all numbers in arr[] are negative
    maxSum = arr[0];
    *start = *finish = 0;

    // Find the maximum element in array
    for (i = 1; i < n; i++)
    {
        if (arr[i] > maxSum)
        {
            maxSum = arr[i];
            *start = *finish = i;
        }
    }
    return maxSum;
}

// The main function that finds maximum sum rectangle in M[][]
void findMaxSum(int M[][COL])
{
    // Variables to store the final output
```

```
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left)
    {
        // Initialize all elements of temp as 0
        memset(temp, 0, sizeof(temp));

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right)
        {
            // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += M[i][right];

            // Find the maximum sum subarray in temp[]. The kadane() function
            // also sets values of start and finish.  So 'sum' is sum of
            // rectangle between (start, left) and (finish, right) which is the
            //  maximum sum with boundary columns strictly as left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then update
            // maxSum and other output values
            if (sum > maxSum)
            {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
```

```
            }

        }

    }


    // Print final values

    printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);

    printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);

    printf("Max sum is: %d\n", maxSum);

}


// Driver program to test above functions

int main()

{

    int M[ROW][COL] = {{1, 2, -1, -4, -20},

                       {-8, -3, 4, 2, 1},

                       {3, 8, 10, 1, 3},

                       {-4, -1, 1, 7, -6}

                      };


    findMaxSum(M);


    return 0;

}
```

Output:

```
(Top, Left) (1, 1)

(Bottom, Right) (3, 3)

Max sum is: 29
```

Time Complexity: O(n^3)
This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Program for nth Catalan Number

Catalan numbers are a sequence of natural numbers that occurs in many interesting counting problems like following.
1) Count the number of expressions containing n pairs of parentheses which are correctly matched. For n = 3, possible expressions are ((())), ()(()), ()()(), (())(), (()()).
2) Count the number of possible Binary Search Trees with n keys (See this)
3) Count the number of full binary trees (A rooted binary tree is full if every vertex has either two children or no children) with n+1 leaves.
See this for more applications.
The first few Catalan numbers for n = 0, 1, 2, 3, â€¦ are **1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …**
Recursive Solution
Catalan numbers satisfy the following recursive formula.
$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i \, C_{n-i} \quad \text{for } n \geq 0;$
Following is C++ implementation of above recursive formula.

```
#include<iostream>

using namespace std;


// A recursive function to find nth catalan number

unsigned long int catalan(unsigned int n)

{

    // Base case

    if (n <= 1) return 1;


    // catalan(n) is sum of catalan(i)*catalan(n-i-1)

    unsigned long int res = 0;

    for (int i=0; i<n; i++)

        res += catalan(i)*catalan(n-i-1);


    return res;

}


// Driver program to test above function

int main()

{

    for (int i=0; i<10; i++)

        cout << catalan(i) << " ";

    return 0;

}
```

Output :

```
1 1 2 5 14 42 132 429 1430 4862
```

Time complexity of above implementation is equivalent to nth catalan number.
$T(n) = \sum_{i=0}^{n-1} T(i) * T(n-i) \quad \text{for } n \geq 0;$
The value of nth catalan number is exponential that makes the time complexity exponential.

Dynamic Programming Solution
We can observe that the above recursive implementation does a lot of repeated work (we can the same by drawing recursion tree). Since there are overlapping subproblems, we can use dynamic programming for this. Following is a Dynamic programming based implementation in C++.

```
#include<iostream>

using namespace std;


// A dynamic programming based function to find nth

// Catalan number

unsigned long int catalanDP(unsigned int n)

{

    // Table to store results of subproblems

    unsigned long int catalan[n+1];


    // Initialize first two values in table

    catalan[0] = catalan[1] = 1;


    // Fill entries in catalan[] using recursive formula

    for (int i=2; i<=n; i++)

    {

        catalan[i] = 0;

        for (int j=0; j<i; j++)

            catalan[i] += catalan[j] * catalan[i-j-1];

    }


    // Return last entry

    return catalan[n];

}
```

```
// Driver program to test above function

int main()

{

    for (int i = 0; i < 10; i++)

        cout << catalanDP(i) << " ";

    return 0;

}
```

Output:

```
1 1 2 5 14 42 132 429 1430 4862
```

Time Complexity: Time complexity of above implementation is $O(n^2)$

Using Binomial Coefficient
We can also use the below formula to find nth catalan number in O(n) time.
$C_n = \frac{1}{n+1}\binom{2n}{n}$
We have discussed a O(n) approach to find binomial coefficient nCr.

```
#include<iostream>

using namespace std;


// Returns value of Binomial Coefficient C(n, k)

unsigned long int binomialCoeff(unsigned int n, unsigned int k)

{

    unsigned long int res = 1;


    // Since C(n, k) = C(n, n-k)

    if (k > n - k)

        k = n - k;


    // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]

    for (int i = 0; i < k; ++i)

    {

        res *= (n - i);
```

```
        res /= (i + 1);

    }


    return res;

}


// A Binomial coefficient based function to find nth catalan

// number in O(n) time

unsigned long int catalan(unsigned int n)

{

    // Calculate value of 2nCn

    unsigned long int c = binomialCoeff(2*n, n);


    // return 2nCn/(n+1)

    return c/(n+1);

}


// Driver program to test above functions

int main()

{

    for (int i = 0; i < 10; i++)

        cout << catalan(i) << " ";

    return 0;

}
```

Output:

```
1 1 2 5 14 42 132 429 1430 4862
```

Time Complexity: Time complexity of above implementation is O(n).
We can also use below formula to find nth catalan number in O(n) time.

$$C_n = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \qquad \text{for } n \ge 0.$$

References:

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

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

++

Remove minimum elements from either side such that 2*min becomes more than max

Given an unsorted array, trim the array such that twice of minimum is greater than maximum in the trimmed array. Elements should be removed either end of the array.
Number of removals should be minimum.
Examples:

```
arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200}
Output: 4
We need to remove 4 elements (4, 5, 100, 200)
so that 2*min becomes more than max.



arr[] = {4, 7, 5, 6}
Output: 0
We don't need to remove any element as
4*2 > 7 (Note that min = 4, max = 8)

arr[] = {20, 7, 5, 6}
Output: 1
We need to remove 20 so that 2*min becomes
more than max

arr[] = {20, 4, 1, 3}
Output: 3
We need to remove any three elements from ends
like 20, 4, 1 or 4, 1, 3 or 20, 3, 1 or 20, 4, 1
```

Naive Solution:
A naive solution is to try every possible case using recurrence. Following is the naive recursive algorithm. Note that the algorithm only returns minimum numbers of removals to be made, it doesn't print the trimmed array. It can be easily modified to print the trimmed array as well.

```
// Returns minimum number of removals to be made in
// arr[l..h]
```

```
minRemovals(int arr[], int l, int h)

1) Find min and max in arr[l..h]

2) If 2*min > max, then return 0.

3) Else return minimum of "minRemovals(arr, l+1, h) + 1"

   and "minRemovals(arr, l, h-1) + 1"
```

Following is C++ implementation of above algorithm.

```
#include <iostream>

using namespace std;


// A utility function to find minimum of two numbers

int min(int a, int b) {return (a < b)? a : b;}


// A utility function to find minimum in arr[l..h]

int min(int arr[], int l, int h)

{

    int mn = arr[l];

    for (int i=l+1; i<=h; i++)

       if (mn > arr[i])

         mn = arr[i];

    return mn;

}


// A utility function to find maximum in arr[l..h]

int max(int arr[], int l, int h)

{

    int mx = arr[l];

    for (int i=l+1; i<=h; i++)

       if (mx < arr[i])

         mx = arr[i];

    return mx;

}
```

```
// Returns the minimum number of removals from either end
// in arr[l..h] so that 2*min becomes greater than max.
int minRemovals(int arr[], int l, int h)
{
    // If there is 1 or less elements, return 0
    // For a single element, 2*min > max
    // (Assumption: All elements are positive in arr[])
    if (l >= h) return 0;

    // 1) Find minimum and maximum in arr[l..h]
    int mn = min(arr, l, h);
    int mx = max(arr, l, h);

    //If the property is followed, no removals needed
    if (2*mn > mx)
       return 0;

    // Otherwise remove a character from left end and recur,
    // then remove a character from right end and recur, take
    // the minimum of two is returned
    return min(minRemovals(arr, l+1, h),
               minRemovals(arr, l, h-1)) + 1;
}


// Driver program to test above functions
int main()
{
  int arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200};
  int n = sizeof(arr)/sizeof(arr[0]);
  cout << minRemovals(arr, 0, n-1);
  return 0;
}
```

Output:

```
4
```

Time complexity: Time complexity of the above function can be written as following

```
T(n) = 2T(n-1) + O(n)
```

An upper bound on solution of above recurrence would be $O(n \times 2^n)$.

Dynamic Programming:

The above recursive code exhibits many overlapping subproblems. For example minRemovals(arr, l+1, h-1) is evaluated twice. So Dynamic Programming is the choice to optimize the solution. Following is Dynamic Programming based solution.

```
#include <iostream>

using namespace std;


// A utility function to find minimum of two numbers

int min(int a, int b) {return (a < b)? a : b;}


// A utility function to find minimum in arr[l..h]

int min(int arr[], int l, int h)

{

    int mn = arr[l];

    for (int i=l+1; i<=h; i++)

       if (mn > arr[i])

         mn = arr[i];

    return mn;

}


// A utility function to find maximum in arr[l..h]

int max(int arr[], int l, int h)

{

    int mx = arr[l];

    for (int i=l+1; i<=h; i++)

       if (mx < arr[i])

         mx = arr[i];

    return mx;
```

```
}


// Returns the minimum number of removals from either end
// in arr[l..h] so that 2*min becomes greater than max.
int minRemovalsDP(int arr[], int n)
{
    // Create a table to store solutions of subproblems
    int table[n][n], gap, i, j, mn, mx;

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion (similar to http://goo.gl/PQqoS),
    // from diagonal elements to table[0][n-1] which is the result.
    for (gap = 0; gap < n; ++gap)
    {
        for (i = 0, j = gap; j < n; ++i, ++j)
        {
            mn = min(arr, i, j);
            mx = max(arr, i, j);
            table[i][j] = (2*mn > mx)? 0: min(table[i][j-1]+1,
                                              table[i+1][j]+1);
        }
    }
    return table[0][n-1];
}


// Driver program to test above functions
int main()
{
  int arr[] = {20, 4, 1, 3};
  int n = sizeof(arr)/sizeof(arr[0]);
  cout << minRemovalsDP(arr, n);
  return 0;
}
```

Time Complexity: $O(n^3)$ where n is the number of elements in arr[].
Further Optimizations:
The above code can be optimized in many ways.
1) We can avoid calculation of min() and/or max() when min and/or max is/are not changed by removing corner elements.
2) We can pre-process the array and build segment tree in O(n) time. After the segment tree is built, we can query range minimum and maximum in O(Logn) time. The overall time complexity is reduced to $O(n^2Logn)$ time.
A O(n^2) Solution
The idea is to find the maximum sized subarray such that 2*min > max. We run two nested loops, the outer loop chooses a starting point and the inner loop chooses ending point for the current starting point. We keep track of longest subarray with the given property.
Following is C++ implementation of the above approach. Thanks to Richard Zhang for suggesting this solution.

