1. 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.

Given a number n, the task is to find n'th Ugly number.

```
Input  : n = 7
Output : 8
Input  : n = 10
Output : 12
Input  : n = 15
Output : 24
Input  : n = 150
Output : 5832
```

Ans:-

```
# include<stdio.h>

# include<stdlib.h>

int maxDivide(int a, int b)

{

while (a%b == 0)

a = a/b;

return a;

}

int isUgly(int no)

{

no = maxDivide(no, 2);

no = maxDivide(no, 3);

no = maxDivide(no, 5); return (no == 1)? 1 : 0; }
```

```
int getNthUglyNo(int n)
{
int i = 1;
int count = 1;
while (n > count)
{
        i++;
        if (isUgly(i))
        count++;
}
return i;
}
int main()
{
        unsigned no = getNthUglyNo(150);
        printf("150th ugly no. is %d ", no);
        getchar();
        return 0;
}
```

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

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

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

```
//Fibonacci Series using Recursion

#include<bits/stdc++.h>

using namespace std;

int fib(int n)

{

	if (n <= 1)

		return n;

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

}

int main ()

{

	int n = 9;

	cout << fib(n);

	getchar();

	return 0;

}
```

3. 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.
4) Given a number n, return the number of ways you can draw n chords in a circle with 2 x n points such that no 2 chords intersect.
See this for more applications.
The first few Catalan numbers for n = 0, 1, 2, 3, … are **1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862,..**

```
#include<iostream>

using namespace std;

unsigned long int catalan(unsigned int n)

{

    if (n <= 1) return 1;

    unsigned long int res = 0;

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

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

    return res;

}

int main()

{

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

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

    return 0;

}
```

4. 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.

```
#include <bits/stdc++.h>
using namespace std;
int binomialCoeff(int n, int k)
{
	if (k == 0 || k == n)
		return 1;
	return binomialCoeff(n - 1, k - 1) + binomialCoeff(n - 1, k);
}
int main()
{
	int n = 5, k = 2;
	cout << "Value of C("<<n<<", "<<k<<") is " << binomialCoeff(n, k);
	return 0;
}
```

5. 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:

$$P(n,k) = \underbrace{n.(n-1).(n-2)...(n-k+1)}_{k \text{ factors}},$$

which is 0 when $k > n$, and otherwise equal to

$$\frac{n!}{(n-k)!}$$

```
#include<bits/stdc++.h>

int permutationCoeff(int n, int k) {

	int P[n + 1][k + 1];

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

	{

	for (int j = 0; j <= std::min(i, k); j++)

		{
```

```
            if (j == 0)
                P[i][j] = 1;
            else
                P[i][j] = P[i - 1][j] + (j * P[i - 1][j - 1]);
            P[i][j + 1] = 0;
        }
    }
    return P[n][k];
}
int main()
{
    int n = 10, k = 2;
    printf("Value of P(%d, %d) is %d ", n, k, permutationCoeff(n, k));
    return 0;
}
```

6. Given a gold mine of n*m dimensions. Each field in this mine contains a positive integer which is the amount of gold in tons. Initially the miner is at first column but can be at any row. He can move only (right->,right up /,right down\) that is from a given cell, the miner can move to the cell diagonally up towards the right or right or diagonally down towards the right. Find out maximum amount of gold he can collect.

Examples:

```
Input : mat[][] = {{1, 3, 3},
                   {2, 1, 4},
                  {0, 6, 4}};
Output : 12
{(1,0)->(2,1)->(2,2)}

Input: mat[][] = { {1, 3, 1, 5},
```

```
                    {2, 2, 4, 1},
                    {5, 0, 2, 3},
                    {0, 6, 1, 2}};
Output : 16
(2,0) -> (1,1) -> (1,2) -> (0,3) OR
(2,0) -> (3,1) -> (2,2) -> (2,3)

Input : mat[][] = {{10, 33, 13, 15},
                   {22, 21, 04, 1},
                   {5, 0, 2, 3},
                   {0, 6, 14, 2}};
Output : 83
```

```
#include<bits/stdc++.h>
using namespace std;
const int MAX = 100;
int getMaxGold(int gold[][MAX], int m, int n)
{
    int goldTable[m][n];
    memset(goldTable, 0, sizeof(goldTable));
    for (int col=n-1; col>=0; col--)
    {
        for (int row=0; row<m; row++)
        {
            int right = (col==n-1)? 0: goldTable[row][col+1];
            int right_up = (row==0 || col==n-1)? 0: goldTable[row-1][col+1];
            int right_down = (row==m-1 || col==n-1)? 0: goldTable[row+1][col+1];
```

```
                        goldTable[row][col] = gold[row][col] + max(right, max(right_up,
right_down));
                }
        }
        int res = goldTable[0][0];
        for (int i=1; i<m; i++)
                res = max(res, goldTable[i][0]);
        return res;
}
int main()
{
        int gold[MAX][MAX]= { {1, 3, 1, 5},
                {2, 2, 4, 1},
                {5, 0, 2, 3},
                {0, 6, 1, 2}
        };
        int m = 4, n = 4;
        cout << getMaxGold(gold, m, n);
        return 0; }
```

7. Coin Change | DP-7

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.

```
#include<stdio.h>
int count( int S[], int m, int n )
{
	if (n == 0)
		return 1;
	if (n < 0)
		return 0;
	if (m <=0 && n >= 1)
		return 0;
	return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}
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;
}
```

8. Friends Pairing Problem

Last Updated: 16-05-2019

Given n friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.

```
Input  : n = 3
Output : 4
```

```
Explanation
{1}, {2}, {3} : all single
{1}, {2, 3} : 2 and 3 paired but 1 is single.
{1, 2}, {3} : 1 and 2 are paired but 3 is single.
{1, 3}, {2} : 1 and 3 are paired but 2 is single.
Note that {1, 2} and {2, 1} are considered same.
```

```
#include <bits/stdc++.h>
using namespace std;
int countFriendsPairings(int n)
{
	int dp[n + 1];
	for (int i = 0; i <= n; i++) {
		if (i <= 2)
			dp[i] = i;
		else
			dp[i] = dp[i - 1] + (i - 1) * dp[i - 2];
	}

	return dp[n];
}

int main()
{
	int n = 4;
	cout << countFriendsPairings(n) << endl;
	return 0;
}
```

9. Subset Sum Problem | DP-25

Last Updated: 25-08-2020

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*.

Example:

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

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
There is no subset that add up to 30.
```

```
bool isSubsetSum(int set[], int n, int sum)
{
    if (sum == 0)
        return true;
    if (n == 0)
        return false;
    if (set[n - 1] > sum)
        return isSubsetSum(set, n - 1, sum);
    return isSubsetSum(set, n - 1, sum)
           || isSubsetSum(set, n - 1, sum - set[n - 1]);
}

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;
}
```

10. Subset Sum Problem in O(sum) space

Given an array 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:

```
Input : arr[] = {4, 1, 10, 12, 5, 2},
          sum = 9

Output : TRUE

{4, 5} is a subset with sum 9.
```

```
Input : arr[] = {1, 8, 2, 5},
          sum = 4
Output : FALSE
There exists no subset with sum 4.
```

```
bool isSubsetSum(int arr[], int n, int sum)
{

    bool subset[2][sum + 1];
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= sum; j++) {
            if (j == 0)
                subset[i % 2][j] = true;
            else if (i == 0)
                subset[i % 2][j] = false;
            else if (arr[i - 1] <= j)
                subset[i % 2][j] = subset[(i + 1) % 2]
             [j - arr[i - 1]] || subset[(i + 1) % 2][j];
            else
                subset[i % 2][j] = subset[(i + 1) % 2][j];
        }
    }

    return subset[n % 2][sum];
}
 int main()
{
    int arr[] = { 6, 2, 5 };
    int sum = 7;
    int n = sizeof(arr) / sizeof(arr[0]);
    if (isSubsetSum(arr, n, sum) == true)
        printf("There exists a subset with given sum");
    else
        printf("No subset exists with given sum");
    return 0;
}
```

11. Subset with sum divisible by m

Last Updated: 26-12-2018
Given a set of non-negative distinct integers, and a value m, determine if there is a subset of the given set with sum divisible by m.
Input Constraints
Size of set i.e., n <= 1000000, m <= 1000

Examples:

```
Input : arr[] = {3, 1, 7, 5};
        m = 6;
```

```
Output : YES


Input : arr[] = {1, 6};
        m = 5;
Output : NO
```

```
bool modularSum(int arr[], int n, int m)
{
    if (n > m)
        return true;
    bool DP[m];
    memset(DP, false, m);
    for (int i=0; i<n; i++)
    {
        if (DP[0])
            return true;
        bool temp[m];
        memset(temp,false,m);
        for (int j=0; j<m; j++)
        {
            if (DP[j] == true)
            {
                if (DP[(j+arr[i]) % m] == false)
                    temp[(j+arr[i]) % m] = true;
            }
        }
        for (int j=0; j<m; j++)
            if (temp[j])
                DP[j] = true;
        DP[arr[i]%m] = true;
    }

    return DP[0];
}

int main()
{
    int arr[] = {1, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    int m = 5;

    modularSum(arr, n, m) ?  cout << "YES\n" : cout << "NO\n";

    return 0;
}
```

12. Largest divisible pairs subset

Given an array of n distinct elements, find length of the largest subset such that every pair in the subset is such that the larger element of the pair is divisible by smaller

element.
Examples:

```
Input : arr[] = {10, 5, 3, 15, 20}

Output : 3

Explanation: The largest subset is 10, 5, 20.

10 is divisible by 5, and 20 is divisible by 10.


Input : arr[] = {18, 1, 3, 6, 13, 17}

Output : 4

Explanation: The largest subset is 18, 1, 3, 6,

In the subsequence, 3 is divisible by 1,

6 by 3 and 18 by 6.
```

```
#include <bits/stdc++.h>
using namespace std;

// function to find the longest Subsequence
int largestSubset(int a[], int n)
{
    // dp[i] is going to store size of largest
    // divisible subset beginning with a[i].
    int dp[n];

    // Since last element is largest, d[n-1] is 1
    dp[n - 1] = 1;

    // Fill values for smaller elements.
    for (int i = n - 2; i >= 0; i--) {

        // Find all multiples of a[i] and consider
        // the multiple that has largest subset
        // beginning with it.
        int mxm = 0;
        for (int j = i + 1; j < n; j++)
            if (a[j] % a[i] == 0 || a[i] % a[j] == 0)
                mxm = max(mxm, dp[j]);

        dp[i] = 1 + mxm;
    }

    // Return maximum value from dp[]
    return *max_element(dp, dp + n);
}

// driver code to check the above function
int main()
```

```
{
    int a[] = { 1, 3, 6, 13, 17, 18 };
    int n = sizeof(a) / sizeof(a[0]);
    cout << largestSubset(a, n) << endl;
    return 0;
}
```

13. Perfect Sum Problem (Print all subsets with given sum)

Given an array of integers and a sum, the task is to print all subsets of given array with sum equal to given sum.

Examples:

```
Input : arr[] = {2, 3, 5, 6, 8, 10}
        sum = 10
Output : 5 2 3
         2 8
         10


Input : arr[] = {1, 2, 3, 4, 5}
        sum = 10
Output : 4 3 2 1
         5 3 2
         5 4 1
```

```
#include <bits/stdc++.h>
using namespace std;

// dp[i][j] is going to store true if sum j is
// possible with array elements from 0 to i.
bool** dp;

void display(const vector<int>& v)
{
    for (int i = 0; i < v.size(); ++i)
        printf("%d ", v[i]);
    printf("\n");
}

// A recursive function to print all subsets with the
// help of dp[][]. Vector p[] stores current subset.
void printSubsetsRec(int arr[], int i, int sum, vector<int>& p)
```

```
{
    // If we reached end and sum is non-zero. We print
    // p[] only if arr[0] is equal to sun OR dp[0][sum]
    // is true.
    if (i == 0 && sum != 0 && dp[0][sum])
    {
        p.push_back(arr[i]);
        display(p);
        return;
    }

    // If sum becomes 0
    if (i == 0 && sum == 0)
    {
        display(p);
        return;
    }

    // If given sum can be achieved after ignoring
    // current element.
    if (dp[i-1][sum])
    {
        // Create a new vector to store path
        vector<int> b = p;
        printSubsetsRec(arr, i-1, sum, b);
    }

    // If given sum can be achieved after considering
    // current element.
    if (sum >= arr[i] && dp[i-1][sum-arr[i]])
    {
        p.push_back(arr[i]);
        printSubsetsRec(arr, i-1, sum-arr[i], p);
    }
}

// Prints all subsets of arr[0..n-1] with sum 0.
void printAllSubsets(int arr[], int n, int sum)
{
    if (n == 0 || sum < 0)
       return;

    // Sum 0 can always be achieved with 0 elements
    dp = new bool*[n];
    for (int i=0; i<n; ++i)
    {
        dp[i] = new bool[sum + 1];
        dp[i][0] = true;
    }

    // Sum arr[0] can be achieved with single element
    if (arr[0] <= sum)
       dp[0][arr[0]] = true;

    // Fill rest of the entries in dp[][]
```

```
    for (int i = 1; i < n; ++i)
        for (int j = 0; j < sum + 1; ++j)
            dp[i][j] = (arr[i] <= j) ? dp[i-1][j] ||
                                       dp[i-1][j-arr[i]]
                                     : dp[i - 1][j];
    if (dp[n-1][sum] == false)
    {
        printf("There are no subsets with sum %d\n", sum);
        return;
    }

