**Minimum insertions to form a palindrome**

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’.

TC : O(n^2)

SC : O(n) // just use two 1D array

**Count number of ways to reach a given score in a game**

Consider a game where a player can score 3 or 5 or 10 points in a move. Given a total score n, find number of ways to reach the given score.

Examples:

Input: n = 20

Output: 4

There are following 4 ways to reach 20

(10, 10)

(5, 5, 10)

(5, 5, 5, 5)

(3, 3, 3, 3, 3, 5)

**Soln :**

table[0] = 1; // and set other elements to 0

// One by one consider given 3 moves and update the table[]

// values after the index greater than or equal to the

// value of the picked move

for (i=3; i<=n; i++)

table[i] += table[i-3];

for (i=5; i<=n; i++)

table[i] += table[i-5];

for (i=10; i<=n; i++)

table[i] += table[i-10];

return table[n];

# 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 that ***from each cell you can either move only to right or down.***

**Soln. :**

**For first row.. dp[0][j]=1**

**Similarily for first column.. dp[i][0]=1**

**Else**

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

**TC : O(mn)**

**SC : O(n) as only count[2][n] is enough because we only need previous row to compute current row values.**

# Count number of binary strings without consecutive 1’s

**Input: N = 3**

**Output: 5**

**// The 5 strings are 000, 001, 010, 100, 101**

Let a[i] be the number of binary strings of length i which do not contain any two consecutive 1’s and which end in 0.

Similarly, let b[i] be the number of such strings which end in 1.

We can append either 0 or 1 to a string ending in 0, but we can only append 0 to a string ending in 1.

This yields the recurrence relation:

a[i] = a[i - 1] + b[i - 1]

b[i] = a[i - 1]

Base : a[1] = b[1] = 1

This is following Fibonacci pattern.

Here answer is **Fib(n+2)**

TC : O(logn)

# Count possible ways to construct buildings

Given an input number of sections and each section has 2 plots on either sides of the road. Find all possible ways to construct buildings in the plots such that there is a space between any 2 buildings.

N = 3

Output = 25

3 sections, which means possible ways for one side are

BSS, BSB, SSS, SBS, SSB where B represents a building

and S represents an empty space

Total possible ways are 25, because a way to place on

one side can correspond to any of 5 ways on other side.

**The problem is the same finding number of binary strings which does not contain consecutive 1’s. (Here buildings)**

**Find that and square it because the same combination can happen on the other side.**

**Sol.**

**Fib(n+2)** ways are possible such that no buildings will be consecutive.

Here we need to square it so fib(n+2)^2.

TC : (logn) as finding Fibonacci number takes log n.

# Minimum number of squares whose sum equals to given number n

Same as finding min coins to make the sum. But here consider the value as square.

Update the min count dp[i] = min(dp[i], 1 + dp[i-x\*x]) where x loops through 1 to [sqrt(i)]

TC : O(n\*sqrt(n))

SC : O(n)

# Find if string is K-Palindrome or notsad

# Given a string, find out if the string is K-Palindrome or not. A K-palindrome string transforms into a palindrome on removing at most k characters from it.

abcdecba, k = 1

**Output :** Yes

String can become palindrome by removing

1 character i.e. either d or e

The idea is to find the longest palindromic subsequence of the given string.

If the difference between longest palindromic subsequence and the original string is less than equal to k, then the string is k-palindrome else it is not k-palindrome.

TC : O(n^2)

SC : O(n) // optimized space using only 2 1D array

**Longest Repeating Subsequence**

Given a string, find length of the longest repeating subseequence such that the two subsequence don’t have same string character at same position, i.e., any i’th character in the two subsequences shouldn’t have the same index in the original string.

Input: str = "aab"

Output: 1

The two subssequence are 'a'(first) and 'a'(second).

Note that 'b' cannot be considered as part of subsequence

as it would be at same index in both.

Input: str = "aabb"

Output: 2

The idea is to find the LCS(str, str) where str is the input string with the restriction that when both the characters are same, they shouldn’t be on the same index in the two strings.

if (str[i-1] == str[j-1] && i!=j)

   dp[i][j] =  1 + dp[i-1][j-1];

else // If characters do not match

   dp[i][j] = max(dp[i][j-1], dp[i-1][j]);

# Printing Longest Common Subsequence (Printing All)

Following is detailed algorithm to print the all LCS.