```
// A O(n*n) solution to find the minimum of elements to

// be removed

#include <iostream>

#include <climits>

using namespace std;


// Returns the minimum number of removals from either end

// in arr[l..h] so that 2*min becomes greater than max.

int minRemovalsDP(int arr[], int n)

{

    // Initialize starting and ending indexes of the maximum

    // sized subarray with property 2*min > max

    int longest_start = -1, longest_end = 0;


    // Choose different elements as starting point

    for (int start=0; start<n; start++)

    {

        // Initialize min and max for the current start

        int min = INT_MAX, max = INT_MIN;


        // Choose different ending points for current start

        for (int end = start; end < n; end ++)

        {

            // Update min and max if necessary

            int val = arr[end];
```

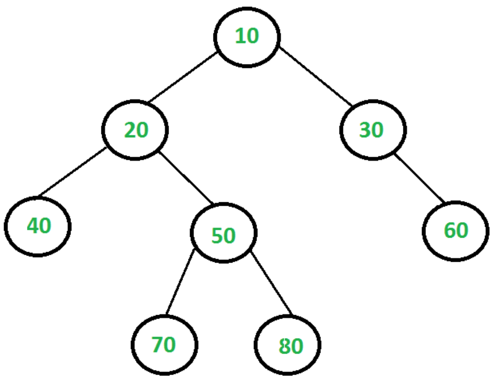
```
            if (val < min) min = val;
            if (val > max) max = val;

            // If the property is violated, then no
            // point to continue for a bigger array
            if (2 * min <= max) break;

            // Update longest_start and longest_end if needed
            if (end - start > longest_end - longest_start ||
                longest_start == -1)
            {
                longest_start = start;
                longest_end = end;
            }
        }
    }

    // If not even a single element follow the property,
    // then return n
    if (longest_start == -1) return n;

    // Return the number of elements to be removed
    return (n - (longest_end - longest_start + 1));
}

// Driver program to test above functions
int main()
{
    int arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << minRemovalsDP(arr, n);
    return 0;
}
```

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

++

Dynamic Programming | Set 26 (Largest Independent Set Problem)

Given a Binary Tree, find size of the **L**argest **I**ndependent **S**et(LIS) in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.
For example, consider the following binary tree. The largest independent set(LIS) is {10, 40, 60, 70, 80} and size of the LIS is 5.



A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

1) Optimal Substructure:

Let LISS(X) indicates size of largest independent set of a tree with root X.

```
     LISS(X) = MAX { (1 + sum of LISS for all grandchildren of X),

                     (sum of LISS for all children of X) }
```

The idea is simple, there are two possibilities for every node X, either X is a member of the set or not a member. If X is a member, then the value of LISS(X) is 1 plus LISS of all grandchildren. If X is not a member, then the value is sum of LISS of all children.

2) Overlapping Subproblems

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

```
// A naive recursive implementation of Largest Independent Set problem

#include <stdio.h>

#include <stdlib.h>


// A utility function to find max of two integers

int max(int x, int y) { return (x > y)? x: y; }


/* A binary tree node has data, pointer to left child and a pointer to
```

```
   right child */
struct node
{
    int data;
    struct node *left, *right;
};

// The function returns size of the largest independent set in a given
// binary tree
int LISS(struct node *root)
{
    if (root == NULL)
       return 0;

    // Caculate size excluding the current node
    int size_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int size_incl = 1;
    if (root->left)
       size_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
       size_incl += LISS(root->right->left) + LISS(root->right->right);

    // Return the maximum of two sizes
    return max(size_incl, size_excl);
}


// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
```

```
    temp->data = data;

    temp->left = temp->right = NULL;

    return temp;

}


// Driver program to test above functions

int main()

{

    // Let us construct the tree given in the above diagram

    struct node *root         = newNode(20);

    root->left                = newNode(8);

    root->left->left          = newNode(4);

    root->left->right         = newNode(12);

    root->left->right->left   = newNode(10);

    root->left->right->right  = newNode(14);

    root->right               = newNode(22);

    root->right->right        = newNode(25);


    printf ("Size of the Largest Independent Set is %d ", LISS(root));


    return 0;

}
```

Output:

```
Size of the Largest Independent Set is 5
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, LISS of node with value 50 is evaluated for node with values 10 and 20 as 50 is grandchild of 10 and child of 20.
Since same suproblems are called again, this problem has Overlapping Subprolems property. So LISS problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.
Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field 'liss' is added to tree nodes. The initial value of 'liss' is set as 0 for all nodes. The recursive function LISS() calculates 'liss' for a node only if it is not already set.

```
/* Dynamic programming based program for Largest Independent Set problem */
```

```
#include <stdio.h>

#include <stdlib.h>


// A utility function to find max of two integers

int max(int x, int y) { return (x > y)? x: y; }


/* A binary tree node has data, pointer to left child and a pointer to

   right child */

struct node

{

    int data;

    int liss;

    struct node *left, *right;

};


// A memoization function returns size of the largest independent set in

//  a given binary tree

int LISS(struct node *root)

{

    if (root == NULL)

        return 0;


    if (root->liss)

        return root->liss;


    if (root->left == NULL && root->right == NULL)

        return (root->liss = 1);


    // 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;
}

// A utility function to create a node
struct node* newNode(int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->liss = 0;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root         = newNode(20);
    root->left                = newNode(8);
    root->left->left          = newNode(4);
    root->left->right         = newNode(12);
    root->left->right->left   = newNode(10);
    root->left->right->right  = newNode(14);
    root->right               = newNode(22);
    root->right->right        = newNode(25);
```

```
    printf ("Size of the Largest Independent Set is %d ", LISS(root));


    return 0;
}
```

Output

```
Size of the Largest Independent Set is 5
```

Time Complexity: O(n) where n is the number of nodes in given Binary tree.
Following extensions to above solution can be tried as an exercise.
1) Extend the above solution for n-ary tree.
2) The above solution modifies the given tree structure by adding an additional field 'liss' to tree nodes. Extend the solution so that it doesn't modify the tree structure.
3) The above solution only returns size of LIS, it doesn't print elements of LIS. Extend the solution to print all nodes that are part of LIS.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 25 (Subset Sum Problem)

Given a set of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.

```
Examples: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output:  True  //There is a subset (4, 5) with sum 9.
```

Let isSubSetSum(int set[], int n, int sum) be the function to find whether there is a subset of set[] with sum equal to *sum*. n is the number of elements in set[].
The isSubsetSum problem can be divided into two subproblems
…a) Include the last element, recur for n = n-1, sum = sum – set[n-1]
…b) Exclude the last element, recur for n = n-1.
If any of the above the above subproblems return true, then return true.
Following is the recursive formula for isSubsetSum() problem.

```
isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) ||
                           isSubsetSum(arr, n-1, sum-set[n-1])
Base Cases:
isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0
isSubsetSum(set, n, sum) = true, if sum == 0
```

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

```
// A recursive solution for subset sum problem
#include <stdio.h>

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

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

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

// Driver program to test above function
int main()
{
  int set[] = {3, 34, 4, 12, 5, 2};
  int sum = 9;
  int n = sizeof(set)/sizeof(set[0]);
  if (isSubsetSum(set, n, sum) == true)
     printf("Found a subset with given sum");
  else
     printf("No subset with given sum");
  return 0;
```

```
}
```

Output:

```
 Found a subset with given sum
```

The above solution may try all subsets of given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).
We can solve the problem in Pseudo-polynomial time using Dynamic programming. We create a boolean 2D table subset[][] and fill it in bottom up manner. The value of subset[i][j] will be true if there is a subset of set[0..j-1] with sum equal to i., otherwise false. Finally, we return subset[sum][n]

```
// A Dynamic Programming solution for subset sum problem

#include <stdio.h>


// Returns true if there is a subset of set[] with sun equal to given sum

bool isSubsetSum(int set[], int n, int sum)

{

    // The value of subset[i][j] will be true if there is a subset of set[0..j-1]

    //  with sum equal to i

    bool subset[sum+1][n+1];


    // If sum is 0, then answer is true

    for (int i = 0; i <= n; i++)

      subset[0][i] = true;


    // If sum is not 0 and set is empty, then answer is false

    for (int i = 1; i <= sum; i++)

      subset[i][0] = false;


     // Fill the subset table in botton up manner

     for (int i = 1; i <= sum; i++)

     {

       for (int j = 1; j <= n; j++)

       {

         subset[i][j] = subset[i][j-1];
```

```
          if (i >= set[j-1])
            subset[i][j] = subset[i][j] || subset[i - set[j-1]][j-1];
        }
      }

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

      return subset[sum][n];
}

// Driver program to test above function
int main()
{
  int set[] = {3, 34, 4, 12, 5, 2};
  int sum = 9;
  int n = sizeof(set)/sizeof(set[0]);
  if (isSubsetSum(set, n, sum) == true)
     printf("Found a subset with given sum");
  else
     printf("No subset with given sum");
  return 0;
}
```

Output:

```
Found a subset with given sum
```

Time complexity of the above solution is O(sum*n).
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 24 (Optimal Binary Search Tree)

Given a sorted array *keys[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches to *keys[i]*. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.
Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

```
Example 1

Input:  keys[] = {10, 12}, freq[] = {34, 50}

There can be following two possible BSTs

        10                       12

          \                     /

           12                 10

          I                     II

Frequency of searches of 10 and 12 are 34 and 50 respectively.

The cost of tree I is 34*1 + 50*2 = 134

The cost of tree II is 50*1 + 34*2 = 118


Example 2

Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs

    10                12                  20         10              20

      \             /    \              /              \            /

      12          10     20           12               20         10

        \                           /                 /             \

         20                       10                12               12

     I               II             III             IV             V

Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142
```

1) Optimal Substructure:
The optimal cost for freq[i..j] can be recursively calculated using following formula.

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

We need to calculate ***optCost(0, n-1)*** to find the result.
The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make *rth* node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.

We add sum of frequencies from i to j (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

2) Overlapping Subproblems

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

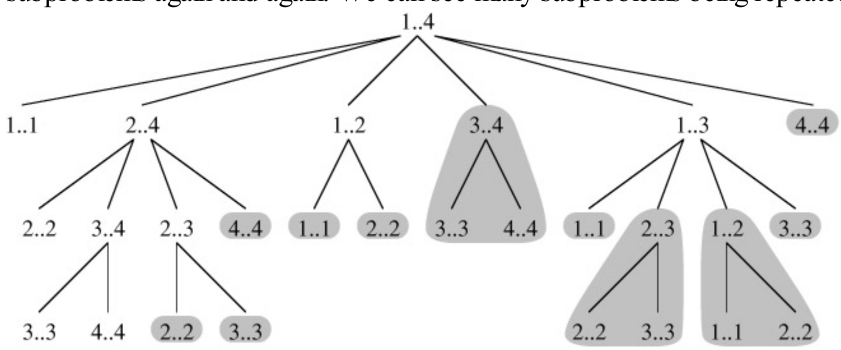
```
// A naive recursive implementation of optimal binary search tree problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal binary search tree
int optCost(int freq[], int i, int j)
{
   // Base cases
   if (j < i)      // If there are no elements in this subarray
     return 0;
   if (j == i)     // If there is one element in this subarray
     return freq[i];

   // Get sum of freq[i], freq[i+1], ... freq[j]
   int fsum = sum(freq, i, j);

   // Initialize minimum value
   int min = INT_MAX;

   // One by one consider all elements as root and recursively find cost
   // of the BST, compare the cost with min and update min if needed
   for (int r = i; r <= j; ++r)
   {
       int cost = optCost(freq, i, r-1) + optCost(freq, r+1, j);
       if (cost < min)
          min = cost;
   }
```

```
    // Return minimum value
    return min + fsum;
}

// The main function that calculates minimum cost of a Binary Search Tree.
// It mainly uses optCost() to find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
     // Here array keys[] is assumed to be sorted in increasing order.
     // If keys[] is not sorted, then add code to sort keys, and rearrange
     // freq[] accordingly.
     return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

```
Cost of Optimal BST is 142
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for freq[1..4].