    // Now recursively traverse dp[][] to find all
    // paths from dp[n-1][sum]
    vector<int> p;
    printSubsetsRec(arr, n-1, sum, p);
}

// Driver code
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    int sum = 10;
    printAllSubsets(arr, n, sum);
    return 0;
}
```

14. Compute nCr % p | Set 1 (Introduction and Dynamic Programming Solution)

Last Updated: 09-04-2020

Given three numbers n, r and p, compute value of ${}^{n}C_{r}$ mod p.

Example:

```
Input:  n = 10, r = 2, p = 13

Output: 6

Explanation: 10C2 is 45 and 45 % 13 is 6.
```

```
// A Dynamic Programming based solution to compute nCr % p
#include <bits/stdc++.h>
using namespace std;

// Returns nCr % p
int nCrModp(int n, int r, int p)
{
    // Optimization for the cases when r is large
    if (r > n - r)
        r = n - r;

    // The array C is going to store last row of
    // pascal triangle at the end. And last entry
```

```
    // of last row is nCr
    int C[r + 1];
    memset(C, 0, sizeof(C));

    C[0] = 1; // Top row of Pascal Triangle

    // One by constructs remaining rows of Pascal
    // Triangle from top to bottom
    for (int i = 1; i <= n; i++) {

        // Fill entries of current row using previous
        // row values
        for (int j = min(i, r); j > 0; j--)

            // nCj = (n-1)Cj + (n-1)C(j-1);
            C[j] = (C[j] + C[j - 1]) % p;
    }
    return C[r];
}

// Driver program
int main()
{
    int n = 10, r = 2, p = 13;
    cout << "Value of nCr % p is " << nCrModp(n, r, p);
    return 0;
}
```

15. Choice of Area

Consider a game, in which you have two types of powers, A and B and there are 3 types of Areas X, Y and Z. Every second you have to switch between these areas, each area has specific properties by which your power A and power B increase or decrease. We need to keep choosing areas in such a way that our survival time is maximized. Survival time ends when any of the powers, A or B reaches less than 0.
Examples:

```
Initial value of Power A = 20

Initial value of Power B = 8


Area X (3, 2) : If you step into Area X,

                A increases by 3,

                B increases by 2


Area Y (-5, -10) : If you step into Area Y,

                   A decreases by 5,
```

B decreases by 10

Area Z (-20, 5) : If you step into Area Z,
A decreases by 20,
B increases by 5

It is possible to choose any area in our first step.
We can survive at max 5 unit of time by following
these choice of areas :
X -> Z -> X -> Y -> X

```
//  C++ code to get maximum survival time
#include <bits/stdc++.h>
using namespace std;

//  structure to represent an area
struct area
{
    //  increment or decrement in A and B
    int a, b;
    area(int a, int b) : a(a), b(b)
    {}
};

//  Utility method to get maximum of 3 integers
int max(int a, int b, int c)
{
    return max(a, max(b, c));
}

//  Utility method to get maximum survival time
int maxSurvival(int A, int B, area X, area Y, area Z,
                int last, map<pair<int, int>, int>& memo)
{
    //  if any of A or B is less than 0, return 0
    if (A <= 0 || B <= 0)
        return 0;
    pair<int, int> cur = make_pair(A, B);

    //  if already calculated, return calculated value
    if (memo.find(cur) != memo.end())
        return memo[cur];

    int temp;

    //  step to areas on basis of last chose area
    switch(last)
    {
```

```
    case 1:
        temp = 1 + max(maxSurvival(A + Y.a, B + Y.b,X, Y, Z, 2, memo),
                       maxSurvival(A + Z.a, B + Z.b,
                                   X, Y, Z, 3, memo));
        break;
    case 2:
        temp = 1 + max(maxSurvival(A + X.a, B + X.b,
                                   X, Y, Z, 1, memo),
                       maxSurvival(A + Z.a, B + Z.b,
                                   X, Y, Z, 3, memo));
        break;
    case 3:
        temp = 1 + max(maxSurvival(A + X.a, B + X.b,
                                   X, Y, Z, 1, memo),
                       maxSurvival(A + Y.a, B + Y.b,
                                   X, Y, Z, 2, memo));
        break;
    }

    //  store the result into map
    memo[cur] = temp;

    return temp;
}

//  method returns maximum survival time
int getMaxSurvivalTime(int A, int B, area X, area Y, area Z)
{
    if (A <= 0 || B <= 0)
        return 0;
    map< pair<int, int>, int > memo;

    //  At first, we can step into any of the area
    return
        max(maxSurvival(A + X.a, B + X.b, X, Y, Z, 1, memo),
            maxSurvival(A + Y.a, B + Y.b, X, Y, Z, 2, memo),
            maxSurvival(A + Z.a, B + Z.b, X, Y, Z, 3, memo));
}

//  Driver code to test above method
int main()
{
    area X(3, 2);
    area Y(-5, -10);
    area Z(-20, 5);

    int A = 20;
    int B = 8;
    cout << getMaxSurvivalTime(A, B, X, Y, Z);

    return 0;
}
```

17. Cutting a Rod | DP-13

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
```

```
// 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 %dn", cutRod(arr, size));
    getchar();  return 0; }
```

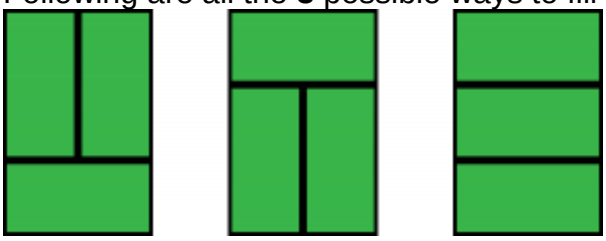
18. Tiling with Dominoes

Last Updated: 05-06-2018

Given a *3 x n* board, find the number of ways to fill it with *2 x 1* dominoes.

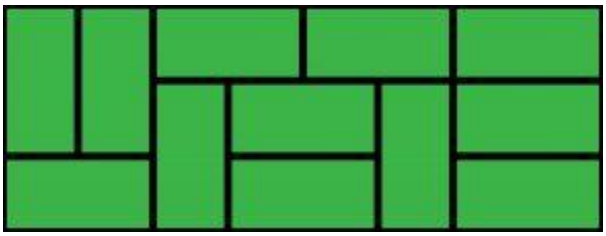
Example 1

Following are all the **3** possible ways to fill up a **3 x 2** board.



Example 2

Here is one possible way of filling a 3 x 8 board. You have to find all the possible ways to do so.



Examples :

```
Input : 2

Output : 3


Input : 8

Output : 153


Input : 12

Output : 2131
```

```
// C++ program to find no. of ways
// to fill a 3xn board with 2x1 dominoes.
#include <iostream>
using namespace std;

int countWays(int n)
{
    int A[n + 1], B[n + 1];
    A[0] = 1, A[1] = 0, B[0] = 0, B[1] = 1;
    for (int i = 2; i <= n; i++) {
        A[i] = A[i - 2] + 2 * B[i - 1];
        B[i] = A[i - 1] + B[i - 2];
```

```
    }

    return A[n];
}

int main()
{
    int n = 8;
    cout << countWays(n);
    return 0;
}
```

19. Newman–Shanks–Williams prime

In mathematics, a **Newman–Shanks–Williams prime** (NSW prime) is a prime number p which can be written in the form:
Recurrence relation for Newman–Shanks–Williams prime is:
Sn=2*Sn-1+Sn-2
The first few terms of the sequence are 1, 1, 3, 7, 17, 41, 99, ….
Examples:

```
Input : n = 3

Output : 7


Input : n = 4

Output : 17
```

```
 // CPP Program to find Newman-Shanks-Williams prime

#include <bits/stdc++.h>
using namespace std;

// return nth Newman-Shanks-Williams prime
int nswp(int n)
{
    // Base case
    if (n == 0 || n == 1)
        return 1;

    // Recursive step
    return 2 * nswp(n - 1) + nswp(n - 2);
}

// Driven Program
int main()
{
    int n = 3;

    cout << nswp(n) << endl;
    return 0; }
```

20. Golomb sequence

In mathematics, the **Golomb sequence** is a non-decreasing integer sequence where n-th term is equal to number of times n appears in the sequence.
The first few values are
1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, ……

Explanation of few terms:
Third term is 2, note that three appears 2 times.
Second term is 2, note that two appears 2 times.
Fourth term is 3, note that four appears 3 times.

Given a positive integer n. The task is to find the first n terms of Golomb sequence.

Examples :

```
Input : n = 4

Output : 1 2 2 3



Input : n = 6

Output : 1 2 2 3 3 4
```

```
/ C++ Program to find first
// n terms of Golomb sequence.
#include <bits/stdc++.h>
using namespace std;

// Return the nth element
// of Golomb sequence
int findGolomb(int n)
{
    // base case
    if (n == 1)
        return 1;

    // Recursive Step
    return 1 + findGolomb(n -findGolomb(findGolomb(n - 1)));
}

// Print the first n
// term of Golomb Sequence
void printGolomb(int n)
{
    // Finding first n
    // terms of Golomb Sequence.
    for (int i = 1; i <= n; i++)
        cout << findGolomb(i) << " ";
}

int main()
{
```

```
    int n = 9;
    printGolomb(n);
    return 0;
}
```

21. Moser-de Bruijn Sequence

Given an integer ‘n’, print the first ‘n’ terms of the Moser-de Bruijn Sequence.

The **Moser-de Bruijn sequence** is the sequence obtained by adding up the distinct powers of the number 4 (For example 1, 4, 16, 64, etc).

Examples :

```
Input : 5
Output : 0 1 4 5 16

Input : 10
Output : 0 1 4 5 16 17 20 21 64 65
```

```
1) S(2 * n) = 4 * S(n)
2) S(2 * n + 1) = 4 * S(n) + 1
with S(0) = 0 and S(1) = 1
```

```
/ CPP code to generate first 'n' terms
// of the Moser-de Bruijn Sequence
#include <bits/stdc++.h>
using namespace std;

// Function to generate nth term
// of Moser-de Bruijn Sequence
int gen(int n)
{
    // S(0) = 0
    if (n == 0)
        return 0;

    // S(1) = 1
    else if (n == 1)
        return 1;

    // S(2 * n) = 4 * S(n)
    else if (n % 2 == 0)
        return 4 * gen(n / 2);

    // S(2 * n + 1) = 4 * S(n) + 1
    else if (n % 2 == 1)
        return 4 * gen(n / 2) + 1;
}

// Generating the first 'n' terms
// of Moser-de Bruijn Sequence
void moserDeBruijn(int n)
{
```

```
    for (int i = 0; i < n; i++)
        cout << gen(i) << " ";
    cout << "\n";
}

// Driver Code
int main()
{
    int n = 15;
    cout << "First " << n << " terms of "
         << "Moser-de Bruijn Sequence : \n";
    moserDeBruijn(n);
    return 0;
}
```

22. Newman-Conway Sequence

Newman-Conway Sequence is the one which generates the following integer sequence.
1 1 2 2 3 4 4 4 5 6 7 7…

In mathematical terms, the sequence P(n) of Newman-Conway numbers is defined by recurrence relation

```
P(n) = P(P(n - 1)) + P(n - P(n - 1))
```

with seed values P(1) = 1 and P(2) = 1

Given a number n, print n-th number in Newman-Conway Sequence.

```
// C++ program for n-th
// element of Newman-Conway Sequence
#include <bits/stdc++.h>
using namespace std;

// Recursive Function to find the n-th element
int sequence(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return sequence(sequence(n - 1))
                + sequence(n - sequence(n - 1));
}

// Driver Program
int main()
{
    int n = 10;
    cout << sequence(n);
    return 0;

}
```

23. Print n terms of Newman-Conway Sequence

Newman-Conway numbers is the one which generates the following integer sequence.
1 1 2 2 3 4 4 4 5 6 7 7….. and follows below recursive formula.

```
P(n) = P(P(n - 1)) + P(n - P(n - 1))
```

Given a number n then print n terms of Newman-Conway Sequence

Examples:

```
Input : 13

Output : 1 1 2 2 3 4 4 4 5 6 7 7 8


Input : 20

Output : 1 1 2 2 3 4 4 4 5 6 7 7 8 8 8 8 9 10 11 12
```

```
// C++ Program to print n terms
// of Newman-Conway Sequence
#include <bits/stdc++.h>
using namespace std;

// Function to find
// the n-th element
void sequence(int n)
{
    // Declare array to store sequence
    int f[n + 1];
    f[0] = 0;
    f[1] = 1;
    f[2] = 1;

    cout << f[1] << " " << f[2] << " ";

    for (int i = 3; i <= n; i++) {
        f[i] = f[f[i - 1]] + f[i - f[i - 1]];
        cout << f[i] << " ";
    }
}

// Driver Program
int main()
{
    int n = 13;
    sequence(n);
    return 0;
}
```

24. Print Fibonacci sequence using 2 variables

Print the Fibonacci sequence. The first Fibonacci numbers are:
0,1,1,2,3,5,8,13,21,34,55,89,144

```
// Simple CPP Program to print Fibonacci
// sequence
#include <iostream>
using std::cout;
void fib(int n)
{
    int a = 0, b = 1, c;
    if (n >= 0)
        cout << a << " ";
    if (n >= 1)
        cout << b << " ";
    for (int i = 2; i <= n; i++) {
        c = a + b;
        cout << c << " ";
        a = b;
        b = c;
    }
}

// Driver code
int main()
{
    fib(9);
    return 0;
}
```

25. Print Fibonacci Series in reverse order

Given a number n then print n terms of fibonacci series in reverse order.