We construct L[m+1][n+1] table as discussed in the [previous](http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/) post and traverse the 2D array starting from L[m][n]. For current cell L[i][j] in the matrix,

1. If the last characters of X and Y are same (i.e. X[i-1] == Y[j-1]), then the charcater must be present in all LCS of substring X[0…i-1] and Y[0..j-1].

We simply recurse for L[i-1][j-1] in the matrix and append current character to all LCS possible of substring X[0…i-2] and Y[0..j-2].

1. If the last characters of X and Y are not same (i.e. X[i-1] != Y[j-1]), then LCS can be constructed from either top side of the matrix (i.e. L[i-1][j]) or from left side of matrix (i.e. L[i][j-1]) depending upon which value is greater.

If both the values are equal(i.e. L[i-1][j] == L[i][j-1]), then it will be constructed from both sides of matrix. So based on values at L[i-1][j] and L[i][j-1], we go in direction of greater value or go in both directions if the values are equal.

Check the code once at <http://www.geeksforgeeks.org/printing-longest-common-subsequence-set-2-printing/>

**Find maximum length Snake sequence**

Given a grid of numbers, find maximum length Snake sequence and print it. If multiple snake sequences exists with the maximum length, print any one of them.

A snake sequence is made up of adjacent numbers in the grid such that for each number, the number on the right or the number below it is +1 or -1 its value. For example, if you are at location (x, y) in the grid, you can either move right i.e. (x, y+1) if that number is ± 1 or move down i.e. (x+1, y) if that number is ± 1.

**9**,6,5,2  
**8,7,6,5**  
7,3,1, **6**  
1, 1, 1, **7**

In above grid, the longest snake sequence is: (9, 8, 7, 6, 5, 6, 7)

Solution is simple.

Let T[i][i] represent maximum length of a snake which ends at cell (i, j), then for given matrix M, the DP relation is defined as –

T[0][0] = 0

T[i][j] = max(T[i][j], T[i][j – 1] + 1) if M[i][j] = M[i][j – 1] ± 1

T[i][j] = max(T[i][j], T[i – 1][j] + 1) if M[i][j] = M[i – 1][j] ± 1

TC : O(mn)

SC : O(n) (use only 2 1D array and use it alternatively.)

For printing, store (i,j) where max was found and backtrace the cells and print the sequence accordingly.

**Permutation Coefficient**

Permutation refers to the process of arranging all the members of a given set to form a sequence. The number of permutations on a set of n elements is given by n! , where “!” represents factorial.  
The **Permutation Coefficient** represented by P(n, k) is used to represent the number of ways to obtain an ordered subset having k elements from a set of n elements.

Mathematically it’s given as:  
[permu](http://www.test.geeksforgeeks.org/wp-content/uploads/permu.png)

Examples:

P(10, 2) = 90 i.e. 10\*9

P(10, 3) = 720 i.e. 10\*9\*8

P(10, 0) = 1

P(10, 1) = 10

for(int i=n,j=k;j>0;j--,i--) ans=ans\*i; (initially ans=1)

**High-effort vs. Low-effort Tasks Problem**

You are given n days and for each day (di) you could either perform a high effort tasks (hi) or a low effort tasks (li) or no task with the constraint that you can choose a high-effort tasks only if you chose no task on the previous day. Write a program to find the maximum amount of tasks you can perform within these n days.

No. of days (n) = 5

Day L.E. H.E

1 1 3

2 5 6

3 4 8

4 5 7

5 3 6

Maximum amount of tasks

= 3 + 5 + 4 + 5 + 3

= 20

int task\_dp[n+1];

// If n = 0, no solution exists

task\_dp[0] = 0;

// If n = 1, high effort task on that day will be the solution

task\_dp[1] = high[0];

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

task\_dp[i] = max(high[i-1] + task\_dp[i-2],low[i-1] + task\_dp[i-1]);

TC : O(n)

SC : O(n)