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

Dynamic Programming Solution

Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values. So how to fill the 2D array in such manner> The idea used in the implementation is same as Matrix Chain Multiplication problem, we use a variable 'L' for chain length and increment 'L', one by one. We calculate column number 'j' using the values of 'i' and 'L'.

```
// Dynamic Programming code for Optimal Binary Search Tree Problem

#include <stdio.h>

#include <limits.h>


// A utility function to get sum of array elements freq[i] to freq[j]

int sum(int freq[], int i, int j);


/* A Dynamic Programming based function that calculates minimum cost of

   a Binary Search Tree. */

int optimalSearchTree(int keys[], int freq[], int n)

{

    /* Create an auxiliary 2D matrix to store results of subproblems */

    int cost[n][n];


    /* cost[i][j] = Optimal cost of binary search tree that can be

       formed from keys[i] to keys[j].
```

```
       cost[0][n-1] will store the resultant cost */

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ... .
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[][]
        for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and chain length L
            int j = i+L-1;
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
               // c = cost when keys[r] becomes root of this subtree
               int c = ((r > i)? cost[i][r-1]:0) +
                       ((r < j)? cost[r+1][j]:0) +
                       sum(freq, i, j);
               if (c < cost[i][j])
                  cost[i][j] = c;
            }
        }
    }
    return cost[0][n-1];
}

// A utility function to get sum of array elements freq[i] to freq[j]
```

```
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
       s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

```
Cost of Optimal BST is 142
```

Notes
1) The time complexity of the above solution is O(n^4). The time complexity can be easily reduced to O(n^3) by pre-calculating sum of frequencies instead of calling sum() again and again.
2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen 'r' in the innermost loop.

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

Dynamic Programming | Set 23 (Bellmanâ€“Ford Algorithm)

Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed [Dijkstra's algorithm]() for this problem. Dijksra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time*

complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

Algorithm

Following are the detailed steps.

Input: Graph and a source vertex *src*

Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

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

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

…..**a)** Do following for each edge u-v

………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]

………………….dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

……If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

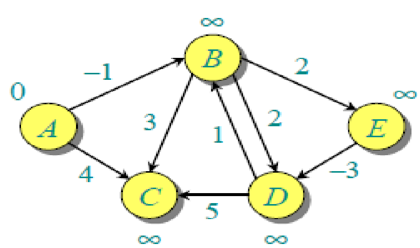
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

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 (Proof is simple, you can refer this or MIT Video Lecture)

Example

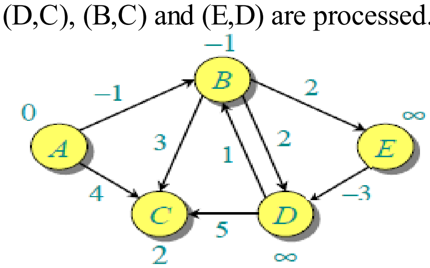
Let us understand the algorithm with following example graph. The images are taken from this source.

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*



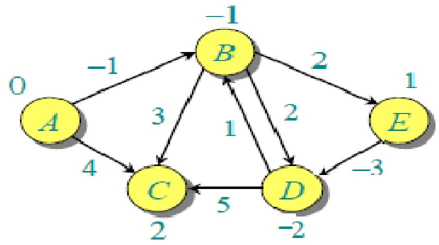
A	B	C	D	E
0	∞	∞	∞	∞

Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Implementation:

```
// A C / C++ program for Bellman-Ford's single source shortest path algorithm.


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>


// a structure to represent a weighted edge in graph

struct Edge

{

    int src, dest, weight;

};


// a structure to represent a connected, directed and weighted graph

struct Graph

{

    // V-> Number of vertices, E-> Number of edges

    int V, E;


    // graph is represented as an array of edges.

    struct Edge* edge;

};


// Creates a graph with V vertices and E edges

struct Graph* createGraph(int V, int E)

{
```

```
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to all other
// vertices using Bellman-Ford algorithm.  The function also detects negative
// weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices as INFINITE
    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest path from src
    // to any other vertex can have at-most |V| - 1 edges
```

```
    for (int i = 1; i <= V-1; i++)

    {

        for (int j = 0; j < E; j++)

        {

            int u = graph->edge[j].src;

            int v = graph->edge[j].dest;

            int weight = graph->edge[j].weight;

            if (dist[u] + weight < dist[v])

                dist[v] = dist[u] + weight;

        }

    }


    // Step 3: check for negative-weight cycles.  The above step guarantees

    // shortest distances if graph doesn't contain negative weight cycle.

    // If we get a shorter path, then there is a cycle.

    for (int i = 0; i < E; i++)

    {

        int u = graph->edge[i].src;

        int v = graph->edge[i].dest;

        int weight = graph->edge[i].weight;

        if (dist[u] + weight < dist[v])

            printf("Graph contains negative weight cycle");

    }


    printArr(dist, V);


    return;

}


// Driver program to test above functions

int main()

{

    /* Let us create the graph given in above example */
```

```
    int V = 5;  // Number of vertices in graph

    int E = 8;  // Number of edges in graph

    struct Graph* graph = createGraph(V, E);


    // add edge 0-1 (or A-B in above figure)

    graph->edge[0].src = 0;

    graph->edge[0].dest = 1;

    graph->edge[0].weight = -1;


    // add edge 0-2 (or A-C in above figure)

    graph->edge[1].src = 0;

    graph->edge[1].dest = 2;

    graph->edge[1].weight = 4;


    // add edge 1-2 (or B-C in above figure)

    graph->edge[2].src = 1;

    graph->edge[2].dest = 2;

    graph->edge[2].weight = 3;


    // add edge 1-3 (or B-D in above figure)

    graph->edge[3].src = 1;

    graph->edge[3].dest = 3;

    graph->edge[3].weight = 2;


    // add edge 1-4 (or A-E in above figure)

    graph->edge[4].src = 1;

    graph->edge[4].dest = 4;

    graph->edge[4].weight = 2;


    // add edge 3-2 (or D-C in above figure)

    graph->edge[5].src = 3;

    graph->edge[5].dest = 2;

    graph->edge[5].weight = 5;
```

```
    // add edge 3-1 (or D-B in above figure)

    graph->edge[6].src = 3;

    graph->edge[6].dest = 1;

    graph->edge[6].weight = 1;


    // add edge 4-3 (or E-D in above figure)

    graph->edge[7].src = 4;

    graph->edge[7].dest = 3;

    graph->edge[7].weight = -3;


    BellmanFord(graph, 0);


    return 0;

}
```

Output:

```
Vertex   Distance from Source

0                0

1                -1

2                2

3                -2

4                1
```

Notes
1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
2) Bellman-Ford works better (better than Dijksra's) for distributed systems. Unlike Dijksra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.
Exercise
1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.
2) Can we use Dijksra's algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijksra's algorithm for the modified graph. Will this algorithm work?
References:
http://www.youtube.com/watch?v=Ttezuzs39nk
http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf

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

Longest Palindromic Substring | Set 1

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

Method 1 (Brute Force)

The simple approach is to check each substring whether the substring is a palindrome or not. We can run three loops, the outer two loops pick all substrings one by one by fixing the corner characters, the inner loop checks whether the picked substring is palindrome or not.

Time complexity: O (n^3)

Auxiliary complexity: O (1)

Method 2 (Dynamic Programming)

The time complexity can be reduced by storing results of subproblems. The idea is similar to [this]post. We maintain a boolean table[n][n] that is filled in bottom up manner. The value of table[i][j] is true, if the substring is palindrome, otherwise false. To calculate table[i][j], we first check the value of table[i+1][j-1], if the value is true and str[i] is same as str[j], then we make table[i][j] true. Otherwise, the value of table[i][j] is made false.

```
// A dynamic programming solution for longest palindr.

// This code is adopted from following link

// http://www.leetcode.com/2011/11/longest-palindromic-substring-part-i.html


#include <stdio.h>

#include <string.h>


// A utility function to print a substring str[low..high]

void printSubStr( char* str, int low, int high )

{

    for( int i = low; i <= high; ++i )

        printf("%c", str[i]);

}


// This function prints the longest palindrome substring of str[].

// It also returns the length of the longest palindrome

int longestPalSubstr( char *str )

{

    int n = strlen( str ); // get length of input string
```

```
    // table[i][j] will be false if substring str[i..j] is not palindrome.

    // Else table[i][j] will be true

    bool table[n][n];

    memset( table, 0, sizeof( table ) );


    // All substrings of length 1 are palindromes

    int maxLength = 1;

    for( int i = 0; i < n; ++i )

        table[i][i] = true;


    // check for sub-string of length 2.

    int start = 0;

    for( int i = 0; i < n-1; ++i )

    {

        if( str[i] == str[i+1] )

        {

            table[i][i+1] = true;

            start = i;

            maxLength = 2;

        }

    }


    // Check for lengths greater than 2. k is length of substring

    for( int k = 3; k <= n; ++k )

    {

        // Fix the starting index

        for( int i = 0; i < n - k + 1 ; ++i )

        {

            // Get the ending index of substring from starting index i and length k

            int j = i + k - 1;


            // checking for sub-string from ith index to jth index iff str[i+1]

            // to str[j-1] is a palindrome
```

```
            if( table[i+1][j-1] && str[i] == str[j] )
            {
                table[i][j] = true;

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

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

    return maxLength; // return length of LPS
}

// Driver program to test above functions
int main()
{
    char str[] = "forgeeksskeegfor";
    printf("\nLength is: %d\n", longestPalSubstr( str ) );
    return 0;
}
```

Output:

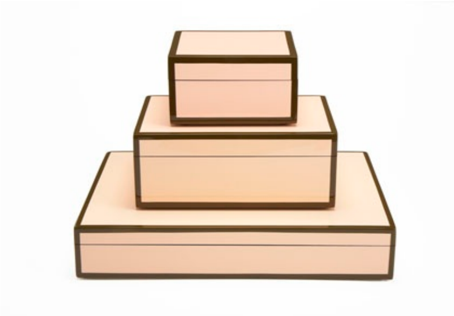
```
Longest palindrome substring is: geeksskeeg
Length is: 10
```

Time complexity: O (n^2)
Auxiliary Space: O (n^2)
We will soon be adding more optimized methods as separate posts.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

++

Dynamic Programming | Set 22 (Box Stacking Problem)

You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.
Source: http://people.csail.mit.edu/bdean/6.046/dp/. The link also has video for explanation of solution.



The [Box Stacking problem is a variation of LIS problem](). We need to build a maximum height stack.
Following are the key points to note in the problem statement:
1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.
2) We can rotate boxes. For example, if there is a box with dimensions {1x2x3} where 1 is height, 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.
Following is the **solution** based on [DP solution of LIS problem]().
1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.
2) Sort the above generated 3n boxes in decreasing order of base area.
3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.
MSH(i) = Maximum possible Stack Height with box i at top of stack
MSH(i) = { Max (MSH(j)) + height(i) } where j < i and width(j) > width(i) and depth(j) > depth(i).
If there is no such j then MSH(i) = height(i)
4) To get overall maximum height, we return max(MSH(i)) where 0 < i < n
Following is C++ implementation of the above solution.

```
/* Dynamic Programming implementation of Box Stacking problem */

#include<stdio.h>

#include<stdlib.h>



/* Representation of a box */

struct Box

{

  // h -> height, w -> width, d -> depth

  int h, w, d;  // for simplicity of solution, always keep w <= d
```

```
};

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

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

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

/* Returns the height of the tallest stack that can be formed with give type of boxes */
int maxStackHeight( Box arr[], int n )
{
   /* Create an array of all rotations of given boxes
      For example, for a box {1, 2, 3}, we consider three
      instances{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}} */
   Box rot[3*n];
   int index = 0;
   for (int i = 0; i < n; i++)
   {
      // Copy the original box
      rot[index] = arr[i];
      index++;
```

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

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

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

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

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

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

    /* Compute optimized msh values in bottom up manner */
    for (int i = 1; i < n; i++ )
```

```
      for (int j = 0; j < i; j++ )
         if ( rot[i].w < rot[j].w &&
              rot[i].d < rot[j].d &&
              msh[i] < msh[j] + rot[i].h
            )
         {
              msh[i] = msh[j] + rot[i].h;
         }


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

   return max;
}

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

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

  return 0;
}
```

Output:

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

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

```
10 x 12 x 32

12 x 10 x 32

32 x 10 x 12

4 x 6 x 7

4 x 5 x 6

6 x 4 x 7

5 x 4 x 6

7 x 4 x 6

6 x 4 x 5

1 x 2 x 3

2 x 1 x 3

3 x 1 x 2
```

The height 60 is obtained by boxes { {**3**, 1, 2}, {**1**, 2, 3}, {**6**, 4, 5}, {**4**, 5, 6}, {**4**, 6, 7}, {**32**, 10, 12}, {**10**, 12, 32}}
Time Complexity: O(n^2)
Auxiliary Space: O(n)
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 21 (Variations of LIS)

We have discussed Dynamic Programming solution for Longest Increasing Subsequence problem in this post and a O(nLogn) solution in this post. Following are commonly asked variations of the standard LIS problem.
1. Building Bridges: Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x-coordinates a(1) … a(n) and n cities on the northern bank with x-coordinates b(1) … b(n). You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city i on the northern bank to city i on the southern bank.

```
8      1      4      3      5      2      6      7

<---- Cities on the other bank of river---->

--------------------------------------------

  <--------------- River--------------->

--------------------------------------------

1      2      3      4      5      6      7      8

<------- Cities on one bank of river------->
```

Source: Dynamic Programming Practice Problems. The link also has well explained solution for the problem.