Examples:

```
Input : n = 5

Output : 3 2 1 1 0


Input : n = 8

Output : 13 8 5 3 2 1 1 0
```

```
// CPP Program to print Fibonacci
// series in reverse order
#include <bits/stdc++.h>
using namespace std;

void reverseFibonacci(int n)
{
```

```
    int a[n];

    // assigning first and second elements
    a[0] = 0;
    a[1] = 1;

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

        // storing sum in the
        // preceding location
        a[i] = a[i - 2] + a[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {

        // printing array in
        // reverse order
        cout << a[i] << " ";
    }
}

// Driver function
int main()
{
    int n = 5;
    reverseFibonacci(n);
    return 0;
}
```

26. Count even length binary sequences with same sum of first and second half bits

Given a number n, find count of all binary sequences of length 2n such that sum of first n bits is same as sum of last n bits.

Examples:

```
Input:  n = 1

Output: 2

There are 2 sequences of length 2*n, the

sequences are 00 and 11


Input:  n = 2

Output: 6

There are 6 sequences of length 2*n, the

sequences are 0101, 0110, 1010, 1001, 0000
```

and 1111

```
// A Naive Recursive C++ program to count even
// length binary sequences such that the sum of
// first and second half bits is same
#include<bits/stdc++.h>
using namespace std;

// diff is difference between sums first n bits
// and last n bits respectively
int countSeq(int n, int diff)
{
    // We can't cover difference of more
    // than n with 2n bits
    if (abs(diff) > n)
        return 0;

    // n == 1, i.e., 2 bit long sequences
    if (n == 1 && diff == 0)
        return 2;
    if (n == 1 && abs(diff) == 1)
        return 1;

    int res = // First bit is 0 & last bit is 1
              countSeq(n-1, diff+1) +

              // First and last bits are same
              2*countSeq(n-1, diff) +

              // First bit is 1 & last bit is 0
              countSeq(n-1, diff-1);

    return res;
}

// Driver program
int main()
{
    int n = 2;
    cout << "Count of sequences is "
         << countSeq(2, 0);
    return 0;
}
```

27. Sequences of given length where every element is more than or equal to twice of previous

Last Updated: 21-04-2020

Given two integers m & n, find the number of possible sequences of length n such that each of the next element is greater than or equal to twice of the previous element but less than or equal to m.

Examples :

```
Input : m = 10, n = 4

Output : 4

There should be n elements and value of last

element should be at-most m.

The sequences are {1, 2, 4, 8}, {1, 2, 4, 9},

                 {1, 2, 4, 10}, {1, 2, 5, 10}


Input : m = 5, n = 2

Output : 6

The sequences are {1, 2}, {1, 3}, {1, 4},

                  {1, 5}, {2, 4}, {2, 5}
```

```
// C program to count total number of special sequences
// of length n where
#include <stdio.h>

// Recursive function to find the number of special
// sequences
int  getTotalNumberOfSequences(int m, int n)
{
    // A special sequence cannot exist if length
    // n is more than the maximum value m.
    if (m < n)
        return 0;

    // If n is 0, found an empty special sequence
    if (n == 0)
        return 1;

    // There can be two possibilities : (1) Reduce
    // last element value (2) Consider last element
    // as m and reduce number of terms
    return getTotalNumberOfSequences (m-1, n) +
           getTotalNumberOfSequences (m/2, n-1);
}

// Driver Code
int main()
{
    int m = 10;
    int n = 4;
    printf("Total number of possible sequences %d",
                  getTotalNumberOfSequences(m, n));
    return 0;
}
```

28. Longest Common Subsequence | DP-4

Last Updated: 30-09-2019

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”.

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1,2,..n-1. Recall from theory of permutation and combination that number of combinations with 1 element are ${}^nC_1$. Number of combinations with 2 elements are ${}^nC_2$ and so forth and so on. We know that ${}^nC_0 + {}^nC_1 + {}^nC_2 + \ldots {}^nC_n = 2^n$. So a string of length n has 2^n-1 different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be $O(n * 2^n)$. Note that it takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

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.

```
/* A Naive recursive implementation of LCS problem */
#include <bits/stdc++.h>
using namespace std;

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 code */
int main()
{
```

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

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

    cout<<"Length of LCS is "<< lcs( X, Y, m, n ) ;

    return 0;
}
```

29. Longest Repeated Subsequence

Given a string, print the longest repeating subsequence 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 = "aabb"
Output: "ab"

Input: str = "aab"
Output: "a"
The two subsequence 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.
```

```
int findLongestRepeatingSubSeq(string str)
{
    int n = str.length();

    // Create and initialize DP table
    int dp[n+1][n+1];
    for (int i=0; i<=n; i++)
        for (int j=0; j<=n; j++)
            dp[i][j] = 0;

    // Fill dp table (similar to LCS loops)
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            // If characters match and indexes are
            // not same
            if (str[i-1] == str[j-1] && i != j)
                dp[i][j] =  1 + dp[i-1][j-1];
```

```
            // If characters do not match
            else
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
        }
    }
    return dp[n][n];
}
```

30. Longest Increasing Subsequence | DP-3

We have already discussed Overlapping Subproblems and Optimal Substructure properties.
Now, let us discuss the 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, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.

```
Input: arr[] = {3, 10, 2, 1, 20}
Output: Length of LIS = 3
The longest increasing subsequence is 3, 10, 20

Input: arr[] = {3, 2}
Output: Length of LIS = 1
The longest increasing subsequences are {3} and {2}

Input: arr[] = {50, 3, 10, 7, 40, 80}
Output: Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}
```

```
/* Dynamic Programming C++ implementation
   of LIS problem */
#include<bits/stdc++.h>
using namespace std;

/* lis() returns the length of the longest
  increasing subsequence in arr[] of size n */
int lis( int arr[], int n )
{
    int lis[n];

    lis[0] = 1;

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

```
    {
        lis[i] = 1;
        for (int j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
    return *max_element(lis, lis+n);
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of lis is %d\n", lis( arr, n ) );

    return 0;
}
```

31. A Space Optimized Solution of LCS

Last Updated: 26-12-2018

Given two strings, find the length of longest subsequence present in both of them.

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**.

We have discussed typical dynamic programming based solution for LCS. We can optimize space used by lcs problem. We know recurrence relation of LCS problem is

```
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs(string &X, string &Y)
{
    int m = X.length(), n = Y.length();
    int L[m+1][n+1];

    /* 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 (int i=0; i<=m; i++)
    {
        for (int 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];
}
```

32. LCS (Longest Common Subsequence) of three strings

Given 3 strings of all having length < 100,the task is to find the longest common sub-sequence in all three given sequences.

Examples:

```
Input : str1 = "geeks"
        str2 = "geeksfor"
        str3 = "geeksforgeeks"
Output : 5
Longest common subsequence is "geeks"
i.e., length = 5


Input : str1 = "abcd1e2"
        str2 = "bc12ea"
        str3 = "bd1ea"
Output : 3
Longest common subsequence is "b1e"
i.e. length = 3.
```

```
// C++ program to find LCS of three strings
#include<bits/stdc++.h>
using namespace std;

    string X = "AGGT12";
    string Y = "12TXAYB";
    string Z = "12XBA";

int dp[100][100][100];

/* Returns length of LCS for X[0..m-1], Y[0..n-1]
and Z[0..o-1] */
int lcsOf3(int i, int j,int k)
{
    if(i==-1||j==-1||k==-1)
```

```
        return 0;
    if(dp[i][j][k]!=-1)
        return dp[i][j][k];

    if(X[i]==Y[j] && Y[j]==Z[k])
        return dp[i][j][k] = 1+lcsOf3(i-1,j-1,k-1);
    else
        return dp[i][j][k] = max(max(lcsOf3(i-1,j,k),
                            lcsOf3(i,j-1,k)),lcsOf3(i,j,k-1));
}

// Driver code
int main()
{
    memset(dp, -1,sizeof(dp));
    int m = X.length();
    int n = Y.length();
    int o = Z.length();

    cout << "Length of LCS is " << lcsOf3(m-1,n-1,o-1);
}
```

33. Maximum Sum Increasing Subsequence | DP-14

Last Updated: 04-04-2019

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

```
/* Dynamic Programming implementation of Maximum Sum Increasing Subsequence
(MSIS) problem */
#include <bits/stdc++.h>
using namespace std;

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

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

    /* Compute maximum sum values
    in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
```

```
            if (arr[i] > arr[j] &&
                msis[i] < msis[j] + arr[i])
                msis[i] = msis[j] + arr[i];

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

    return max;
}

// Driver Code
int main()
{
    int arr[] = {1, 101, 2, 3, 100, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Sum of maximum sum increasing "
            "subsequence is " << maxSumIS( arr, n ) << endl;
    return 0;
}
```

34. Maximum product of an increasing subsequence

Given an array of numbers, find the maximum product formed by multiplying numbers of an increasing subsequence of that array.

Note: A single number is supposed to be an increasing subsequence of size 1.
Examples:

```
Input : arr[] = { 3, 100, 4, 5, 150, 6 }

Output : 45000

Maximum product is 45000 formed by the

increasing subsequence 3, 100, 150. Note

that the longest increasing subsequence

is different {3, 4, 5, 6}


Input : arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 }

Output : 21780000

Maximum product is 21780000 formed by the

increasing subsequence 10, 22, 33, 50, 60.
```

```
/* Dynamic programming C++ implementation of maximum
   product of an increasing subsequence */
```

```
#include <bits/stdc++.h>
#define ll long long int
using namespace std;

// Returns product of maximum product increasing
// subsequence.
ll lis(ll arr[], ll n)
{
    ll mpis[n];

    /* Initialize MPIS values */
    for (int i = 0; i < n; i++)
        mpis[i] = arr[i];

    /* Compute optimized MPIS values considering
       every element as ending element of sequence */
    for (int i = 1; i < n; i++)
        for (int j = 0; j < i; j++)
            if (arr[i] > arr[j] && mpis[i] < (mpis[j] * arr[i]))
                mpis[i] = mpis[j] * arr[i];

    /* Pick maximum of all product values */
    return *max_element(mpis, mpis + n);
}

/* Driver program to test above function */
int main()
{
    ll arr[] = { 3, 100, 4, 5, 150, 6 };
    ll n = sizeof(arr) / sizeof(arr[0]);
    printf("%lld", lis(arr, n));
    return 0;
}
```

35. Count all subsequences having product less than K

Given a non negative array, find the number of subsequences having product smaller than K.

Examples:

```
Input : [1, 2, 3, 4]
        k = 10
Output :11
The subsequences are {1}, {2}, {3}, {4},
{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4},
{1, 2, 3}, {1, 2, 4}
```

```
Input  : [4, 8, 7, 2]
         k = 50
Output : 9
```

```
// CPP program to find number of subarrays having
// product less than k.
#include <bits/stdc++.h>
using namespace std;

// Function to count numbers of such subsequences
// having product less than k.
int productSubSeqCount(vector<int> &arr, int k)
{
    int n = arr.size();
    int dp[k + 1][n + 1];
    memset(dp, 0, sizeof(dp));

    for (int i = 1; i <= k; i++) {
        for (int j = 1; j <= n; j++) {

            // number of subsequence using j-1 terms
            dp[i][j] = dp[i][j - 1];

            // if arr[j-1] > i it will surely make product greater
            // thus it won't contribute then
            if (arr[j - 1] <= i && arr[j - 1] > 0)

                // number of subsequence using 1 to j-1 terms
                // and j-th term
                dp[i][j] += dp[i/arr[j-1]][j-1] + 1;
        }
    }
    return dp[k][n];
}

// Driver code
int main()
{
    vector<int> A;
    A.push_back(1);
    A.push_back(2);
    A.push_back(3);
    A.push_back(4);
    int k = 10;
    cout << productSubSeqCount(A, k) << endl;
}
```

36. Maximum subsequence sum such that no three are consecutive

Last Updated: 25-04-2019

Given a sequence of positive numbers, find the maximum sum that can be formed which has no three consecutive elements present.

Examples :

```
Input: arr[] = {1, 2, 3}

Output: 5

We can't take three of them, so answer is

2 + 3 = 5


Input: arr[] = {3000, 2000, 1000, 3, 10}

Output: 5013

3000 + 2000 + 3 + 10 = 5013


Input: arr[] = {100, 1000, 100, 1000, 1}

Output: 2101

100 + 1000 + 1000 + 1 = 2101


Input: arr[] = {1, 1, 1, 1, 1}

Output: 4


Input: arr[] = {1, 2, 3, 4, 5, 6, 7, 8}

Output: 27
```

```
// C++ program to find the maximum sum such that
// no three are consecutive
#include <bits/stdc++.h>
using namespace std;

// Returns maximum subsequence sum such that no three
// elements are consecutive
int maxSumWO3Consec(int arr[], int n)
{
    // Stores result for subarray arr[0..i], i.e.,
    // maximum possible sum in subarray arr[0..i]
    // such that no three elements are consecutive.
    int sum[n];

    // Base cases (process first three elements)
    if (n >= 1)
        sum[0] = arr[0];
```