2. Maximum Sum Increasing Subsequence: Given an array of n positive integers. Write a program to find the 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 {1, 2, 3, 100}. The solution to this problem has been published here.

3. The Longest Chain 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. The solution to this problem has been published here.

4. 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.
Source: Dynamic Programming Practice Problems. The link also has well explained solution for the problem.

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

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

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if b < c. Chain of pairs can be formed in this fashion. Find the longest chain which can be formed from a given set of pairs.
Source: Amazon Interview | Set 2
For example, if the given pairs are {{5, 24}, {39, 60}, {15, 28}, {27, 40}, {50, 90} }, then the longest chain that can be formed is of length 3, and the chain is {{5, 24}, {27, 40}, {50, 90}}
This problem is a variation of standard Longest Increasing Subsequence problem. Following is a simple two step process.
1) Sort given pairs in increasing order of first (or smaller) element.
2) Now run a modified LIS process where we compare the second element of already finalized LIS with the first element of new LIS being constructed.
The following code is a slight modification of method 2 of this post.

```
#include<stdio.h>

#include<stdlib.h>


// Structure for a pair

struct pair

{

  int a;

  int b;
```

```
};


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

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

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

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

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

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

   return max;
}
```

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

Output:

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

Time Complexity: O(n^2) where n is the number of pairs.
The given problem is also a variation of Activity Selection problem and can be solved in (nLogn) time. To solve it as a activity selection problem, consider the first element of a pair as start time in activity selection problem, and the second element of pair as end time. Thanks to Palash for suggesting this approach.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 19 (Word Wrap Problem)

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width.
The word processors like MS Word do task of placing line breaks. The idea is to have balanced lines. In other words, not have few lines with lots of extra spaces and some lines with small amount of extra spaces.

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

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

Following is the optimized arrangement of words in 3 lines
Geeks for Geeks
```

```
presents word

wrap problem



The total extra spaces in line 1, line 2 and line 3 are 0, 2 and 3 respectively.

So optimal value of total cost is 0 + 2*2 + 3*3 = 13
```

Please note that the total cost function is not sum of extra spaces, but sum of cubes (or square is also used) of extra spaces. The idea behind this cost function is to balance the spaces among lines. For example, consider the following two arrangement of same set of words:
1) There are 3 lines. One line has 3 extra spaces and all other lines have 0 extra spaces. Total extra spaces = 3 + 0 + 0 = 3. Total cost = 3*3*3 + 0*0*0 + 0*0*0 = 27.
2) There are 3 lines. Each of the 3 lines has one extra space. Total extra spaces = 1 + 1 + 1 = 3. Total cost = 1*1*1 + 1*1*1 + 1*1*1 = 3.
Total extra spaces are 3 in both scenarios, but second arrangement should be preferred because extra spaces are balanced in all three lines. The cost function with cubic sum serves the purpose because the value of total cost in second scenario is less.
Method 1 (Greedy Solution)
The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives optimal solution for many cases, but doesn't give optimal solution in all cases. For example, consider the following string "aaa bb cc ddddd" and line width as 6. Greedy method will produce following output.

```
aaa bb

cc

ddddd
```

Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So total cost is 0 + 64 + 1 = 65.
But the above solution is not the best solution. Following arrangement has more balanced spaces. Therefore less value of total cost function.

```
aaa

bb cc

ddddd
```

Extra spaces in the above 3 lines are 3, 1 and 1 respectively. So total cost is 27 + 1 + 1 = 29.
Despite being sub-optimal in some cases, the greedy approach is used by many word processors like MS Word and OpenOffice.org Writer.
Method 2 (Dynamic Programming)
The following Dynamic approach strictly follows the algorithm given in solution of Cormen book. First we compute costs of all possible lines in a 2D table lc[][]. The value lc[i][j] indicates the cost to put words from i to j in a single line where i and j are indexes of words in the input sequences. If a sequence of words from i to j cannot fit in a single line, then lc[i][j] is considered infinite (to avoid it from being a part of the solution). Once we have the lc[][] table constructed, we can calculate total cost using following recursive formula. In the following formula, C[j] is the optimized total cost for arranging words from 1 to j.

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

The above recursion has overlapping subproblem property. For example, the solution of subproblem c(2) is used by c(3), C(4) and so on. So Dynamic Programming is used to store the results of subproblems. The array c[] can be computed from left to right, since each value depends only on earlier values.
To print the output, we keep track of what words go on what lines, we can keep a parallel p array that points to where each c value came from. The last line starts at word p[n] and goes through word n. The previous line starts at word p[p[n]] and goes through word p[nÂ] – 1, etc. The function printSolution() uses p[] to print the solution.
In the below program, input is an array l[] that represents lengths of words in a sequence. The value l[i] indicates length of the ith word (i starts from 1) in theinput sequence.

```
// A Dynamic programming solution for Word Wrap Problem

#include <limits.h>

#include <stdio.h>

#define INF INT_MAX


// A utility function to print the solution

int printSolution (int p[], int n);


// l[] represents lengths of different words in input sequence. For example,

// l[] = {3, 2, 2, 5} is for a sentence like "aaa bb cc ddddd".  n is size of

// l[] and M is line width (maximum no. of characters that can fit in a line)

void solveWordWrap (int l[], int n, int M)

{

    // For simplicity, 1 extra space is used in all below arrays


    // extras[i][j] will have number of extra spaces if words from i

    // to j are put in a single line

    int extras[n+1][n+1];


    // lc[i][j] will have cost of a line which has words from

    // i to j

    int lc[n+1][n+1];


    // c[i] will have total cost of optimal arrangement of words

    // from 1 to i
```

```
int c[n+1];

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

int i, j;

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

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

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

    printSolution(p, n);
}

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

// Driver program to test above functions
int main()
```

```
{
    int l[] = {3, 2, 2, 5};
    int n = sizeof(l)/sizeof(l[0]);
    int M = 6;
    solveWordWrap (l, n, M);
    return 0;
}
```

Output:

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

Time Complexity: O(n^2)
Auxiliary Space: O(n^2) The auxiliary space used in the above program cane be optimized to O(n) (See the reference 2 for details)
References:
http://en.wikipedia.org/wiki/Word_wrap
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 18 (Partition problem)

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

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


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

Following are the two main steps to solve this problem:
1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.
The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Recursive Solution

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

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

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

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

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

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

   /* else, check if sum can be obtained by any of the following
      (a) including the last element
```

```
      (b) excluding the last element
   */
   return isSubsetSum (arr, n-1, sum) || isSubsetSum (arr, n-1, sum-arr[n-1]);
}

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

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

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

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

```
}
```

Output:

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

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

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

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

             equal to i, otherwise false
```

```
// A Dynamic Programming solution to partition problem

#include <stdio.h>


// Returns true if arr[] can be partitioned in two subsets of

// equal sum, otherwise false

bool findPartiion (int arr[], int n)

{

    int sum = 0;

    int i, j;


    // Caculcate sun of all elements

    for (i = 0; i < n; i++)

      sum += arr[i];


    if (sum%2 != 0)

       return false;


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


    // initialize top row as true

    for (i = 0; i <= n; i++)
```

```
       part[0][i] = true;

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

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

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

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

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

```
  if (findPartiion(arr, n) == true)

     printf("Can be divided into two subsets of equal sum");

  else

     printf("Can not be divided into two subsets of equal sum");

  getchar();

  return 0;

}
```

Output:

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

Following diagram shows the values in partition table. The diagram is taken form the [wiki page of partition problem](http://en.wikipedia.org/wiki/Partition_problem).

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

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

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

Time Complexity: O(sum*n)
Auxiliary Space: O(sum*n)
Please note that this solution will not be feasible for arrays with big sum.
References:
http://en.wikipedia.org/wiki/Partition_problem

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

Dynamic Programming | Set 17 (Palindrome Partitioning)

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

Solution

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

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

```
// i is the starting index and j is the ending index. i must be passed as 0 and j as n-1

minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.

minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.


// If none of the above conditions is true, then minPalPartion(str, i, j) can be

// calculated recursively using the following formula.

minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +

                                 minPalPartion(str, k+1, j) }

                           where k varies from i to j-1
```

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

```
// Dynamic Programming Solution for Palindrome Partitioning Problem

#include <stdio.h>

#include <string.h>

#include <limits.h>


// A utility function to get minimum of two integers

int min (int a, int b) { return (a < b)? a : b; }


// Returns the minimum number of cuts needed to partition a string

// such that every part is a palindrome

int minPalPartion(char *str)

{

    // Get the length of the string

    int n = strlen(str);
```

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

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

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

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

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

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


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

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

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

Output:

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

Time Complexity: O(n^3)
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

++

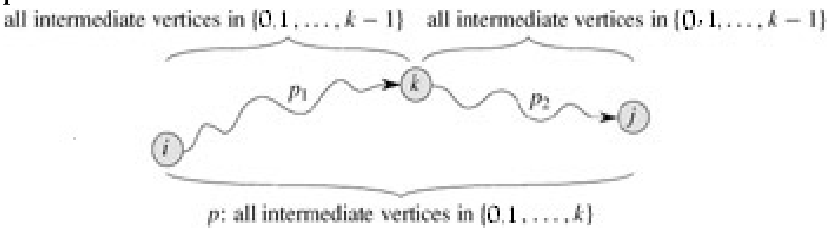
Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

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

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

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

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

there are two possible cases.
1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].
The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is C implementation of the Floyd Warshall algorithm.

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


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


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


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


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


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

```
            dist[i][j] = graph[i][j];


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

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

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances"
```

```
            " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

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

    // Print the solution
```

```
    floydWarshell(graph);

    return 0;

}
```

Output:

```
Following matrix shows the shortest distances between every pair of vertices

      0      5      8      9

    INF      0      3      4

    INF    INF      0      1

    INF    INF    INF      0
```

Time Complexity: O(V^3)
The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.
Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmatic overflow.

```
#include<limits.h>



#define INF INT_MAX

..........................

if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j])

    dist[i][j] = dist[i][k] + dist[k][j];

...........................
```

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

Dynamic Programming | Set 15 (Longest Bitonic Subsequence)

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

```
Input arr[] = {1, 11, 2, 10, 4, 5, 2, 1};
```

```
Output: 6 (A Longest Bitonic Subsequence of length 6 is 1, 2, 10, 4, 2, 1)


Input arr[] = {12, 11, 40, 5, 3, 1}

Output: 5 (A Longest Bitonic Subsequence of length 5 is 12, 11, 5, 3, 1)


Input arr[] = {80, 60, 30, 40, 20, 10}

Output: 5 (A Longest Bitonic Subsequence of length 5 is 80, 60, 30, 20, 10)
```

Source: Microsoft Interview Question

Solution

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

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

```
/* Dynamic Programming implementation of longest bitonic subsequence problem */

#include<stdio.h>

#include<stdlib.h>


/* lbs() returns the length of the Longest Bitonic Subsequence in

    arr[] of size n. The function mainly creates two temporary arrays

    lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.


    lis[i] ==> Longest Increasing subsequence ending with arr[i]

    lds[i] ==> Longest decreasing subsequence starting with arr[i]

*/

int lbs( int arr[], int n )

{

   int i, j;


   /* Allocate memory for LIS[] and initialize LIS values as 1 for

      all indexes */

   int *lis = new int[n];

   for ( i = 0; i < n; i++ )

      lis[i] = 1;
```

```
    /* Compute LIS values from left to right */
    for ( i = 1; i < n; i++ )
       for ( j = 0; j < i; j++ )
          if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
             lis[i] = lis[j] + 1;

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

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


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

/* Driver program to test above function */
int main()
{
  int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15};
  int n = sizeof(arr)/sizeof(arr[0]);
```

```
  printf("Length of LBS is %d\n", lbs( arr, n ) );


  getchar();

  return 0;

}
```

Output:

```
 Length of LBS is 7
```

Time Complexity: O(n^2)
Auxiliary Space: O(n)
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
++

Dynamic Programming | Set 14 (Maximum Sum Increasing Subsequence)

Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the intgers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 (1 + 2 + 3 + 100), if the input array is {3, 4, 5, 10}, then output should be 22 (3 + 4 + 5 + 10) and if the input array is {10, 5, 4, 3}, then output should be 10
Solution
This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. We need a slight change in the Dynamic Programming solution of LIS problem. All we need to change is to use sum as a criteria instead of length of increasing subsequence. Following is C implementation for Dynamic Programming solution of the problem.

```
/* Dynamic Programming implementation of Maximum Sum Increasing

   Subsequence (MSIS) problem */

#include<stdio.h>


/* maxSumIS() returns the maximum sum of increasing subsequence in arr[] of

   size n */

int maxSumIS( int arr[], int n )

{

   int *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;

}


/* Driver program to test above function */

int main()

{

  int arr[] = {1, 101, 2, 3, 100, 4, 5};

  int n = sizeof(arr)/sizeof(arr[0]);

  printf("Sum of maximum sum increasing subsequence is %d\n",

         maxSumIS( arr, n ) );


  getchar();

  return 0;

}
```

Time Complexity: O(n^2)
Source: [Maximum Sum Increasing Subsequence Problem]()
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

++

Dynamic Programming | Set 13 (Cutting a Rod)

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

```
length   | 1   2   3   4   5   6   7   8

--------------------------------------------

price    | 1   5   8   9  10  17  17  20
```

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

```
length   | 1   2   3   4   5   6   7   8

--------------------------------------------

price    | 3   5   8   9  10  17  17  20
```

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

1) Optimal Substructure:

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

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

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

2) Overlapping Subproblems

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

```
// A Naive recursive solution for Rod cutting problem

#include<stdio.h>

#include<limits.h>


// A utility function to get the maximum of two integers

int max(int a, int b) { return (a > b)? a : b;}


/* Returns the best obtainable price for a rod of length n and

   price[] as prices of different pieces */
```

```
int cutRod(int price[], int n)
{
   if (n <= 0)
     return 0;
   int max_val = INT_MIN;

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

   return max_val;
}

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

Output:

```
Maximum Obtainable Value is 22
```

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

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

                            cR(4)
                 /        /       \      \
```

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

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

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

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

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

```
          max_val = max(max_val, price[j] + val[i-j-1]);

        val[i] = max_val;

    }


    return val[n];

}


/* Driver program to test above functions */

int main()

{

    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};

    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));

    getchar();

    return 0;

}
```

Output:

```
Maximum Obtainable Value is 22
```

Time Complexity of the above implementation is O(n^2) which is much better than the worst case time complexity of Naive Recursive implementation.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 12 (Longest Palindromic Subsequence)

Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is "BBABCBCAB", then the output should be 7 as "BABCBAB" is the longest palindromic subseuqnce in it. "BBBBB" and "BBCBB" are also palindromic subsequences of the given sequence, but not the longest ones.
The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.
1) Optimal Substructure:
Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].
If last and first characters of X are same, then L(0, n-1) = L(1, n-2) + 2.
Else L(0, n-1) = MAX (L(1, n-1), L(0, n-2)).
Following is a general recursive solution with all cases handled.

```
// Everay single character is a palindrom of length 1

L(i, i) = 1 for all indexes i in given sequence


// IF first and last characters are not same

If (X[i] != X[j])  L(i, j) =  max{L(i + 1, j),L(i, j - 1)}


// If there are only 2 characters and both are same

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


// If there are more than two characters, and first and last

// characters are same

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

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

```
#include<stdio.h>

#include<string.h>


// A utility function to get max of two integers

int max (int x, int y) { return (x > y)? x : y; }


// Returns the length of the longest palindromic subsequence in seq

int lps(char *seq, int i, int j)

{

   // Base Case 1: If there is only 1 character

   if (i == j)

     return 1;


   // Base Case 2: If there are only 2 characters and both are same

   if (seq[i] == seq[j] && i + 1 == j)

     return 2;
```

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

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

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

Output:

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

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

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

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

Dynamic Programming Solution

```
#include<stdio.h>

#include<string.h>


// A utility function to get max of two integers

int max (int x, int y) { return (x > y)? x : y; }


// Returns the length of the longest palindromic subsequence in seq

int lps(char *str)

{

   int n = strlen(str);

   int i, j, cl;

   int L[n][n];  // Create a table to store results of subproblems




   // Strings of length 1 are palindrome of lentgh 1

   for (i = 0; i < n; i++)

      L[i][i] = 1;


    // Build the table. Note that the lower diagonal values of table are

    // useless and not filled in the process. The values are filled in a

    // manner similar to Matrix Chain Multiplication DP solution (See

    // http://www.geeksforgeeks.org/archives/15553). cl is length of

    // substring

    for (cl=2; cl<=n; cl++)

    {

        for (i=0; i<n-cl+1; i++)

        {

            j = i+cl-1;

            if (str[i] == str[j] && cl == 2)

               L[i][j] = 2;

            else if (str[i] == str[j])
```

```
                L[i][j] = L[i+1][j-1] + 2;
            else
                L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }

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

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

Output:

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

Time Complexity of the above implementation is O(n^2) which is much better than the worst case time complexity of Naive Recursive implementation.
This problem is close to the Longest Common Subsequence (LCS) problem. In fact, we can use LCS as a subroutine to solve this problem. Following is the two step solution that uses LCS.
1) Reverse the given sequence and store the reverse in another array say rev[0..n-1]
2) LCS of the given sequence and rev[] will be the longest palindromic sequence.
This solution is also a O(n^2) solution.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
References:
http://users.eecs.northwestern.edu/~dda902/336/hw6-sol.pdf
++

Dynamic Programming | Set 11 (Egg Dropping Puzzle)

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

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:
…..An egg that survives a fall can be used again.
…..A broken egg must be discarded.
…..The effect of a fall is the same for all eggs.
…..If an egg breaks when dropped, then it would break if dropped from a higher floor.
…..If an egg survives a fall then it would survive a shorter fall.
…..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.
If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.
Source: [Wiki for Dynamic Programming]
In this post, we will discuss solution to a general problem with n eggs and k floors. The solution is to try dropping an egg from every floor (from 1 to k) and recursively calculate the minimum number of droppings needed in worst case. The floor which gives the minimum value in worst case is going to be part of the solution.
In the following solutions, we return the minimum number of trails in worst case; these solutions can be easily modified to print floor numbers of every trials also.

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

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

```
# include <stdio.h>

# include <limits.h>


// A utility function to get maximum of two integers

int max(int a, int b) { return (a > b)? a: b; }
```

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

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

    int min = INT_MAX, x, res;

    // Consider all droppings from 1st floor to kth floor and
    // return the minimum of these values plus 1.
    for (x = 1; x <= k; x++)
    {
        res = max(eggDrop(n-1, x-1), eggDrop(n, k-x));
        if (res < min)
            min = res;
    }

    return min + 1;
}

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

```
    return 0;
}
```

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

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

```
                         E(2,4)
                           |
          -------------------------------------
          |             |           |         |
          |             |           |         |
      x=1/\          x=2/\      x=3/ \    x=4/ \
        /  \           /  \       ....      ....
       /    \         /    \
 E(1,0)  E(2,3)     E(1,1)  E(2,2)
          /\  /\...         /  \
      x=1/  \               .....
        /    \
     E(1,0)  E(2,2)
            /   \
            ......

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

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

Dynamic Programming Solution

Following is C/C++ implementation for Egg Dropping problem using Dynamic Programming.

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

```
# include <limits.h>


// A utility function to get maximum of two integers

int max(int a, int b) { return (a > b)? a: b; }


/* Function to get minimum number of trails needed in worst

  case with n eggs and k floors */

int eggDrop(int n, int k)

{

    /* A 2D table where entery eggFloor[i][j] will represent minimum

       number of trials needed for i eggs and j floors. */

    int eggFloor[n+1][k+1];

    int res;

    int i, j, x;


    // We need one trial for one floor and0 trials for 0 floors

    for (i = 1; i <= n; i++)

    {

        eggFloor[i][1] = 1;

        eggFloor[i][0] = 0;

    }


    // We always need j trials for one egg and j floors.

    for (j = 1; j <= k; j++)

        eggFloor[1][j] = j;


    // Fill rest of the entries in table using optimal substructure

    // property

    for (i = 2; i <= n; i++)

    {

        for (j = 2; j <= k; j++)

        {

            eggFloor[i][j] = INT_MAX;
```

```
            for (x = 1; x <= j; x++)

            {

                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);

                if (res < eggFloor[i][j])

                    eggFloor[i][j] = res;

            }

        }

    }


    // eggFloor[n][k] holds the result

    return eggFloor[n][k];

}


/* Driver program to test to pront printDups*/

int main()

{

    int n = 2, k = 36;

    printf ("\nMinimum number of trials in worst case with %d eggs and "

             "%d floors is %d \n", n, k, eggDrop(n, k));

    return 0;

}
```

```
Output:

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

Time Complexity: O(nk^2)
Auxiliary Space: O(nk)
As an exercise, you may try modifying the above DP solution to print all intermediate floors (The floors used for minimum trail solution).
References:
http://archive.ite.journal.informs.org/Vol4No1/Sniedovich/index.php
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

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

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items

respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).
A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.
1) Optimal Substructure:
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).
If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.
2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of 0-1 Knapsack problem */

#include<stdio.h>


// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b)? a : b; }


// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

   // Base Case

   if (n == 0 || W == 0)

       return 0;


   // If weight of the nth item is more than Knapsack capacity W, then

   // this item cannot be included in the optimal solution

   if (wt[n-1] > W)

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


   // Return the maximum of two cases: (1) nth item included (2) not included

   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),

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

                  );

}
```

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

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

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

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

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

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

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

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

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

   return K[n][W];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
```

```
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.
References:
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf
http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
+++

Dynamic Programming | Set 9 (Binomial Coefficient)

Following are common definition of Binomial Coefficients.
1) A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.
2) A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.
The Problem
Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.
1) Optimal Substructure
The value of C(n, k) can recursively calculated using following standard formula for Binomial Cofficients.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    return C[n][k];
}

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

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

int main()

{

    int n = 5, k = 2;

    printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );

    return 0;

}
```

Time Complexity: O(n*k)
Auxiliary Space: O(n*k)
Following is a space optimized version of the above code. The following code only uses O(k). Thanks to AK for suggesting this method.

```
// A space optimized Dynamic Programming Solution

int binomialCoeff(int n, int k)

{

    int* C = (int*)calloc(k+1, sizeof(int));

    int i, j, res;


    C[0] = 1;


    for(i = 1; i <= n; i++)

    {

        for(j = min(i, k); j > 0; j--)

            C[j] = C[j] + C[j-1];

    }


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


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


    return res;

}
```

Time Complexity: O(n*k)

Auxiliary Space: O(k)

References:

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

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

++

Dynamic Programming | Set 8 (Matrix Chain Multiplication)

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

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

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

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

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

Clearly the first parenthesization requires less number of operations.

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

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

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

```
((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30
```

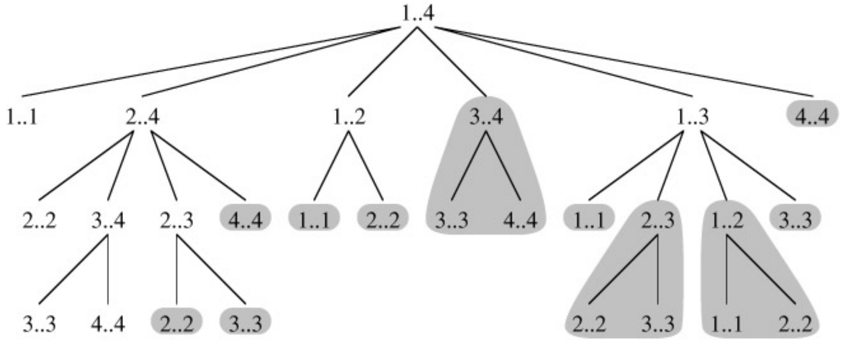
```
Input: p[] = {10, 20, 30}

Output: 6000

There are only two matrices of dimensions 10x20 and 20x30. So there

is only one way to multiply the matrices, cost of which is 10*20*30
```

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

```
/* A naive recursive implementation that simply follows the above optimal

 substructure property */

#include<stdio.h>

#include<limits.h>


// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

int MatrixChainOrder(int p[], int i, int j)

{

    if(i == j)

        return 0;

    int k;

    int min = INT_MAX;

    int count;


    // place parenthesis at different places between first and last matrix,

    // recursively calculate count of multiplcations for each parenthesis

    // placement and return the minimum count

    for (k = i; k <j; k++)