```
    if (n >= 2)
        sum[1] = arr[0] + arr[1];

    if (n > 2)
        sum[2] = max(sum[1], max(arr[1] + arr[2], arr[0] + arr[2]));

    // Process rest of the elements
    // We have three cases
    // 1) Exclude arr[i], i.e., sum[i] = sum[i-1]
    // 2) Exclude arr[i-1], i.e., sum[i] = sum[i-2] + arr[i]
    // 3) Exclude arr[i-2], i.e., sum[i-3] + arr[i] + arr[i-1]
    for (int i = 3; i < n; i++)
        sum[i] = max(max(sum[i - 1], sum[i - 2] + arr[i]),
                     arr[i] + arr[i - 1] + sum[i - 3]);

    return sum[n - 1];
}

// Driver code
int main()
{
    int arr[] = { 100, 1000 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << maxSumWO3Consec(arr, n);
    return 0;
}
```

37. Longest subsequence such that difference between adjacents is one

Last Updated: 24-05-2018

Given an array of n size, the task is to find the longest subsequence such that difference between adjacents is one.

Examples:

```
Input :  arr[] = {10, 9, 4, 5, 4, 8, 6}
Output :  3
As longest subsequences with difference 1 are, "10, 9, 8",
"4, 5, 4" and "4, 5, 6"

Input :  arr[] = {1, 2, 3, 2, 3, 7, 2, 1}
Output :  7
As longest consecutive sequence is "1, 2, 3, 2, 3, 2, 1"
```

```
// C++ program to find the longest subsequence such
// the difference between adjacent elements of the
// subsequence is one.
#include<bits/stdc++.h>
using namespace std;

// Function to find the length of longest subsequence
```

```
int longestSubseqWithDiffOne(int arr[], int n)
{
    // Initialize the dp[] array with 1 as a
    // single element will be of 1 length
    int dp[n];
    for (int i = 0; i< n; i++)
        dp[i] = 1;

    // Start traversing the given array
    for (int i=1; i<n; i++)
    {
        // Compare with all the previous elements
        for (int j=0; j<i; j++)
        {
            // If the element is consecutive then
            // consider this subsequence and update
            // dp[i] if required.
            if ((arr[i] == arr[j]+1) ||
                (arr[i] == arr[j]-1))

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

    // Longest length will be the maximum value
    // of dp array.
    int result = 1;
    for (int i = 0 ; i < n ; i++)
        if (result < dp[i])
            result = dp[i];
    return result;
}

// Driver code
int main()
{
    // Longest subsequence with one difference is
    // {1, 2, 3, 4, 3, 2}
    int arr[] = {1, 2, 3, 4, 5, 3, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << longestSubseqWithDiffOne(arr, n);
    return 0;
}
```

38. Maximum length subsequence with difference between adjacent elements as either 0 or 1

Last Updated: 04-06-2018

Given an array of **n** integers. The problem is to find maximum length of the subsequence with difference between adjacent elements as either 0 or 1.
Examples:

Input : arr[] = {2, 5, 6, 3, 7, 6, 5, 8}

Output : 5

The subsequence is {5, 6, 7, 6, 5}.

Input : arr[] = {-2, -1, 5, -1, 4, 0, 3}

Output : 4

The subsequence is {-2, -1, -1, 0}.

```
// C++ implementation to find maximum length
// subsequence with difference between adjacent
// elements as either 0 or 1
#include <bits/stdc++.h>
using namespace std;

// function to find maximum length subsequence
// with difference between adjacent elements as
// either 0 or 1
int maxLenSub(int arr[], int n)
{
    int mls[n], max = 0;

    // Initialize mls[] values for all indexes
    for (int i=0; i<n; i++)
        mls[i] = 1;

    // Compute optimized maximum length subsequence
    // values in bottom up manner
    for (int i=1; i<n; i++)
        for (int j=0; j<i; j++)
            if (abs(arr[i] - arr[j]) <= 1 &&
                    mls[i] < mls[j] + 1)
                mls[i] = mls[j] + 1;

    // Store maximum of all 'mls' values in 'max'
    for (int i=0; i<n; i++)
        if (max < mls[i])
            max = mls[i];

    // required maximum length subsequence
    return max;
}

// Driver program to test above
int main()
{
    int arr[] = {2, 5, 6, 3, 7, 6, 5, 8};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum length subsequence = "
         << maxLenSub(arr, n);
    return 0; }
```

39. Maximum sum increasing subsequence from a prefix and a given element after prefix is must

Last Updated: 17-09-2020

Given an array of n positive integers, write a program to find the maximum sum of increasing subsequence from prefix till ith index and also including a given kth element which is after i, i.e., k > i .

Examples :

Input: *arr[] = {1, 101, 2, 3, 100, 4, 5} i-th index = 4 (Element at 4th index is 100) K-th index = 6 (Element at 6th index is 5.)*
Output: *11*
Explanation:
So we need to calculate the maximum sum of subsequence (1 101 2 3 100 5) such that 5 is necessarily included in the subsequence, so answer is 11 by subsequence (1 2 3 5).
Input: *arr[] = {1, 101, 2, 3, 100, 4, 5} i-th index = 2 (Element at 2nd index is 2) K-th index = 5 (Element at 5th index is 4.)*
Output: *7*
Explanation:
So we need to calculate the maximum sum of subsequence (1 101 2 4) such that 4 is necessarily included in the subsequence, so answer is 7 by subsequence (1 2 4).

```
// CPP program to find maximum sum increasing
// subsequence till i-th index and including
// k-th index.
#include <bits/stdc++.h>
#define ll long long int
using namespace std;

ll pre_compute(ll a[], ll n, ll index, ll k)
{
    ll dp[n][n] = { 0 };

    // Initializing the first row of the dp[][].
    for (int i = 0; i < n; i++) {
        if (a[i] > a[0])
            dp[0][i] = a[i] + a[0];
        else
            dp[0][i] = a[i];
    }

    // Creating the dp[][] matrix.
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (a[j] > a[i] && j > i) {
                if (dp[i - 1][i] + a[j] > dp[i - 1][j])
                    dp[i][j] = dp[i - 1][i] + a[j];
                else
                    dp[i][j] = dp[i - 1][j];
```

```
            }
            else
                dp[i][j] = dp[i - 1][j];
        }
    }

    // To calculate for i=4 and k=6.
    return dp[index][k];
}

int main()
{
    ll a[] = { 1, 101, 2, 3, 100, 4, 5 };
    ll n = sizeof(a) / sizeof(a[0]);
    ll index = 4, k = 6;
    printf("%lld", pre_compute(a, n, index, k));
    return 0;
}
```

40. Maximum Length Chain of Pairs | DP-20

Last Updated: 14-10-2020

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}}

```
// CPP program for above approach
#include <bits/stdc++.h>
using namespace std;

// Structure for a Pair
class Pair
{
    public:
    int a;
    int b;
};

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

    return max;
}


/* Driver code */
int main()
{
    Pair arr[] = { {5, 24}, {15, 25},
                          {27, 40}, {50, 60} };
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Length of maximum size chain is " << maxChainLength( arr, n );
    return 0;
}
```

41. Print Maximum Length Chain of Pairs

Last Updated: 12-04-2019

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.

Examples:

```
Input:  (5, 24), (39, 60), (15, 28), (27, 40), (50, 90)
Output: (5, 24), (27, 40), (50, 90)

Input:  (11, 20), {10, 40), (45, 60), (39, 40)
Output: (11, 20), (39, 40), (45, 60)
```

```
#include <bits/stdc++.h>
using namespace std;

struct Pair
```

```
{
    int a;
    int b;
};

// comparator function for sort function
int compare(Pair x, Pair y)
{
    return x.a < y.a;
}

// Function to construct Maximum Length Chain
// of Pairs
void maxChainLength(vector<Pair> arr)
{
    // Sort by start time
    sort(arr.begin(), arr.end(), compare);

    // L[i] stores maximum length of chain of
    // arr[0..i] that ends with arr[i].
    vector<vector<Pair> > L(arr.size());

    // L[0] is equal to arr[0]
    L[0].push_back(arr[0]);

    // start from index 1
    for (int i = 1; i < arr.size(); i++)
    {
        // for every j less than i
        for (int j = 0; j < i; j++)
        {
            // L[i] = {Max(L[j])} + arr[i]
            // where j < i and arr[j].b < arr[i].a
            if ((arr[j].b < arr[i].a) &&
                (L[j].size() > L[i].size()))
                L[i] = L[j];
        }
        L[i].push_back(arr[i]);
    }

    // print max length vector
    vector<Pair> maxChain;
    for (vector<Pair> x : L)
        if (x.size() > maxChain.size())
            maxChain = x;

    for (Pair pair : maxChain)
        cout << "(" << pair.a << ", "
             << pair.b << ") ";
}

// Driver Function
int main()
{
    Pair a[] = {{5, 29}, {39, 40}, {15, 28},
```

```
                {27, 40}, {50, 90}};
    int n = sizeof(a)/sizeof(a[0]);

    vector<Pair> arr(a, a + n);

    maxChainLength(arr);

    return 0;
}
```

42. Path with maximum average value

Last Updated: 19-08-2019

Given a square matrix of size N*N, where each cell is associated with a specific cost. A path is defined as a specific sequence of cells which starts from the top left cell move only right or down and ends on bottom right cell. We want to find a path with the maximum average over all existing paths. Average is computed as total cost divided by the number of cells visited in the path.

Examples:

```
Input : Matrix = [1, 2, 3
                 4, 5, 6
                 7, 8, 9]
Output : 5.8
Path with maximum average is, 1 -> 4 -> 7 -> 8 -> 9
Sum of the path is 29 and average is 29/5 = 5.8
```

```
//    C/C++ program to find maximum average cost path
#include <bits/stdc++.h>
using namespace std;

// Maximum number of rows and/or columns
const int M = 100;

// method returns maximum average of all path of
// cost matrix
double maxAverageOfPath(int cost[M][M], int N)
{
    int dp[N+1][N+1];
    dp[0][0] = cost[0][0];

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

    /* Initialize first row of dp array */
    for (int j = 1; j < N; j++)
        dp[0][j] = dp[0][j-1] + cost[0][j];
```

```
    /* Construct rest of the dp array */
    for (int i = 1; i < N; i++)
        for (int j = 1; j <= N; j++)
            dp[i][j] = max(dp[i-1][j],
                           dp[i][j-1]) + cost[i][j];

    // divide maximum sum by constant path
    // length : (2N - 1) for getting average
    return (double)dp[N-1][N-1] / (2*N-1);
}

/* Driver program to test above functions */
int main()
{
    int cost[M][M] = { {1, 2, 3},
        {6, 5, 4},
        {7, 3, 9}
    };
    printf("%f", maxAverageOfPath(cost, 3));
    return 0;
}
```

43. Maximum games played by winner

Last Updated: 05-10-2018

There are N players which are playing a tournament. We need to find the maximum number of games the winner can play. In this tournament, two players are allowed to play against each other only if the difference between games played by them is not more than one.

Examples:

```
Input  : N = 3

Output : 2

Maximum games winner can play = 2

Assume that player are P1, P2 and P3

First, two players will play let (P1, P2)

Now winner will play against P3,

making total games played by winner = 2


Input  : N = 4

Output : 2

Maximum games winner can play = 2

Assume that player are P1, P2, P3 and P4
```

```
First two pairs will play lets (P1, P2) and
(P3, P4). Now winner of these two games will
play against each other, making total games
played by winner = 2
```

```
// C/C++ program to find maximum number of
// games played by winner
#include <bits/stdc++.h>
using namespace std;

// method returns maximum games a winner needs
// to play in N-player tournament
int maxGameByWinner(int N)
{
    int dp[N];

    // for 0 games, 1 player is needed
    // for 1 game, 2 players are required
    dp[0] = 1;
    dp[1] = 2;

    // loop until i-th Fibonacci number is
    // less than or equal to N
    int i = 2;
    do {
        dp[i] = dp[i - 1] + dp[i - 2];
    } while (dp[i++] <= N);

    // result is (i - 2) because i will be
    // incremented one extra in while loop
    // and we want the last value which is
    // smaller than N, so one more decrement
    return (i - 2);
}

// Driver code to test above methods
int main()
{
    int N = 10;
    cout << maxGameByWinner(N) << endl;
    return 0;
}
```

44. Maximum path sum in a triangle.

Last Updated: 23-05-2018

We have given numbers in form of triangle, by starting at the top of the triangle and moving to adjacent numbers on the row below, find the maximum total from top to bottom.

Examples :

```
Input :
   3
  7 4
 2 4 6
8 5 9 3
Output : 23
Explanation : 3 + 7 + 4 + 9 = 23


Input :
   8
 -4 4
 2 2 6
1 1 1 1
Output : 19
Explanation : 8 + 4 + 6 + 1 = 19
```

```
// C++ program for Dynamic
// Programming implementation of
// Max sum problem in a triangle
#include<bits/stdc++.h>
using namespace std;
#define N 3

//  Function for finding maximum sum
int maxPathSum(int tri[][N], int m, int n)
{
     // loop for bottom-up calculation
     for (int i=m-1; i>=0; i--)
     {
        for (int j=0; j<=i; j++)
        {
            // for each element, check both
            // elements just below the number
            // and below right to the number
            // add the maximum of them to it
            if (tri[i+1][j] > tri[i+1][j+1])
                tri[i][j] += tri[i+1][j];
            else
                tri[i][j] += tri[i+1][j+1];
        }
     }