    {
```

```
        count = MatrixChainOrder(p, i, k) +

                MatrixChainOrder(p, k+1, j) +

                p[i-1]*p[k]*p[j];


        if (count < min)

            min = count;

    }


    // Return minimum count

    return min;

}


// Driver program to test above function

int main()

{

    int arr[] = {1, 2, 3, 4, 3};

    int n = sizeof(arr)/sizeof(arr[0]);


    printf("Minimum number of multiplications is %d ",

                          MatrixChainOrder(arr, 1, n-1));


    getchar();

    return 0;

}
```

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



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

Dynamic Programming Solution

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

```
// See the Cormen book for details of the following algorithm

#include<stdio.h>

#include<limits.h>


// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

int MatrixChainOrder(int p[], int n)

{


    /* For simplicity of the program, one extra row and one extra column are

       allocated in m[][].  0th row and 0th column of m[][] are not used */

    int m[n][n];


    int i, j, k, L, q;


    /* m[i,j] = Minimum number of scalar multiplications needed to compute

       the matrix A[i]A[i+1]...A[j] = A[i..j] where dimention of A[i] is

       p[i-1] x p[i] */


    // cost is zero when multiplying one matrix.

    for (i = 1; i < n; i++)

        m[i][i] = 0;


    // L is chain length.

    for (L=2; L<n; L++)

    {

        for (i=1; i<=n-L+1; i++)

        {

            j = i+L-1;

            m[i][j] = INT_MAX;

            for (k=i; k<=j-1; k++)
```

```
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

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

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

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

    getchar();
    return 0;
}
```

Time Complexity: O(n^3)
Auxiliary Space: O(n^2)
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
References:
http://en.wikipedia.org/wiki/Matrix_chain_multiplication
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm
++

Dynamic Programming | Set 7 (Coin Change)

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

```
#include<stdio.h>


// Returns the count of ways we can sum  S[0...m-1] coins to get sum n

int count( int S[], int m, int n )

{

    // If n is 0 then there is 1 solution (do not include any coin)

    if (n == 0)

        return 1;


    // If n is less than 0 then no solution exists

    if (n < 0)

        return 0;


    // If there are no coins and n is greater than 0, then no solution exist

    if (m <=0 && n >= 1)

        return 0;


    // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]

    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );

}


// Driver program to test above function

int main()

{

    int i, j;

    int arr[] = {1, 2, 3};
```

```
    int m = sizeof(arr)/sizeof(arr[0]);
    printf("%d ", count(arr, m, 4));
    getchar();
    return 0;
}
```

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

```
C() --> count()
                              C({1,2,3}, 5)
                           /                 \
                         /                     \
             C({1,2,3}, 2)                 C({1,2}, 5)
            /     \                          /         \
           /        \                      /            \
C({1,2,3}, -1)  C({1,2}, 2)        C({1,2}, 3)    C({1}, 5)
               /     \             /    \             /     \
             /         \         /       \          /        \
    C({1,2},0)  C({1},2)   C({1,2},1) C({1},3)    C({1}, 4)  C({}, 5)
                   / \       / \        / \         /     \
                  /   \     /   \      /   \       /       \
                .      . .     .   .     .   C({1}, 3) C({}, 4)
                                                  /  \
                                                 /    \
                                                .      .
```

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

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

```
int count( int S[], int m, int n )

{

    int i, j, x, y;


    // We need n+1 rows as the table is consturcted in bottom up manner using

    // the base case 0 value case (n = 0)

    int table[n+1][m];


    // Fill the enteries for 0 value case (n = 0)

    for (i=0; i<m; i++)

        table[0][i] = 1;


    // Fill rest of the table enteries in bottom up manner

    for (i = 1; i < n+1; i++)

    {

        for (j = 0; j < m; j++)

        {

            // Count of solutions including S[j]

            x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;


            // Count of solutions excluding S[j]

            y = (j >= 1)? table[i][j-1]: 0;


            // total count

            table[i][j] = x + y;

        }

    }

    return table[n][m-1];

}


// Driver program to test above function

int main()

{
```

```
    int arr[] = {1, 2, 3};

    int m = sizeof(arr)/sizeof(arr[0]);

    int n = 4;

    printf(" %d ", count(arr, m, n));

    return 0;

}
```

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

```
int count( int S[], int m, int n )

{

    // table[i] will be storing the number of solutions for

    // value i. We need n+1 rows as the table is consturcted

    // in bottom up manner using the base case (n = 0)

    int table[n+1];


    // Initialize all table values as 0

    memset(table, 0, sizeof(table));


    // Base case (If given value is 0)

    table[0] = 1;


    // Pick all coins one by one and update the table[] values

    // after the index greater than or equal to the value of the

    // picked coin

    for(int i=0; i<m; i++)

        for(int j=S[i]; j<=n; j++)

            table[j] += table[j-S[i]];


    return table[n];

}
```

Thanks to [Rohan Laishram]() for suggesting this space optimized version.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

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

Minimum number of jumps to reach end

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

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}

Output: 3 (1-> 3 -> 8 ->9)
```

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps eg to 5 or 8 or 9.
Method 1 (Naive Recursive Approach)
A naive approach is to start from the first element and recursively call for all the elements reachable from first element. The minimum number of jumps to reach end from first can be calculated using minimum number of jumps needed to reach end from the elements reachable from first.
minJumps(start, end) = Min (minJumps(k, end)) for all k reachable from start

```
#include <stdio.h>

#include <limits.h>


// Returns minimum number of jumps to reach arr[h] from arr[l]

int minJumps(int arr[], int l, int h)

{

   // Base case: when source and destination are same

   if (h == l)

     return 0;


   // When nothing is reachable from the given source

   if (arr[l] == 0)

     return INT_MAX;


   // Traverse through all the points reachable from arr[l]. Recursively

   // get the minimum number of jumps needed to reach arr[h] from these

   // reachable points.

   int min = INT_MAX;
```

```
    for (int i = l+1; i <= h && i <= l + arr[l]; i++)

    {

        int jumps = minJumps(arr, i, h);

        if(jumps != INT_MAX && jumps + 1 < min)

            min = jumps + 1;

    }


    return min;

}


// Driver program to test above function

int main()

{

  int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};

  int n = sizeof(arr)/sizeof(arr[0]);

  printf("Minimum number of jumps to reach end is %d ", minJumps(arr, 0, n-1));

  return 0;

}
```

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

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

```
#include <stdio.h>

#include <limits.h>


// 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] = jumps[j] + 1;
                 break;
            }
        }
    }
    return jumps[n-1];
}

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

Thanks to [paras](#)for suggesting this method.
Time Complexity: O(n^2)

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

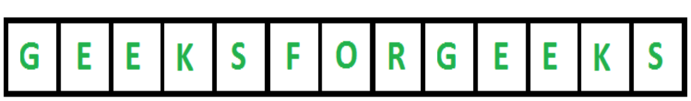
```
int minJumps(int arr[], int n)
{
    int *jumps = new int[n];  // jumps[0] will hold the result
    int min;

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

    int i, j;

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

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

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

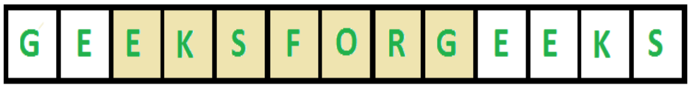
```
        {
            min = INT_MAX;  // initialize min value

            // following loop checks with all reachable points and
            // takes the minimum
            for (j = i+1; j < n && j <= arr[i] + i; j++)
            {
                if (min > jumps[j])
                    min = jumps[j];
            }

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

    return jumps[0];
}
```

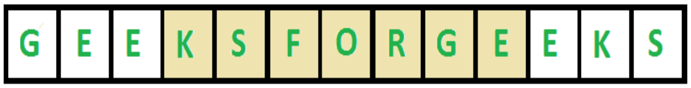
Time Complexity: O(n^2) in worst case.
Thanks to [Ashish]for suggesting this solution.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
+++

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 â€œABDEFGABEFâ€ are â€œBDEFGAâ€ and "DEFGAB", with length 6. For â€œBBBBâ€ the longest substring is â€œBâ€, 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.

Method 1 (Simple)

We can consider all substrings one by one and check for each substring whether it contains all unique characters or not. There will be n*(n+1)/2 substrings. Whether a substirng contains all unique characters or not can be checked in linear time by scanning it from left to right and keeping a map of visited characters. Time complexity of this solution would be O(n^3).

Method 2 (Linear Time)

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).

Implementation

```
#include<stdlib.h>

#include<stdio.h>

#define NO_OF_CHARS 256


int min(int a, int b);


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 = 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;
}

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

/* Driver program to test above function */
int main()
{
    char str[] = "ABDEFGABEF";
    printf("The input string is %s \n", str);
    int len =  longestUniqueSubsttr(str);
    printf("The length of the longest non-repeating character substring is %d", len);

    getchar();
    return 0;
}
```

Output

```
The input string is ABDEFGABEF
The length of the longest non-repeating character substring is 6
```

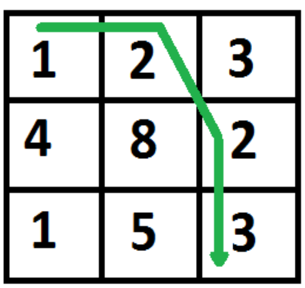
Time Complexity: O(n + d) where n is length of the input string and d is number of characters in input string alphabet. For example, if string consists of lowercase English characters then value of d is 26.
Auxiliary Space: O(d)
Algorithmic Paradigm: Dynamic Programming
As an exercise, try the modified version of the above problem where you need to print the maximum length NRCS also (the above program only prints length of it).
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 6 (Min Cost Path)

Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns cost of minimum cost path to reach (m, n) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach (m, n) is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1) and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.
For example, in the following figure, what is the minimum cost path to (2, 2)?

1	2	3
4	8	2
1	5	3

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



1) Optimal Substructure
The path to reach (m, n) must be through one of the 3 cells: (m-1, n-1) or (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as "minimum of the 3 cells plus cost[m][n]".
minCost(m, n) = min (minCost(m-1, n-1), minCost(m-1, n), minCost(m, n-1)) + cost[m][n]
2) Overlapping Subproblems
Following is simple recursive implementation of the MCP (Minimum Cost Path) problem. The implementation simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of MCP(Minimum Cost Path) problem */

#include<stdio.h>

#include<limits.h>

#define R 3

#define C 3
```

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

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

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

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

```
}
```

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

```
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)
```

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

```
/* Dynamic Programming implementation of MCP problem */

#include<stdio.h>

#include<limits.h>

#define R 3

#define C 3


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


int minCost(int cost[R][C], int m, int n)

{

     int i, j;


     // Instead of following line, we can use int tc[m+1][n+1] or

     // dynamically allocate memoery to save space. The following line is

     // used to keep te program simple and make it working on all compilers.

     int tc[R][C];
```

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

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

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

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

     return tc[m][n];
}

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

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

```
    printf(" %d ", minCost(cost, 2, 2));

    return 0;

}
```

Time Complexity of the DP implementation is O(mn) which is much better than Naive Recursive implementation.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 5 (Edit Distance)

Continuing further on dynamic programming series, *edit distance* is an interesting algorithm.
Problem: 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.
Identifying Recursive Methods:
What will be sub-problem in this case? Consider finding edit distance of part of the strings, say small prefix. Let us denote them as [1...i] and [1...j] for some 1< i < m and 1 < j < n. Clearly it is solving smaller instance of final problem,Â denote it as E(i, j). Our goal is finding E(m, n) and minimizing the cost.
In the prefix, we can right align the strings in three ways (i, -), (-, j) and (i, j). TheÂ hyphenÂ symbol (-) representing no character. An example can make it more clear.
Given strings SUNDAY and SATURDAY. We want to convert SUNDAY into SATURDAY with minimum edits. Let us pick i = 2 and j = 4 i.e. prefix strings are SUN and SATUÂ respectively (assume the strings indices start at 1). The right most characters can be aligned in three different ways.
Case 1: Align characters U and U. They are equal, no edit is required. We still left with the problem of i = 1 and j = 3, E(i-1, j-1).
Case 2: Align right character from first string and no character from second string. We need a deletion (D) here. We still left with problem of i = 1 and j = 4, E(i-1, j).
Case 3: Align right character from second string and no character from first string. We need an insertion (I) here. We still left with problem of i = 2 and j = 3, E(i, j-1).
Combining all the subproblems minimum cost of aligning prefix strings ending at i and j given by
E(i, j) = min([E(i-1, j) + D], [E(i, j-1) + I], Â [E(i-1, j-1) + R if i,j characters are not same])
We still not yet done. What will be base case(s)?
When both of the strings are of size 0, the cost is 0. When only one of the string is zero, we need edit operations as that of non-zero length string. Mathematically,
E(0, 0) = 0,Â E(i, 0) = i,Â E(0, j) = j
Now it is easy to complete recursive method. Go through the code for recursive algorithm (edit_distance_recursive).
Dynamic Programming Method:
We can calculate the complexity of recursive expression fairly easily.
T(m, n) = T(m-1, n-1) + T(m, n-1) + T(m-1, n) + C
The complexity of T(m, n) can be calculated by successiveÂ substitutionÂ method or solving homogeneous equation of two variables. It will result in an exponential complexity algorithm. It is evident from the recursion tree that it will be solving subproblems again and again. Few strings result in many overlapping subproblems (try the below program with strings *exponential* and *polynomial* and note the delay in recursive method).
We can tabulate the repeating subproblems and look them up when required next time (bottom up). A two dimensional array formed by the strings can keep track of the minimum cost till the current characterÂ comparison. The visualization code will help in understanding the construction of matrix.
The time complexity of dynamic programming method is *O(mn)* as we need to construct the table fully. The space complexity is also *O(mn)*. If we need only the cost of edit, we just need *O(min(m, n))* space as it is required only to keep track of the current row and previous row.
Usually the costs D, I and R are not same. In such case the problem can be represented as an acyclicÂ directed graph (DAG) with weights on each edge, and finding shortest path gives edit distance.

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

```
// Dynamic Programming implementation of edit distance

#include<stdio.h>

#include<stdlib.h>

#include<string.h>


// Change these strings to test the program

#define STRING_X "SUNDAY"

#define STRING_Y "SATURDAY"


#define SENTINEL (-1)

#define EDIT_COST (1)


inline

int min(int a, int b) {

   return a < b ? a : b;

}


// Returns Minimum among a, b, c

int Minimum(int a, int b, int c)

{

    return min(min(a, b), c);

}


// Strings of size m and n are passed.

// Construct the Table for X[0...m, m+1], Y[0...n, n+1]

int EditDistanceDP(char X[], char Y[])

{

    // Cost of alignment

    int cost = 0;

    int leftCell, topCell, cornerCell;
```

```
    int m = strlen(X)+1;

    int n = strlen(Y)+1;


    // T[m][n]

    int *T = (int *)malloc(m * n * sizeof(int));


    // Initialize table

    for(int i = 0; i < m; i++)

        for(int j = 0; j < n; j++)

            *(T + i * n + j) = SENTINEL;


    // Set up base cases

    // T[i][0] = i

    for(int i = 0; i < m; i++)

        *(T + i * n) = i;


    // T[0][j] = j

    for(int j = 0; j < n; j++)

        *(T + j) = j;


    // Build the T in top-down fashion

    for(int i = 1; i < m; i++)

    {

        for(int j = 1; j < n; j++)

        {

            // T[i][j-1]

            leftCell = *(T + i*n + j-1);

            leftCell += EDIT_COST; // deletion


            // T[i-1][j]

            topCell = *(T + (i-1)*n + j);

            topCell += EDIT_COST; // insertion
```

```
            // Top-left (corner) cell

            // T[i-1][j-1]

            cornerCell = *(T + (i-1)*n + (j-1) );


            // edit[(i-1), (j-1)] = 0 if X[i] == Y[j], 1 otherwise

            cornerCell += (X[i-1] != Y[j-1]); // may be replace


            // Minimum cost of current cell

            // Fill in the next cell T[i][j]

            *(T + (i)*n + (j)) = Minimum(leftCell, topCell, cornerCell);

        }

    }


    // Cost is in the cell T[m][n]

    cost = *(T + m*n - 1);

    free(T);

    return cost;

}


// Recursive implementation

int EditDistanceRecursion( char *X, char *Y, int m, int n )

{

    // Base cases

    if( m == 0 && n == 0 )

        return 0;


    if( m == 0 )

        return n;


    if( n == 0 )

        return m;


    // Recurse
```

```
    int left = EditDistanceRecursion(X, Y, m-1, n) + 1;

    int right = EditDistanceRecursion(X, Y, m, n-1) + 1;

    int corner = EditDistanceRecursion(X, Y, m-1, n-1) + (X[m-1] != Y[n-1]);


    return Minimum(left, right, corner);

}


int main()

{

    char X[] = STRING_X; // vertical

    char Y[] = STRING_Y; // horizontal


    printf("Minimum edits required to convert %s into %s is %d\n",

           X, Y, EditDistanceDP(X, Y) );

    printf("Minimum edits required to convert %s into %s is %d by recursion\n",

           X, Y, EditDistanceRecursion(X, Y, strlen(X), strlen(Y)));


    return 0;

}
```

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

Dynamic Programming | Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 and Set 2 respectively. We also discussed one example problem in Set 3. Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.
LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", '"acefg", .. etc are subsequences of "abcdefg". So a string of length n has 2^n different possible subsequences.
It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.
Examples:
LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.
LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.
The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important

properties of a Dynamic Programming (DP) Problem.

1) Optimal Substructure:

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

If last characters of both sequences match (or X[m-1] == Y[n-1]) then

L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then

L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Examples:

1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)

2) Consider the input strings “ABCDGH” and “AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:

L(â€œABCDGHâ€, â€œAEDFHRâ€) = MAX (L(â€œABCDGâ€, â€œAEDFH**R**â€), L(â€œABCDG**H**â€, â€œAEDFHâ€))

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

2) Overlapping Subproblems:

Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

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

int max(int a, int b);

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

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

```
}


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

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

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

  getchar();
  return 0;
}
```

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

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

In the above partial recursion tree, lcs(“AXY”, “AYZ”) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

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


int max(int a, int b);
```

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

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

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

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

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

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

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

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

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

  getchar();
  return 0;
}
```

Time Complexity of the above implementation is O(mn) which is much better than the worst case time complexity of Naive Recursive implementation.
The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.
Printing Longest Common Subsequence
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
References:
http://www.youtube.com/watch?v=V5hZoJ6uK-s
http://www.algorithmist.com/index.php/Longest_Common_Subsequence
http://www.ics.uci.edu/~eppstein/161/960229.html
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem
++

Dynamic Programming | Set 3 (Longest Increasing Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 and Set 2 respectively.
Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.
The longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, length of LIS for { 10, 22, 9, 33, 21, 50, 41, 60, 80 } is 6 and LIS is {10, 22, 33, 50, 60, 80}.
Optimal Substructure:
Let arr[0..n-1] be the input array and L(i) be the length of the LIS till index i such that arr[i] is part of LIS and arr[i] is the last element in LIS, then L(i) can be recursively written as.
L(i) = { 1 + Max (L(j)) } where j < i and arr[j] < arr[i] and if there is no such j then L(i) = 1
To get LIS of a given array, we need to return max(L(i)) where 0 < i < n

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

Overlapping Subproblems:

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

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

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

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

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

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

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

    // returns max
    return max;
}

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

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

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

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

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

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

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

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

   /* Pick maximum of all LIS values */
```

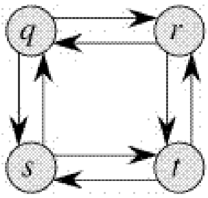
```
    for ( i = 0; i < n; i++ )
        if ( max < lis[i] )
            max = lis[i];

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

    return max;
}

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

  getchar();
  return 0;
}
```

Note that the time complexity of the above Dynamic Programmig (DP) solution is O(n^2) and there is a O(nLogn) solution for the LIS problem (see this). We have not discussed the nLogn solution here as the purpose of this post is to explain Dynamic Programmig with a simple example.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
++

Dynamic Programming | Set 2 (Optimal Substructure Property)

As we discussed in Set 1, following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.
1) Overlapping Subproblems
2) Optimal Substructure
We have already discussed Overlapping Subproblem property in the Set 1. Let us discuss Optimal Substructure property here.
2) Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
For example the shortest path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path

from x to v. The standard All Pair Shortest Path algorithms like Floydâ€“Warshall and Bellmanâ€“Ford are typical examples of Dynamic Programming.
On the other hand the Longest path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the CLRS book. There are two longest paths from q to t: q -> r ->t and q ->s->t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q->r->t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q->s->t->r.



We will be covering some example problems in future posts on Dynamic Programming.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
References:
http://en.wikipedia.org/wiki/Optimal_substructure
CLRS book
++

Dynamic Programming | Set 1 (Overlapping Subproblems Property)

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.
1) Overlapping Subproblems
2) Optimal Substructure
1) Overlapping Subproblems:
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these donâ€™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 doesnâ€™t have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
/* simple recursive program for Fibonacci numbers */

int fib(int n)

{

   if ( n <= 1 )

      return n;

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

}
```

Recursion tree for execution of *fib(5)*

```
                         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. There are following two different ways to store the values so that these values can be reused.
a) Memoization (Top Down):
b) Tabulation (Bottom Up):
a) Memoization (Top Down): The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.
Following is the memoized version for nth Fibonacci Number.

```
/* Memoized version for nth Fibonacci number */

#include<stdio.h>

#define NIL -1

#define MAX 100


int lookup[MAX];


/* Function to initialize NIL values in lookup table */

void _initialize()

{

  int i;

  for (i = 0; i < MAX; i++)

    lookup[i] = NIL;

}


/* function for nth Fibonacci number */

int fib(int n)
```

```
{
   if(lookup[n] == NIL)
   {
    if ( n <= 1 )
      lookup[n] = n;
    else
      lookup[n] = fib(n-1) + fib(n-2);
   }

   return lookup[n];
}

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

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

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

```
  return f[n];
}

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

Both tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in tabulated version, starting from the first entry, all entries are filled one by one. Unlike the tabulated version, all entries of the lookup table are not necessarily filled in memoized version. For example, memoized solution of LCS problem doesn't necessarily fill all entries.
To see the optimization achieved by memoized and tabulated versions over the basic recursive version, see the time taken by following runs for 40th Fibonacci number.
Simple recursive program
Memoized version
tabulated version
Also see method 2 of Ugly Number post for one more simple example where we have overlapping subproblems and we store the results of subproblems.
We will be covering Optimal Substructure Property and some more example problems in future posts on Dynamic Programming.
Try following questions as an exercise of this post.
1) Write a memoized version for LCS problem. Note that the tabular version is given in the CLRS book.
2) How would you choose between Memoization and Tabulation?
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
References:
http://www.cs.uiuc.edu/class/fa08/cs573/lectures/05-dynprog.pdf
http://web.iiit.ac.in/~avidullu/pdfs/dynprg/Dynamic%20Programming%20Lesson.pdf
http://www.youtube.com/watch?v=V5hZoJ6uK-s
++

Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141, ……..
In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation
$F_n = F_{n-1} + F_{n-2}$
with seed values
$F_0 = 0 \quad \text{and} \quad F_1 = 1.$
Write a function *int fib(int n)* that returns F_n. For example, if *n* = 0, then *fib()* should return 0. If n = 1, then it should return 1. For n > 1, it should return $F_{n-1} + F_{n-2}$
Following are different methods to get the nth Fibonacci number.
Method 1 (Use recursion)
A simple method that is a direct recusrive implementation mathematical recurance relation given above.

```
#include<stdio.h>

int fib(int n)

{

   if (n <= 1)

      return n;

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

}


int main ()

{

  int n = 9;

  printf("%d", fib(n));

  getchar();

  return 0;

}
```

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

```
                         fib(5)

                     /             \

               fib(4)                fib(3)

             /      \                /     \

         fib(3)      fib(2)         fib(2)    fib(1)

        /     \        /    \       /    \

  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)

  /    \

fib(1) fib(0)
```

Extra Space: O(n) if we consider the fuinction call stack size, otherwise O(1).
Method 2 (Use Dynamic Programming)
We can avoid the repeated work done is the method 1 by storing the Fibonacci numbers calculated so far.