     // return the top element
     // which stores the maximum sum
```

```
     return tri[0][0];
}

/* Driver program to test above functions */
int main()
{
   int tri[N][N] = {  {1, 0, 0},
                      {4, 8, 0},
                      {1, 5, 3} };
   cout << maxPathSum(tri, 2, 2);
   return 0;
}
```

45. Minimum Sum Path in a Triangle

Last Updated: 20-01-2020

Given a triangular structure of numbers, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

Examples :

```
Input :

   2

  3 7

 8 5 6

6 1 9 3

Output : 11

Explanation : 2 + 3 + 5 + 1 = 11


Input :

   3

  6 4

 5 2 7

9 1 8 6

Output : 10

Explanation : 3 + 4 + 2 + 1 = 10
```

```
// C++ program for Dynamic
// Programming implementation of
// Min Sum Path in a Triangle
#include <bits/stdc++.h>
using namespace std;
```

```
// Util function to find minimum sum for a path
int minSumPath(vector<vector<int> >& A)
{
    // For storing the result in a 1-D array,
    // and simultaneously updating the result.
    int memo[A.size()];
    int n = A.size() - 1;

    // For the bottom row
    for (int i = 0; i < A[n].size(); i++)
        memo[i] = A[n][i];

    // Calculation of the remaining rows,
    // in bottom up manner.
    for (int i = A.size() - 2; i >= 0; i--)
        for (int j = 0; j < A[i].size(); j++)
            memo[j] = A[i][j] + min(memo[j],
                                    memo[j + 1]);

    // return the top element
    return memo[0];
}

/* Driver program to test above functions */
int main()
{
    vector<vector<int> > A{ { 2 },
                            { 3, 9 },
                            { 1, 6, 7 } };
    cout << minSumPath(A);
    return 0;
}
```

46. Maximum sum of a path in a Right Number Triangle

Last Updated: 22-06-2018

Given a right triangle of numbers, find the largest of the sum of numbers that appear on the paths starting from the top towards the base, so that on each path the next number is located directly below or below-and-one-place-to-the-right.

Examples :

```
Input : 1

        1 2

        4 1 2

        2 3 1 1

Output : 9

Explanation : 1 + 1 + 4 + 3
```

Input : 2

4 1

1 2 7

Output : 10

Explanation : 2 + 1 + 7

```
// C++ program to print maximum sum
// in a right triangle of numbers
#include<bits/stdc++.h>
using namespace std;

// function to find maximum sum path
int maxSum(int tri[][3], int n)
{
    // Adding the element of row 1 to both the
    // elements of row 2 to reduce a step from
    // the loop
    if (n > 1)
        tri[1][1] = tri[1][1] + tri[0][0];
        tri[1][0] = tri[1][0] + tri[0][0];

    // Traverse remaining rows
    for(int i = 2; i < n; i++) {
        tri[i][0] = tri[i][0] + tri[i-1][0];
        tri[i][i] = tri[i][i] + tri[i-1][i-1];

        //Loop to traverse columns
        for (int j = 1; j < i; j++){

            // Checking the two conditions,
            // directly below and below right.
            // Considering the greater one

            // tri[i] would store the possible
            // combinations of sum of the paths
            if (tri[i][j] + tri[i-1][j-1] >=
                           tri[i][j] + tri[i-1][j])

                tri[i][j] = tri[i][j] + tri[i-1][j-1];
            else
                tri[i][j] = tri[i][j]+tri[i-1][j];
        }
    }

    // array at n-1 index (tri[i]) stores
    // all possible adding combination, finding
    // the maximum one out of them
    int max=tri[n-1][0];

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

```
    {
        if(max<tri[n-1][i])
            max=tri[n-1][i];
    }

    return max;
}

// driver program
int main(){

    int tri[3][3] = {{1}, {2,1}, {3,3,2}};

    cout<<maxSum(tri, 3);

    return 0;
}
```

47. Size of The Subarray With Maximum Sum

Last Updated: 09-05-2020
An array is given, find length of the subarray having maximum sum.

Examples :

```
Input :  a[] = {1, -2, 1, 1, -2, 1}

Output : Length of the subarray is 2

Explanation: Subarray with consecutive elements

and maximum sum will be {1, 1}. So length is 2


Input : ar[] = { -2, -3, 4, -1, -2, 1, 5, -3 }

Output : Length of the subarray is 5

Explanation: Subarray with consecutive elements

and maximum sum will be {4, -1, -2, 1, 5}.
```

```
// C++ program to print length of the largest
// contiguous array sum
#include<bits/stdc++.h>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0,
       start =0, end = 0, s=0;

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

```
        if (max_so_far < max_ending_here)
        {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }

        if (max_ending_here < 0)
        {
            max_ending_here = 0;
            s = i + 1;
        }
    }

    return (end - start + 1);
}

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

48. Maximum sum of pairs with specific difference

Last Updated: 25-04-2019

Given an array of integers and a number k. We can pair two number of array if difference between them is strictly less than k. The task is to find maximum possible sum of disjoint pairs. Sum of P pairs is sum of all 2P numbers of pairs.

Examples:

```
Input  : arr[] = {3, 5, 10, 15, 17, 12, 9}, K = 4

Output : 62

Then disjoint pairs with difference less than K are,

(3, 5), (10, 12), (15, 17)

So maximum sum which we can get is 3 + 5 + 12 + 10 + 15 + 17 = 62

Note that an alternate way to form disjoint pairs is,

(3, 5), (9, 12), (15, 17), but this pairing produces lesser sum.


Input  : arr[] = {5, 15, 10, 300}, k = 12

Output : 25
```

```
// C++ program to find maximum pair sum whose
// difference is less than K
#include <bits/stdc++.h>
using namespace std;

// method to return maximum sum we can get by
// finding less than K difference pair
int maxSumPairWithDifferenceLessThanK(int arr[], int N, int K)
{
    // Sort input array in ascending order.
    sort(arr, arr+N);

    // dp[i] denotes the maximum disjoint pair sum
    // we can achieve using first i elements
    int dp[N];

    //  if no element then dp value will be 0
    dp[0] = 0;

    for (int i = 1; i < N; i++)
    {
        // first give previous value to dp[i] i.e.
        // no pairing with (i-1)th element
        dp[i] = dp[i-1];

        // if current and previous element can form a pair
        if (arr[i] - arr[i-1] < K)
        {
            // update dp[i] by choosing maximum between
            // pairing and not pairing
            if (i >= 2)
                dp[i] = max(dp[i], dp[i-2] + arr[i] + arr[i-1]);
            else
                dp[i] = max(dp[i], arr[i] + arr[i-1]);
        }
    }

    //  last index will have the result
    return dp[N - 1];
}

//  Driver code to test above methods
int main()
{
    int arr[] = {3, 5, 10, 15, 17, 12, 9};
    int N = sizeof(arr)/sizeof(int);

    int K = 4;
    cout << maxSumPairWithDifferenceLessThanK(arr, N, K);
    return 0;
}
```

49. Maximum number of segments of lengths a, b and c

Last Updated: 16-10-2018

Given a positive integer N, find the maximum number of segments of lengths a, b and c that can be formed from N .

Examples :

```
Input : N = 7, a = 5, b, = 2, c = 5

Output : 2

N can be divided into 2 segments of lengths

2 and 5. For the second example,


Input : N = 17, a = 2, b = 1, c = 3

Output : 17

N can be divided into 17 segments of 1 or 8

segments of 2 and 1 segment of 1. But 17 segments

of 1 is greater than 9 segments of 2 and 1.
```

```
// C++ implementation to divide N into
// maximum number of segments
// of length a, b and c
#include <bits/stdc++.h>
using namespace std;

// function to find the maximum
// number of segments
int maximumSegments(int n, int a,
                    int b, int c)
{
    // stores the maximum number of
    // segments each index can have
    int dp[n + 1];

    // initialize with -1
    memset(dp, -1, sizeof(dp));

    // 0th index will have 0 segments
    // base case
    dp[0] = 0;

    // traverse for all possible
    // segments till n
    for (int i = 0; i < n; i++)
    {
        if (dp[i] != -1) {

            // conditions
        if(i + a <= n )    //avoid buffer overflow
                dp[i + a] = max(dp[i] + 1,
```

```
                            dp[i + a]);

        if(i + b <= n ) //avoid buffer overflow
                dp[i + b] = max(dp[i] + 1,
                            dp[i + b]);

        if(i + c <= n )    //avoid buffer overflow
                dp[i + c] = max(dp[i] + 1,
                            dp[i + c]);
        }
    }
    return dp[n];
}

// Driver code
int main()
{
    int n = 7, a = 5, b = 2, c = 5;
    cout << maximumSegments(n, a, b, c);
    return 0;
}
```

50. Recursively break a number in 3 parts to get maximum sum

Last Updated: 24-07-2019

Given a number n, we can divide it in only three parts n/2, n/3 and n/4 (we will consider only integer part). The task is to find the maximum sum we can make by dividing number in three parts recursively and summing up them together.

Examples:

```
Input : n = 12

Output : 13

// We break n = 12 in three parts {12/2, 12/3, 12/4}

// = {6, 4, 3},  now current sum is = (6 + 4 + 3) = 13

// again we break 6 = {6/2, 6/3, 6/4} = {3, 2, 1} = 3 +

// 2 + 1 = 6 and further breaking 3, 2 and 1 we get maximum

// summation as 1, so breaking 6 in three parts produces

// maximum sum 6 only similarly breaking 4 in three

// parts we can get maximum sum 4 and same for 3 also.

// Thus maximum sum by breaking number in parts  is=13
```

```
// A simple recursive C++ program to find
// maximum sum by recursively breaking a
// number in 3 parts.
```

```
#include<bits/stdc++.h>
using namespace std;

// Function to find the maximum sum
int breakSum(int n)
{
    // base conditions
    if (n==0 || n == 1)
        return n;

    // recursively break the number and return
    // what maximum you can get
    return max((breakSum(n/2) + breakSum(n/3) +
                breakSum(n/4)),  n);
}

// Driver program to run the case
int main()
{
    int n = 12;
    cout << breakSum(n);
    return 0;
}
```

51. Maximum value with the choice of either dividing or considering as it is

Last Updated: 14-05-2018

We are given a number n, we need to find the maximum sum possible with the help of following function:

F(n) = max((F(n/2) + F(n/3) + F(n/4) + F(n/5)), n). To calculate F(n,) we may either have n as our result or we can further break n into four part as in given function definition. This can be done as much time as we can. Find the maximum possible sum you can get from a given N. Note : 1 can not be break further so F(1) = 1 as a base case.

Examples :

```
Input : n = 10

Output : MaxSum = 12

Explanation:

f(10) = f(10/2) + f(10/3) + f(10/4) + f(10/5)

      = f(5)  +   f(3)  +  f(2)   +  f(2)

      = 12

5, 3 and 2 cannot be further divided.
```

```
Input : n = 2

Output : MaxSum = 2
```

```
// CPP program for maximize result when
// we have choice to divide or consider
// as it is.
#include <bits/stdc++.h>
using namespace std;

// function for calculating max possible result
int maxDP(int n)
{
    int res[n + 1];
    res[0] = 0;
    res[1] = 1;

    // Compute remaining values in bottom
    // up manner.
    for (int i = 2; i <= n; i++)
        res[i] = max(i, (res[i / 2] + res[i / 3] + res[i / 4] + res[i / 5]));

    return res[n];
}

// driver program
int main()
{
    int n = 60;
    cout << "MaxSum =" << maxDP(n);
    return 0;
}
```

52. Maximum weight path ending at any element of last row in a matrix

Last Updated: 19-12-2018

Given a matrix of integers where every element represents weight of the cell. Find the path having the maximum weight in matrix [N X N]. Path Traversal Rules are:

- It should begin from top left element.
- The path can end at any element of last row.
- We can move to following two cells from a cell (i, j).
    1. Down Move : (i+1, j)
    2. Diagonal Move : (i+1, j+1)

Examples:

```
Input : N = 5

        mat[5][5] = {{ 4, 2 ,3 ,4 ,1 },

                     { 2 , 9 ,1 ,10 ,5 },
```

```
                    {15, 1 ,3 , 0 ,20 },
                    {16 ,92, 41, 44 ,1},
                    {8, 142, 6, 4, 8} };
Output : 255
Path with max weight : 4 + 2 +15 + 92 + 142 = 255
```

```
// C++ program to find the path having the
// maximum weight in matrix
#include<bits/stdc++.h>
using namespace std;
const int MAX = 1000;

/* Function which return the maximum weight
   path sum */
int maxCost(int mat[][MAX], int N)
{
    // creat 2D matrix to store the sum of the path
    int dp[N][N];
    memset(dp, 0, sizeof(dp));

    dp[0][0] = mat[0][0];

    // Initialize first column of total weight
    // array (dp[i to N][0])
    for (int i=1; i<N; i++)
        dp[i][0] = mat[i][0] + dp[i-1][0];

    // Calculate rest paht sum of weight matrix
    for (int i=1; i<N; i++)
       for (int j=1; j<i+1&&j<N; j++)
          dp[i][j] = mat[i][j] +
                     max(dp[i-1][j-1], dp[i-1][j]);

    // find the max weight path sum to rech
    // the last row
    int result = 0;
    for (int i=0; i<N; i++)
        if (result < dp[N-1][i])
            result = dp[N-1][i];

    // return maximum weight path sum
    return result;
}

// Driver program
int main()
{
    int mat[MAX][MAX] = {  { 4, 1 ,5 ,6 , 1 },
        { 2 ,9 ,2 ,11 ,10 },
        { 15,1 ,3 ,15, 2 },
        { 16, 92, 41,4,3},
        { 8, 142, 6, 4, 8 }
```

```
    };
    int N = 5;
    cout << "Maximum Path Sum : "
        << maxCost(mat, N)<<endl;
    return 0;
}
```

53. Maximum sum in a 2 x n grid such that no two elements are adjacent

Last Updated: 28-02-2019

Given a rectangular grid of dimension 2 x n. We need to find out the maximum sum such that no two chosen numbers are adjacent, vertically, diagonally or horizontally.