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

```
int fib(int n)

{

  /* Declare an array to store fibonacci numbers. */

  int f[n+1];

  int i;


  /* 0th and 1st number of the series are 0 and 1*/

  f[0] = 0;

  f[1] = 1;


  for (i = 2; i <= n; i++)

  {

      /* Add the previous 2 numbers in the series

         and store it */

      f[i] = f[i-1] + f[i-2];

  }


  return f[n];

}


int main ()

{

  int n = 9;

  printf("%d", fib(n));

  getchar();

  return 0;

}
```

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

```
#include<stdio.h>

int fib(int n)

{

  int a = 0, b = 1, c, i;

  if( n == 0)

    return a;

  for (i = 2; i <= n; i++)

  {

     c = a + b;

     a = b;

     b = c;

  }

  return b;

}


int main ()

{

  int n = 9;

  printf("%d", fib(n));

  getchar();

  return 0;

}
```

Time Complexity: O(n)
Extra Space: O(1)
Method 4 (Using power of the matrix {{1,1},{1,0}})
This another O(n) which relies on the fact that if we n times multiply the matrix M = {{1,1},{1,0}} to itself (in other words calculate power(M, n)), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.
The matrix representation gives the following closed expression for the Fibonacci numbers:

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

```
#include <stdio.h>


/* Helper function that multiplies 2 matricies F and M of size 2*2, and

  puts the multiplication result back to F[][] */
```

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

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

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

  return F[0][0];
}

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

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

void power(int F[2][2], int n)
{
```

```
  int i;
  int M[2][2] = {{1,1},{1,0}};

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

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

Time Complexity: O(n)
Extra Space: O(1)
Method 5 (Optimized Method 4)
The method 4 can be optimized to work in O(Logn) time complexity. We can do recursive multiplication to get power(M, n) in the prevous method (Similar to the optimization done in this post)

```
#include <stdio.h>

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

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

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

```
  power(F, n-1);

  return F[0][0];

}


/* Optimized version of power() in method 4 */

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

{

  if( n == 0 || n == 1)

      return;

  int M[2][2] = {{1,1},{1,0}};


  power(F, n/2);

  multiply(F, F);


  if (n%2 != 0)

     multiply(F, M);

}


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

{

  int x =  F[0][0]*M[0][0] + F[0][1]*M[1][0];

  int y =  F[0][0]*M[0][1] + F[0][1]*M[1][1];

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

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


  F[0][0] = x;

  F[0][1] = y;

  F[1][0] = z;

  F[1][1] = w;

}


/* Driver program to test above function */

int main()
```

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

Time Complexity: **O(Logn)**
Extra Space: O(Logn) if we consider the function call stack size, otherwise O(1).
Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.
References:
http://en.wikipedia.org/wiki/Fibonacci_number
http://www.ics.uci.edu/~eppstein/161/960109.html
+++

Maximum size square sub-matrix with all 1s

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

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

The maximum square sub-matrix with all set bits is

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

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

```
1) Construct a sum matrix S[R][C] for the given M[R][C].
```

```
    a) Copy first row and first columns as it is from M[][] to S[][]

    b) For other entries, use following expressions to construct S[][]

        If M[i][j] is 1 then

           S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1

        Else /*If M[i][j] is 0*/

           S[i][j] = 0

2) Find the maximum entry in S[R][C]

3) Using the value and coordinates of maximum entry in S[i], print

   sub-matrix of M[][]
```

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

```
   0  1  1  0  1

   1  1  0  1  0

   0  1  1  1  0

   1  1  2  2  0

   1  2  2  3  1

   0  0  0  0  0
```

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

```
#include<stdio.h>

#define bool int

#define R 6

#define C 5


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

{

  int i,j;

  int S[R][C];

  int max_of_s, max_i, max_j;


  /* Set first column of S[][]*/

  for(i = 0; i < R; i++)

     S[i][0] = M[i][0];
```

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

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

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

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

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

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

```
  printMaxSubSquare(M);

  getchar();

}
```

Time Complexity: O(m*n) where m is number of rows and n is number of columns in the given matrix.
Auxiliary Space: O(m*n) where m is number of rows and n is number of columns in the given matrix.
Algorithmic Paradigm: Dynamic Programming
Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem
++

Ugly Numbers

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

METHOD 1 (Simple)
Thanks to Nedylko Draganov for suggesting this solution.
Algorithm:
Loop for all positive integers until ugly number count is smaller than n, if an integer is ugly than increment ugly number count.
To check if a number is ugly, divide the number by greatest divisible powers of 2, 3 and 5, if the number becomes 1 then it is an ugly number otherwise not.
For example, let us see how to check for 300 is ugly or not. Greatest divisible power of 2 is 4, after dividing 300 by 4 we get 75. Greatest divisible power of 3 is 3, after dividing 75 by 3 we get 25. Greatest divisible power of 5 is 25, after dividing 25 by 25 we get 1. Since we get 1 finally, 300 is ugly number.
Implementation:

```
# include<stdio.h>

# include<stdlib.h>


/*This function divides a by greatest divisible

  power of b*/

int maxDivide(int a, int b)

{

  while (a%b == 0)

   a = a/b;

  return a;

}
```

```
/* Function to check if a number is ugly or not */

int isUgly(int no)

{

  no = maxDivide(no, 2);

  no = maxDivide(no, 3);

  no = maxDivide(no, 5);


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

}


/* Function to get the nth ugly number*/

int getNthUglyNo(int n)

{

  int i = 1;

  int count = 1;   /* ugly number count */


  /*Check for all integers untill ugly count

    becomes n*/

  while (n > count)

  {

    i++;

    if (isUgly(i))

      count++;

  }

  return i;

}


/* Driver program to test above functions */

int main()

{

    unsigned no = getNthUglyNo(150);

    printf("150th ugly no. is %d ",  no);

    getchar();
```

```
    return 0;

}
```

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

METHOD 2 (Use Dynamic Programming)
Here is a time efficient solution with O(n) extra space
Algorithm:

```
1 Declare an array for ugly numbers:  ugly[150]

2 Initialize first ugly no:  ugly[0] = 1

3 Initialize three array index variables i2, i3, i5 to point to

   1st element of the ugly array:

        i2 = i3 = i5 =0;

4 Initialize 3 choices for the next ugly no:

         next_mulitple_of_2 = ugly[i2]*2;

         next_mulitple_of_3 = ugly[i3]*3

         next_mulitple_of_5 = ugly[i5]*5;

5 Now go in a loop to fill all ugly numbers till 150:

For (i = 1; i < 150; i++ )

{

    /* These small steps are not optimized for good

      readability. Will optimize them in C program */

    next_ugly_no  = Min(next_mulitple_of_2,

                                  next_mulitple_of_3,

                                  next_mulitple_of_5);

    if  (next_ugly_no  == next_mulitple_of_2)

    {

        i2 = i2 + 1;

        next_mulitple_of_2 = ugly[i2]*2;

    }

    if  (next_ugly_no  == next_mulitple_of_3)

    {

        i3 = i3 + 1;

        next_mulitple_of_3 = ugly[i3]*3;
```

```
     }
     if  (next_ugly_no  == next_mulitple_of_5)
     {
        i5 = i5 + 1;
        next_mulitple_of_5 = ugly[i5]*5;
     }
     ugly[i] =  next_ugly_no
}/* end of for loop */
6.return next_ugly_no
```

Example:
Let us see how it works

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

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

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

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

```
             = 4

    ugly[] =  | 1 | 2 | 3 |  4 |

    i2 = 2,  i3 =  1, i5 = 0  (i2 got incremented )


Fourth iteration

    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)

              = Min(6, 6, 5)

              = 5

    ugly[] =  | 1 | 2 | 3 |  4 | 5 |

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


Fifth iteration

    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)

              = Min(6, 6, 10)

              = 6

    ugly[] =  | 1 | 2 | 3 |  4 | 5 | 6 |

    i2 = 3,  i3 =  2, i5 = 1  (i2 and i3 got incremented )


Will continue same way till I < 150
```

Program:

```
# include<stdio.h>

# include<stdlib.h>

# define bool int


/* Function to find minimum of 3 numbers */

unsigned min(unsigned , unsigned , unsigned );


/* Function to get the nth ugly number*/

unsigned getNthUglyNo(unsigned n)

{

    unsigned *ugly =
```

```
            (unsigned *)(malloc (sizeof(unsigned)*n));

    unsigned i2 = 0, i3 = 0, i5 = 0;

    unsigned i;

    unsigned next_multiple_of_2 = 2;

    unsigned next_multiple_of_3 = 3;

    unsigned next_multiple_of_5 = 5;

    unsigned next_ugly_no = 1;

    *(ugly+0) = 1;


    for(i=1; i<n; i++)

    {

       next_ugly_no = min(next_multiple_of_2,

                          next_multiple_of_3,

                          next_multiple_of_5);

       *(ugly+i) = next_ugly_no;

       if(next_ugly_no == next_multiple_of_2)

       {

           i2 = i2+1;

           next_multiple_of_2 = *(ugly+i2)*2;

       }

       if(next_ugly_no == next_multiple_of_3)

       {

           i3 = i3+1;

           next_multiple_of_3 = *(ugly+i3)*3;

       }

       if(next_ugly_no == next_multiple_of_5)

       {

           i5 = i5+1;

           next_multiple_of_5 = *(ugly+i5)*5;

       }

    } /*End of for loop (i=1; i<n; i++) */

    return next_ugly_no;

}
```

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

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

Algorithmic Paradigm: Dynamic Programming
Time Complexity: O(n)
Storage Complexity: O(n)
Please write comments if you find any bug in the above program or other ways to solve the same problem.
++

Largest Sum Contiguous Subarray

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

Kadane's Algorithm:

```
Initialize:

    max_so_far = 0

    max_ending_here = 0


Loop for each element of the array

  (a) max_ending_here = max_ending_here + a[i]

  (b) if(max_ending_here < 0)

            max_ending_here = 0

  (c) if(max_so_far < max_ending_here)

            max_so_far = max_ending_here

return max_so_far
```

Explanation:

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

Lets take the example:
{-2, -3, 4, -1, -2, 1, 5, -3}
max_so_far = max_ending_here = 0
for i=0, a[0] = -2
max_ending_here = max_ending_here + (-2)
Set max_ending_here = 0 because max_ending_here < 0
for i=1, a[1] = -3
max_ending_here = max_ending_here + (-3)
Set max_ending_here = 0 because max_ending_here < 0
for i=2, a[2] = 4
max_ending_here = max_ending_here + (4)
max_ending_here = 4
max_so_far is updated to 4 because max_ending_here greater than max_so_far which was 0 till now
for i=3, a[3] = -1
max_ending_here = max_ending_here + (-1)
max_ending_here = 3
for i=4, a[4] = -2
max_ending_here = max_ending_here + (-2)
max_ending_here = 1
for i=5, a[5] = 1
max_ending_here = max_ending_here + (1)
max_ending_here = 2
for i=6, a[6] = 5
max_ending_here = max_ending_here + (5)
max_ending_here = 7
max_so_far is updated to 7 because max_ending_here is greater than max_so_far
for i=7, a[7] = -3

max_ending_here = max_ending_here + (-3)
max_ending_here = 4
Program:

```
#include<stdio.h>
int maxSubArraySum(int a[], int size)
{
   int max_so_far = 0, max_ending_here = 0;
   int i;
   for(i = 0; i < size; i++)
   {
     max_ending_here = max_ending_here + a[i];
     if(max_ending_here < 0)
        max_ending_here = 0;
     if(max_so_far < max_ending_here)
        max_so_far = max_ending_here;
    }
    return max_so_far;
}

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

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

0.

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

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

Time Complexity: O(n)
Algorithmic Paradigm: Dynamic Programming
Following is another simple implementation suggested by **Mohit Kumar**. The implementation handles the case when all numbers in array are negative.

```
#include<stdio.h>

int max(int x, int y)
{ return (y > x)? y : x; }

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

   for (i = 1; i < size; i++)
```

```
    {
        curr_max = max(a[i], curr_max+a[i]);
        max_so_far = max(max_so_far, curr_max);
    }
    return max_so_far;
}

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

Now try below question
Given an array of integers (possibly some of the elements negative), write a C program to find out the *maximum product* possible by adding 'n' consecutive integers in the array, n <= ARRAY_SIZE. Also give where in the array this sequence of n integers starts.

References:

http://en.wikipedia.org/wiki/Kadane%27s_Algorithm

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

++