Examples:

```
Input : 1 4 5
        2 0 0
Output : 7
If we start from 1 then we can add only 5 or 0.
So max_sum = 6 in this case.
If we select 2 then also we can add only 5 or 0.
So max_sum = 7 in this case.
If we select from 4 or 0  then there is no further
elements can be added.
So, Max sum is 7.


Input : 1 2 3 4 5
        6 7 8 9 10
Output : 24
```

```
// C++ program to find maximum sum in a grid such that
// no two elements are adjacent.
#include<bits/stdc++.h>
#define MAX 1000
using namespace std;

// Function to find max sum without adjacent
int maxSum(int grid[2][MAX], int n)
{
    // Sum including maximum element of first column
    int incl = max(grid[0][0], grid[1][0]);
```

```
    // Not including first column's element
    int excl = 0, excl_new;

    // Traverse for further elements
    for (int i = 1; i<n; i++ )
    {
        // Update max_sum on including or excluding
        // of previous column
        excl_new = max(excl, incl);

        // Include current column. Add maximum element
        // from both row of current column
        incl = excl + max(grid[0][i], grid[1][i]);

        // If current column doesn't to be included
        excl = excl_new;
    }

    // Return maximum of excl and incl
    // As that will be the maximum sum
    return max(excl, incl);
}

// Driver code
int main()
{
    int grid[2][MAX] = {{ 1, 2, 3, 4, 5},
                        { 6, 7, 8, 9, 10}};

    int n = 5;
    cout << maxSum(grid, n);

    return 0;
}
```

54. Maximum difference of zeros and ones in binary string | Set 2 (O(n) time)

Last Updated: 21-11-2018

Given a binary string of 0s and 1s. The task is to find the maximum difference between the number of 0s and number of 1s in any sub-string of the given binary string. That is maximize (number of 0s – number of 1s) for any sub-string in the given binary string.

Examples:

```
Input : S = "11000010001"

Output : 6

From index 2 to index 9, there are 7

0s and 1 1s, so number of 0s - number
```

of 1s is 6.

Input : S = "1111"

Output : -1

```
// CPP Program to find the length of
// substring with maximum difference of
// zeros and ones in binary string.
#include <iostream>
using namespace std;

// Returns the length of substring with
// maximum difference of zeroes and ones
// in binary string
int findLength(string str, int n)
{
    int current_sum = 0;
    int max_sum = 0;

    // traverse a binary string from left
    // to right
    for (int i = 0; i < n; i++) {

        // add current value to the current_sum
        // according to the Character
        // if it's '0' add 1 else -1
        current_sum += (str[i] == '0' ? 1 : -1);

        if (current_sum < 0)
            current_sum = 0;

        // update maximum sum
        max_sum = max(current_sum, max_sum);
    }

    // return -1 if string does not contain
    // any zero that means all ones
    // otherwise max_sum
    return max_sum == 0 ? -1 : max_sum;
}

// Driven Program
int main()
{
    string s = "11000010001";
    int n = 11;
    cout << findLength(s, n) << endl;
    return 0;
}
```

55. Maximum path sum for each position with jumps under divisibility condition

Last Updated: 27-11-2019

Given an array of **n** positive integers. Initially we are at first position. We can jump to position y (1 <= y <= n) from position x (1 <= x <= n) if x divides y and x < y. The task is to print maximum sum path ending at every position x.
Note : Since first element is at position 1, we can jump to any position from here as 1 divides all other position numbers.

Examples :

```
Input :  arr[] = {2, 3, 1, 4, 6, 5}
Output : 2 5 3 9 8 10
Maximum sum path ending with position 1 is 2.
For position 1, last position to visit is 1 only.
So maximum sum for position 1 = 2.

Maximum sum path ending with position 2 is 5.
For position 2, path can be jump from position 1
to 2 as 1 divides 2.
So maximum sum for position 2 = 2 + 3 = 5.

For position 3, path can be jump from position 1
to 3 as 1 divides 3.
So maximum sum for position 3 = 2 + 3 = 5.

For position 4, path can be jump from position 1
to 2 and 2 to 4.
So maximum sum for position 4 = 2 + 3 + 4 = 9.

For position 5, path can be jump from position 1
to 5.
So maximum sum for position 5 = 2 + 6 = 8.

For position 6, path can be jump from position
1 to 2 and 2 to 6 or 1 to 3 and 3 to 6.
But path 1 -> 2 -> 6 gives maximum sum for
position 6 = 2 + 3 + 5 = 10.
```

```
// C++ program to print maximum path sum ending with
// each position x such that all path step positions
// divide x.
#include <bits/stdc++.h>
using namespace std;

void printMaxSum(int arr[], int n)
{
    // Create an array such that dp[i] stores maximum
```

```
    // path sum ending with i.
    int dp[n];
    memset(dp, 0, sizeof dp);

    // Calculating maximum sum path for each element.
    for (int i = 0; i < n; i++) {
        dp[i] = arr[i];

        // Finding previous step for arr[i]
        // Moving from 1 to sqrt(i+1) since all the
        // divisors are present from sqrt(i+1).
        int maxi = 0;
        for (int j = 1; j <= sqrt(i + 1); j++) {
            // Checking if j is divisor of i+1.
            if (((i + 1) % j == 0) && (i + 1) != j) {
                // Checking which divisor will provide
                // greater value.
                if (dp[j - 1] > maxi)
                    maxi = dp[j - 1];
                if (dp[(i + 1) / j - 1] > maxi && j != 1)
                    maxi = dp[(i + 1) / j - 1];
            }
        }

        dp[i] += maxi;
    }

    // Printing the answer (Maximum path sum ending
    // with every position i+1.
    for (int i = 0; i < n; i++)
        cout << dp[i] << " ";
}

// Driven Program
int main()
{
    int arr[] = { 2, 3, 1, 4, 6, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);

    printMaxSum(arr, n);

    return 0;
}
```

56. Maximize the sum of selected numbers from an array to make it empty

Last Updated: 05-08-2020

Given an array of N numbers, we need to maximize the sum of selected numbers. At each step, you need to select a number A_i, delete one occurrence of it and delete all occurrences of $\mathbf{A_i-1}$ and $\mathbf{A_i+1}$ (if they exist) in the array. Repeat these steps until the

array gets empty. The problem is to maximize the sum of the selected numbers.
Note: We have to delete all the occureneces of A_i+1 and A_i-1 elements if they are present in the array and not A_{i+1} and A_{i-1}.
Examples:

```
Input : a[] = {1, 2, 3}
Output : 4
Explanation: At first step we select 1, so 1 and
2 are deleted from the sequence leaving us with 3.
Then we select 3 from the sequence and delete it.
So the sum of selected numbers is 1+3 = 4.



Input : a[] =  {1, 2, 2, 2, 3, 4}
Output : 10
Explanation : Select one of the 2's from the array, so
2, 2-1, 2+1 will be deleted and we are left with {2, 2, 4},
since 1 and 3 are deleted. Select 2 in next two steps,
and then select 4 in the last step.
We get a sum of 2+2+2+4=10 which is the maximum possible.
```

```
// CPP program to Maximize the sum of selected
// numbers by deleting three consecutive numbers.
#include <bits/stdc++.h>
using namespace std;

// function to maximize the sum of selected numbers
int maximizeSum(int a[], int n) {

  // stores the occurrences of the numbers
  unordered_map<int, int> ans;

  // marks the occurrence of every number in the sequence
  for (int i = 0; i < n; i++)
    ans[a[i]]++;

  // maximum in the sequence
  int maximum = *max_element(a, a + n);

  // traverse till maximum and apply the recurrence relation
  for (int i = 2; i <= maximum; i++)
    ans[i] = max(ans[i - 1], ans[i - 2] + ans[i] * i);
```

```
  // return the ans stored in the index of maximum
  return ans[maximum];
}

// Driver code
int main()
{
  int a[] = {1, 2, 3};
  int n = sizeof(a) / sizeof(a[0]);
  cout << maximizeSum(a, n);
  return 0;
}
```

57. Maximum subarray sum in an array created after repeated concatenation

Last Updated: 09-05-2020

Given an array and a number k, find the largest sum of contiguous array in the modified array which is formed by repeating the given array k times.

Examples :

```
Input  : arr[] = {-1, 10, 20}, k = 2

Output : 59

After concatenating array twice, we

get {-1, 10, 20, -1, 10, 20} which has

maximum subarray sum as 59.


Input  : arr[] = {-1, -2, -3}, k = 3

Output : -1
```

```
// C++ program to print largest contiguous
// array sum when array is created after
// concatenating a small array k times.
#include<bits/stdc++.h>
using namespace std;

// Returns sum of maximum sum subarray created
// after concatenating a[0..n-1] k times.
int maxSubArraySumRepeated(int a[], int n, int k)
{
    int max_so_far = INT_MIN, max_ending_here = 0;

    for (int i = 0; i < n*k; i++)
    {
```

```
        // This is where it differs from Kadane's
        // algorithm. We use modular arithmetic to
        // find next element.
        max_ending_here = max_ending_here + a[i%n];

        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;

        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {10, 20, -30, -1};
    int n = sizeof(a)/sizeof(a[0]);
    int k = 3;
    cout << "Maximum contiguous sum is "
         << maxSubArraySumRepeated(a, n, k);
    return 0;
}
```

58. Maximum path sum that starting with any cell of 0-th row and ending with any cell of (N-1)-th row

Last Updated: 05-02-2019

Given a N X N matrix Mat[N][N] of positive integers. There are only three possible moves from a cell (i, j)

1. (i+1, j)
2. (i+1, j-1)
3. (i+1, j+1)

Starting from any column in row 0, return the largest sum of any of the paths up to row N-1.

Examples:

```
Input : mat[4][4] = { {4, 2, 3, 4},
                      {2, 9, 1, 10},
                      {15, 1, 3, 0},
                      {16, 92, 41, 44} };

Output :120

path : 4 + 9 + 15 + 92 = 120
```

```
// C++ program to find Maximum path sum
// start any column in row '0' and ends
```

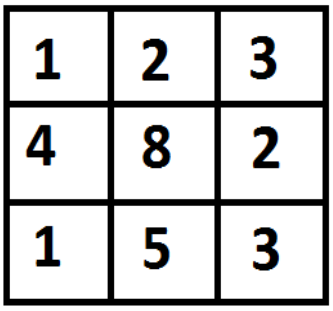
```
// up to any column in row 'n-1'
#include<bits/stdc++.h>
using namespace std;
#define N 4

// function find maximum sum path
int MaximumPath(int Mat[][N])
{
    int result = 0 ;

    // creat 2D matrix to store the sum
    // of the path
    int dp[N][N+2];

    // initialize all dp matrix as '0'
    memset(dp, 0, sizeof(dp));

    // copy all element of first column into
    // 'dp' first column
    for (int i = 0 ; i < N ; i++)
        dp[0][i+1] = Mat[0][i];

    for (int i = 1 ; i < N ; i++)
        for (int j = 1 ; j <= N ; j++)
            dp[i][j] = max(dp[i-1][j-1],
                           max(dp[i-1][j],
                               dp[i-1][j+1])) +
                       Mat[i][j-1] ;

    // Find maximum path sum that end ups
    // at  any column of last row 'N-1'
    for (int i=0; i<=N; i++)
        result = max(result, dp[N-1][i]);

    // return maximum sum path
    return result ;
}

// driver program to test above function
int main()
{
    int Mat[4][4] = { { 4, 2 , 3 , 4 },
        { 2 , 9 , 1 , 10},
        { 15, 1 , 3 , 0 },
        { 16 ,92, 41, 44 }
    };

    cout << MaximumPath ( Mat ) <<endl ;
    return 0;
}
```

59. Min Cost Path | DP-6

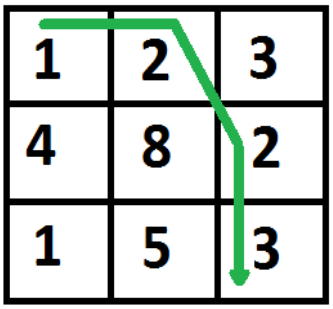
Last Updated: 04-05-2020

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)?



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).




```
/* 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;
}
```

60. Minimum number of jumps to reach end

Last Updated: 27-09-2020

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, they cannot move through that element.

Examples:

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1-> 3 -> 8 -> 9)
Explanation: Jump from 1st element
to 2nd element as there is only 1 step,
now there are three options 5, 8 or 9.
If 8 or 9 is chosen then the end node 9
can be reached. So 3 jumps are made.

Input:  arr[] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
Output: 10
Explanation: In every step a jump
is needed so the count of jumps is 10.
```

```
// C++ implementation of the approach
#include <bits/stdc++.h>
using namespace std;

// Function to return the minimum number
// of jumps to reach arr[h] from arr[l]
int minJumps(int arr[], int n)
```

```
{

    // Base case: when source and
    // destination are same
    if (n == 1)
        return 0;

    // Traverse through all the points
    // reachable from arr[l]
    // Recursivel, get the minimum number
    // of jumps needed to reach arr[h] from
    // these reachable points
    int res = INT_MAX;
    for (int i = n - 2; i >= 0; i--) {
        if (i + arr[i] >= n - 1) {
            int sub_res = minJumps(arr, i + 1);
            if (sub_res != INT_MAX)
                res = min(res, sub_res + 1);
        }
    }

    return res;
}

// Driver Code
int main()
{
    int arr[] = { 1, 3, 6, 3, 2,
                  3, 6, 8, 9, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Minimum number of jumps to";
    cout << " reach the end is " << minJumps(arr, n);
    return 0;
}
```

61. Minimum cost to fill given weight in a bag

Last Updated: 23-09-2020

You are given a bag of size W kg and you are provided costs of packets different weights of oranges in array cost[] where **cost[i]** is basically the cost of **'i'** kg packet of oranges. Where cost[i] = -1 means that **'i'** kg packet of orange is unavailable
Find the minimum total cost to buy exactly W kg oranges and if it is not possible to buy exactly W kg oranges then print -1. It may be assumed that there is an infinite supply of all available packet types.
Note: array starts from index 1.
Examples:

```
Input  : W = 5, cost[] = {20, 10, 4, 50, 100}

Output : 14
```

We can choose two oranges to minimize cost. First
orange of 2Kg and cost 10. Second orange of 3Kg
and cost 4.

Input : W = 5, cost[] = {1, 10, 4, 50, 100}
Output : 5
We can choose five oranges of weight 1 kg.

Input : W = 5, cost[] = {1, 2, 3, 4, 5}
Output : 5
Costs of 1, 2, 3, 4 and 5 kg packets are 1, 2, 3,
4 and 5 Rs respectively.
We choose packet of 5kg having cost 5 for minimum
cost to get 5Kg oranges.

Input : W = 5, cost[] = {-1, -1, 4, 5, -1}
Output : -1
Packets of size 1, 2 and 5 kg are unavailable
because they have cost -1. Cost of 3 kg packet
is 4 Rs and of 4 kg is 5 Rs. Here we have only
weights 3 and 4 so by using these two we can
not make exactly W kg weight, therefore answer
is -1.

```
// C++ program to find minimum cost to get exactly
// W Kg with given packets
#include<bits/stdc++.h>
#define INF 1000000
using namespace std;

// cost[] initial cost array including unavailable packet
// W capacity of bag
int MinimumCost(int cost[], int n, int W)
{
    // val[] and wt[] arrays
    // val[] array to store cost of 'i' kg packet of orange
    // wt[] array weight of packet of orange
    vector<int> val, wt;
```

```
    // traverse the original cost[] array and skip
    // unavailable packets and make val[] and wt[]
    // array. size variable tells the available number
    // of distinct weighted packets
    int size = 0;
    for (int i=0; i<n; i++)
    {
        if (cost[i]!= -1)
        {
            val.push_back(cost[i]);
            wt.push_back(i+1);
            size++;
        }
    }

    n = size;
    int min_cost[n+1][W+1];

    // fill 0th row with infinity
    for (int i=0; i<=W; i++)
        min_cost[0][i] = INF;

    // fill 0'th column with 0
    for (int i=1; i<=n; i++)
        min_cost[i][0] = 0;

    // now check for each weight one by one and fill the
    // matrix according to the condition
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=W; j++)
        {
            // wt[i-1]>j means capacity of bag is
            // less then weight of item
            if (wt[i-1] > j)
                min_cost[i][j] = min_cost[i-1][j];

            // here we check we get minimum cost either
            // by including it or excluding it
            else
                min_cost[i][j] = min(min_cost[i-1][j],
                     min_cost[i][j-wt[i-1]] + val[i-1]);
        }
    }

    // exactly weight W can not be made by given weights
    return (min_cost[n][W]==INF)? -1: min_cost[n][W];
}

// Driver program to run the test case
int main()
{
    int cost[] = {1, 2, 3, 4, 5}, W = 5;
    int n = sizeof(cost)/sizeof(cost[0]);
```

```
    cout << MinimumCost(cost, n, W);
    return 0;
}
```

62. Minimum sum of multiplications of n numbers

Last Updated: 19-12-2018

Given n integers. The task is to minimize the sum of multiplication of all the numbers by taking two adjacent numbers at a time and putting back their sum % 100 till a number is left.

Examples :

```
Input : 40 60 20

Output : 2400

Explanation: There are two possible cases:

1st possibility: Take 40 and 60, so multiplication=2400

and put back (60+40) % 100 = 0, making it 0, 20.

Multiplying 0 and 20 we get 0 so

multiplication = 2400+0 = 2400. Put back (0+20)%100 = 20.

2nd possibility: take 60 and 20, so 60*20 = 1200,

put back (60+20)%100 = 80, making it [40, 80]

multiply 40*80 to get 3200, so multiplication

sum = 1200+3200 = 4400. Put back (40+80)%100 = 20


Input : 5 6

Output : 30

Explanation: Only possibility is 5*6=30
```

```
// CPP program to find the
// minimum sum of multiplication
// of n numbers
#include <bits/stdc++.h>
using namespace std;

// Used in recursive
// memoized solution
long long dp[1000][1000];

// function to calculate the cumulative
// sum from a[i] to a[j]
```

```
long long sum(int a[], int i, int j)
{
    long long ans = 0;
    for (int m = i; m <= j; m++)
        ans = (ans + a[m]) % 100;
    return ans;
}

long long solve(int a[], int i, int j)
{
    // base case
    if (i == j)
        return 0;

    // memoization, if the partition
    // has been called before then
    // return the stored value
    if (dp[i][j] != -1)
        return dp[i][j];

    // store a max value
    dp[i][j] = INT_MAX;

    // we break them into k partitions
    for (int k = i; k < j; k++)
    {
        // store the min of the
        // formula thus obtained
        dp[i][j] = min(dp[i][j], (solve(a, i, k) +
                                  solve(a, k + 1, j) +
              (sum(a, i, k) * sum(a, k + 1, j))));
    }

    // return the minimum
    return dp[i][j];
}

void intialize(int n)
{
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= n; j++)
            dp[i][j] = -1;
}

// Driver code
int main() {
    int a[] = {40, 60, 20};
    int n = sizeof(a) / sizeof(a[0]);
    intialize(n);
    cout << solve(a, 0, n - 1) << endl;
    return 0;
}
```

63. Minimum removals from array to make max – min <= K

Last Updated: 24-09-2020

Given N integers and K, find the minimum number of elements that should be removed such that A_{max}-A_{min}<=K. After removal of elements, A_{max} and A_{min} is considered among the remaining elements.
Examples:

```
Input : a[] = {1, 3, 4, 9, 10, 11, 12, 17, 20}
          k = 4
Output : 5
Explanation: Remove 1, 3, 4 from beginning
and 17, 20 from the end.


Input : a[] = {1, 5, 6, 2, 8}  K=2
Output : 3
Explanation: There are multiple ways to
remove elements in this case.
One among them is to remove 5, 6, 8.
The other is to remove 1, 2, 5
```

```
// CPP program to find minimum removals
// to make max-min <= K
#include <bits/stdc++.h>
using namespace std;

#define MAX 100
int dp[MAX][MAX];

// function to check all possible combinations
// of removal and return the minimum one
int countRemovals(int a[], int i, int j, int k)
{
    // base case when all elements are removed
    if (i >= j)
        return 0;

    // if condition is satisfied, no more
    // removals are required
    else if ((a[j] - a[i]) <= k)
        return 0;
```

```
    // if the state has already been visited
    else if (dp[i][j] != -1)
        return dp[i][j];

    // when Amax-Amin>d
    else if ((a[j] - a[i]) > k) {

        // minimum is taken of the removal
        // of minimum element or removal
        // of the maximum element
        dp[i][j] = 1 + min(countRemovals(a, i + 1, j, k),
                           countRemovals(a, i, j - 1, k));
    }
    return dp[i][j];
}

// To sort the array and return the answer
int removals(int a[], int n, int k)
{
    // sort the array
    sort(a, a + n);

    // fill all stated with -1
    // when only one element
    memset(dp, -1, sizeof(dp));
    if (n == 1)
        return 0;
    else
        return countRemovals(a, 0, n - 1, k);
}

// Driver Code
int main()
{
    int a[] = { 1, 3, 4, 9, 10, 11, 12, 17, 20 };
    int n = sizeof(a) / sizeof(a[0]);
    int k = 4;
    cout << removals(a, n, k);
    return 0;
}
```

64. Minimum steps to minimize n as per given condition

Last Updated: 18-12-2018

Given a number n, count minimum steps to minimize it to 1 according to the following criteria:

- If n is divisible by 2 then we may reduce n to n/2.
- If n is divisible by 3 then you may reduce n to n/3.
- Decrement n by 1.

Examples:

```
Input : n = 10

Output : 3


Input : 6

Output : 2
```

```
// CPP program to minimize n to 1 by given
// rule in minimum steps
#include <bits/stdc++.h>
using namespace std;

// function to calculate min steps
int getMinSteps(int n, int *memo)
{
    // base case
    if (n == 1)
       return 0;
    if (memo[n] != -1)
       return memo[n];

    // store temp value for n as min( f(n-1),
    // f(n/2), f(n/3)) +1
    int res = getMinSteps(n-1, memo);

    if (n%2 == 0)
        res = min(res, getMinSteps(n/2, memo));
    if (n%3 == 0)
        res = min(res, getMinSteps(n/3, memo));

    // store memo[n] and return
    memo[n] = 1 + res;
    return memo[n];
}

// This function mainly initializes memo[] and
// calls getMinSteps(n, memo)
int getMinSteps(int n)
{
    int memo[n+1];

    // initialize memoized array
    for (int i=0; i<=n; i++)
        memo[i] = -1;

    return  getMinSteps(n, memo);
}

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

```
    int n = 10;
    cout << getMinSteps(n);
    return 0;
}
```

65. Edit Distance | DP-5

Last Updated: 13-01-2020

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

1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.

Examples:

```
Input:   str1 = "geek", str2 = "gesek"

Output:  1

We can convert str1 into str2 by inserting a 's'.


Input:   str1 = "cat", str2 = "cut"

Output:  1

We can convert str1 into str2 by replacing 'a' with 'u'.


Input:   str1 = "sunday", str2 = "saturday"

Output:  3

Last three and first characters are same.  We basically

need to convert "un" to "atur".  This can be done using

below three operations.

Replace 'n' with 'r', insert t, insert a
```

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

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

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

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

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

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

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

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

    return 0;
}
```

66. Minimum time to write characters using insert, delete and copy operation

Last Updated: 12-04-2019

We need to write N same characters on a screen and each time we can insert a character, delete the last character and copy and paste all written characters i.e. after copy operation count of total written character will become twice. Now we are given time for insertion, deletion and copying. We need to output minimum time to write N characters on the screen using these operations.

Examples:

```
Input : N = 9
        insert time = 1
        removal time = 2
        copy time = 1
Output : 5
N character can be written on screen in 5 time units as shown below,
insert a character
characters = 1  total time = 1
again insert character
characters = 2  total time = 2
copy characters
characters = 4  total time = 3
copy characters
characters = 8  total time = 4
insert character
characters = 9  total time = 5
```

```
// C++ program to write characters in
// minimum time by inserting, removing
// and copying operation
#include <bits/stdc++.h>
using namespace std;

//  method returns minimum time to write
// 'N' characters
int minTimeForWritingChars(int N, int insert,
                        int remove, int copy)
{
    if (N == 0)
       return 0;
    if (N == 1)
       return insert;

    //  declare dp array and initialize with zero
    int dp[N + 1];
    memset(dp, 0, sizeof(dp));

    //  loop for 'N' number of times
    for (int i = 1; i <= N; i++)
    {
        /*  if current char count is even then
            choose minimum from result for (i-1)
```

```
            chars and time for insertion and
            result for half of chars and time
            for copy  */
        if (i % 2 == 0)
            dp[i] = min(dp[i-1] + insert,
                        dp[i/2] + copy);

        /*  if current char count is odd then
            choose minimum from
            result for (i-1) chars and time for
            insertion and
            result for half of chars and time for
            copy and one extra character deletion*/
        else
            dp[i] = min(dp[i-1] + insert,
                        dp[(i+1)/2] + copy + remove);
    }
    return dp[N];
}

// Driver code to test above methods
int main()
{
    int N = 9;
    int insert = 1, remove = 2, copy = 1;
    cout << minTimeForWritingChars(N, insert,
                                   remove, copy);
    return 0;
}
```

67. Longest Common Substring | DP-29

Last Updated: 24-07-2019

Given two strings ‘X’ and ‘Y’, find the length of the longest common substring.

Examples :

Input : X = “GeeksforGeeks”, y = “GeeksQuiz”
Output : 5
The longest common substring is “Geeks” and is of length 5.

Input : X = “abcdxyz”, y = “xyzabcd”
Output : 4
The longest common substring is “abcd” and is of length 4.

Input : X = “zxabcdezy”, y = “yzabcdezx”
Output : 6
The longest common substring is “abcdez” and is of length 6.

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

```
/* 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.   Note that
    // LCSuff[i][j] contains length of longest
    // common suffix of X[0..i-1] and Y[0..j-1].

    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++)
        {

            // The first row and first column
            // entries have no logical meaning,
            // they are used only for simplicity
            // of program
            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;
}
```

68. Longest Common Substring (Space optimized DP solution)

Last Updated: 18-12-2018

Given two strings ‘X’ and ‘Y’, find the length of longest common substring. Expected space complexity is linear.

Examples :

```
Input : X = "GeeksforGeeks", Y = "GeeksQuiz"

Output : 5

The longest common substring is "Geeks" and is of

length 5.


Input : X = "abcdxyz", Y = "xyzabcd"

Output : 4

The longest common substring is "abcd" and is of

length 4.
```

```
// Space optimized CPP implementation of longest
// common substring.
#include <bits/stdc++.h>
using namespace std;

// Function to find longest common substring.
int LCSubStr(string X, string Y)
{
    // Find length of both the strings.
    int m = X.length();
    int n = Y.length();

    // Variable to store length of longest
    // common substring.
    int result = 0;

    // Matrix to store result of two
    // consecutive rows at a time.
    int len[2][n];

    // Variable to represent which row of
    // matrix is current row.
    int currRow = 0;

    // For a particular value of i and j,
    // len[currRow][j] stores length of longest
    // common substring in string X[0..i] and Y[0..j].
```

```
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                len[currRow][j] = 0;
            }
            else if (X[i - 1] == Y[j - 1]) {
                len[currRow][j] = len[1 - currRow][j - 1] + 1;
                result = max(result, len[currRow][j]);
            }
            else {
                len[currRow][j] = 0;
            }
        }

        // Make current row as previous row and previous
        // row as new current row.
        currRow = 1 - currRow;
    }

    return result;
}

int main()
{
    string X = "GeeksforGeeks";
    string Y = "GeeksQuiz";

    cout << LCSubStr(X, Y);
    return 0;
}
```

69. Sum of all substrings of a string representing a number | Set 1

Last Updated: 06-10-2020

Given an integer represented as a string, we need to get the sum of all possible substrings of this string.

Examples:

```
Input  : num = “1234”

Output : 1670

Sum = 1 + 2 + 3 + 4 + 12 + 23 +

       34 + 123 + 234 + 1234

    = 1670


Input  : num = “421”
```

```
Output : 491

Sum = 4 + 2 + 1 + 42 + 21 + 421 = 491
```

```
// C++ program to print sum of all substring of
// a number represented as a string
#include <bits/stdc++.h>
using namespace std;

// Utility method to convert character digit to
// integer digit
int toDigit(char ch)
{
    return (ch - '0');
}

// Returns sum of all substring of num
int sumOfSubstrings(string num)
{
    int n = num.length();

    // allocate memory equal to length of string
    int sumofdigit[n];

    // initialize first value with first digit
    sumofdigit[0] = toDigit(num[0]);
    int res = sumofdigit[0];

    // loop over all digits of string
    for (int i = 1; i < n; i++) {
        int numi = toDigit(num[i]);

        // update each sumofdigit from previous value
        sumofdigit[i] = (i + 1) * numi + 10 * sumofdigit[i - 1];

        // add current value to the result
        res += sumofdigit[i];
    }

    return res;
}

// Driver code to test above methods
int main()
{
    string num = "1234";
    cout << sumOfSubstrings(num) << endl;
    return 0;
}
```

70. Find n-th element from Stern's Diatomic Series

Last Updated: 02-07-2018

Given an integer n. we have to find the nth term of Stern's Diatomic Series.

Stern's diatomic series is the sequence which generates the following integer sequence 0, 1, 1, 2, 1, 3, 2, 3, 1, 4, 3, 5, 2, 5, 3, 4, ……. It arises in the Calkin-Wilf tree. It is sometimes also known as the **fusc** function.
In mathematical terms, the sequence P(n) of Stern's diatomic series is defined by the recurrence relation.

```
// Program to find the nth element
// of Stern's Diatomic Series
#include <bits/stdc++.h>
using namespace std;

// function to find nth stern'
// diatomic series
int findSDSFunc(int n)
{
    // Initializing the DP array
    int DP[n+1];

    // SET the Base case
    DP[0] = 0;
    DP[1] = 1;

    // Traversing the array from
    // 2nd Element to nth Element
    for (int i = 2; i <= n; i++) {

        // Case 1: for even n
        if (i % 2 == 0)
            DP[i] = DP[i / 2];

        // Case 2: for odd n
        else
            DP[i] = DP[(i - 1) / 2] +
                        DP[(i + 1) / 2];
    }
    return DP[n];
}

// Driver program
int main()
{
    int n = 15;
    cout << findSDSFunc(n) << endl;
    return 0;
}
```

71. Find maximum possible stolen value from houses

Last Updated: 20-03-2020

There are n houses build in a line, each of which contains some value in it. A thief is going to steal the maximal value of these houses, but he can't steal in two adjacent houses because the owner of the stolen houses will tell his two neighbours left and right side. What is the maximum stolen value?

Examples:

Input: hval[] = {6, 7, 1, 3, 8, 2, 4}
Output: 19

Explanation: The thief will steal 6, 1, 8 and 4 from the house.

Input: hval[] = {5, 3, 4, 11, 2}
Output: 16

Explanation: Thief will steal 5 and 11

```
// CPP program to find the maximum stolen value
#include <iostream>
using namespace std;

// calculate the maximum stolen value
int maxLoot(int *hval, int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return hval[0];
    if (n == 2)
        return max(hval[0], hval[1]);

    // dp[i] represent the maximum value stolen
    // so far after reaching house i.
    int dp[n];

    // Initialize the dp[0] and dp[1]
    dp[0] = hval[0];
    dp[1] = max(hval[0], hval[1]);

    // Fill remaining positions
    for (int i = 2; i<n; i++)
        dp[i] = max(hval[i]+dp[i-2], dp[i-1]);

    return dp[n-1];
}

// Driver to test above code
int main()
{
    int hval[] = {6, 7, 1, 3, 8, 2, 4};
    int n = sizeof(hval)/sizeof(hval[0]);
    cout << "Maximum loot possible : "
        << maxLoot(hval, n);
    return 0;
}
```

72. Find number of solutions of a linear equation of n variables

Last Updated: 12-06-2020

Given a linear equation of n variables, find number of non-negative integer solutions of it. For example,let the given equation be “x + 2y = 5”, solutions of this equation are “x = 1, y = 2”, “x = 5, y = 0” and “x = 1. It may be assumed that all coefficients in given equation are positive integers.

Example :

```
Input:  coeff[] = {1, 2}, rhs = 5

Output: 3

The equation "x + 2y = 5" has 3 solutions.

(x=3,y=1), (x=1,y=2), (x=5,y=0)


Input:  coeff[] = {2, 2, 3}, rhs = 4

Output: 3

The equation "2x + 2y + 3z = 4"  has 3 solutions.

(x=0,y=2,z=0), (x=2,y=0,z=0), (x=1,y=1,z=0)
```

```
// A naive recursive C++ program to
// find number of non-negative solutions
// for a given linear equation
#include<bits/stdc++.h>
using namespace std;

// Recursive function that returns
// count of solutions for given rhs
// value and coefficients coeff[start..end]
int countSol(int coeff[], int start,
             int end, int rhs)
{
    // Base case
    if (rhs == 0)
    return 1;

    // Initialize count
    // of solutions
    int result = 0;

    // One by subtract all smaller or
    // equal coefficiants and recur
    for (int i = start; i <= end; i++)
    if (coeff[i] <= rhs)
        result += countSol(coeff, i, end,
```

```
                            rhs - coeff[i]);

    return result;
}

// Driver Code
int main()
{
    int coeff[] = {2, 2, 5};
    int rhs = 4;
    int n = sizeof(coeff) / sizeof(coeff[0]);
    cout << countSol(coeff, 0, n - 1, rhs);
    return 0;
}
```

73. Count number of ways to reach a given score in a game

Last Updated: 12-07-2019

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)


Input: n = 13

Output: 2

There are following 2 ways to reach 13

(3, 5, 5)

(3, 10)
```

```
// A C++ program to count number of
// possible ways to a given score
// can be reached in a game where a
// move can earn 3 or 5 or 10
#include <iostream>
```

```
using namespace std;

// Returns number of ways
// to reach score n
int count(int n)
{
    // table[i] will store count
    // of solutions for value i.
    int table[n + 1], i;

    // Initialize all table
    // values as 0
    for(int j = 0; j < n + 1; j++)
            table[j] = 0;

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

    // 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];
}

// Driver Code
int main(void)
{
    int n = 20;
    cout << "Count for " << n
         << " is " << count(n) << endl;

    n = 13;
    cout <<"Count for "<< n<< " is "
         << count(n) << endl;
    return 0;
}
```

74. Count ways to reach the nth stair using step 1, 2 or 3

Last Updated: 28-09-2020

A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

Examples:

```
Input : 4
Output : 7
Explantion:
Below are the four ways
 1 step + 1 step + 1 step + 1 step
 1 step + 2 step + 1 step
 2 step + 1 step + 1 step
 1 step + 1 step + 2 step
 2 step + 2 step
 3 step + 1 step
 1 step + 3 step

Input : 3
Output : 4
Explantion:
Below are the four ways
 1 step + 1 step + 1 step
 1 step + 2 step
 2 step + 1 step
 3 step
```

```
// C++ Program to find n-th stair using step size
// 1 or 2 or 3.
#include <iostream>
using namespace std;

class GFG {

    // Returns count of ways to reach n-th stair
    // using 1 or 2 or 3 steps.
public:
    int findStep(int n)
    {
        if (n == 1 || n == 0)
            return 1;
        else if (n == 2)
            return 2;

        else
            return findStep(n - 3) + findStep(n - 2)
```

```
                                    + findStep(n - 1);
    }
};

// Driver code
int main()
{
    GFG g;
    int n = 4;
    cout << g.findStep(n);
    return 0;
}
```

75. Count of different ways to express N as the sum of 1, 3 and 4

Last Updated: 03-08-2018

Given N, count the number of ways to express N as sum of 1, 3 and 4.

Examples:

```
Input :  N = 4

Output : 4

Explanation: 1+1+1+1

             1+3

             3+1

             4


Input : N = 5

Output : 6

Explanation: 1 + 1 + 1 + 1 + 1

             1 + 4

             4 + 1

             1 + 1 + 3

             1 + 3 + 1

             3 + 1 + 1
```

```
// CPP program to illustrate the number of
// ways to represent N as sum of 1, 3 and 4.
#include <bits/stdc++.h>
```

```
using namespace std;

// function to count the number of
// ways to represent n as sum of 1, 3 and 4
int countWays(int n)
{
    int DP[n + 1];

    // base cases
    DP[0] = DP[1] = DP[2] = 1;
    DP[3] = 2;

    // iterate for all values from 4 to n
    for (int i = 4; i <= n; i++)
        DP[i] = DP[i - 1] + DP[i - 3] + DP[i - 4];

    return DP[n];
}

// driver code
int main()
{
    int n = 10;
    cout << countWays(n);
    return 0;
}
```

76. Count ways to build street under given constraints

Last Updated: 25-09-2020

There is a street of length n and as we know it has two sides. Therefore a total of 2 * n spots are available. In each of these spots either a house or an office can be built with following 2 restrictions:
1. No two offices on the same side of the street can be adjacent.
2. No two offices on different sides of the street can be exactly opposite to each other i.e. they can't overlook each other.
There are no restrictions on building houses and each spot must either have a house or office.
Given length of the street n, find total number of ways to build the street.
Examples:

```
Input : 2

Output : 7

Please see below diagram for explanation.


Input : 3
```

```
Output : 17
```

```
// C++ program to count ways to build street
// under given constraints
#include <bits/stdc++.h>
using namespace std;

// function to count ways of building
// a street of n rows
long countWays(int n)
{
    long dp[2][n + 1];

    // base case
    dp[0][1] = 1;
    dp[1][1] = 2;

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

        // ways of building houses in both
        // the spots of ith row
        dp[0][i] = dp[0][i - 1] + dp[1][i - 1];

        // ways of building an office in one of
        // the two spots of ith row
        dp[1][i] = dp[0][i - 1] * 2 + dp[1][i - 1];
    }

    // total ways for n rows
    return dp[0][n] + dp[1][n];
}

// driver program for checking above function
int main()
{

    int n = 5;
    cout << "Total no of ways with n = " << n
         << " are: " << countWays(n) << endl;
}
```

77. Count Balanced Binary Trees of Height h

Last Updated: 17-12-2018

Given a height h, count and return the maximum number of balanced binary trees possible with height h. A balanced binary tree is one in which for every node, the difference between heights of left and right subtree is not more than 1.

Examples :

```
Input : h = 3

Output : 15
```

```
Input : h = 4

Output : 315
```

```
// C++ program to count number of balanced
// binary trees of height h.
#include <bits/stdc++.h>
#define mod 1000000007
using namespace std;

long long int countBT(int h) {

    long long int dp[h + 1];
    //base cases
    dp[0] = dp[1] = 1;
    for(int i = 2; i <= h; i++) {
        dp[i] = (dp[i - 1] * ((2 * dp [i - 2])%mod + dp[i - 1])%mod) % mod;
    }
    return dp[h];
}


// Driver program
int main()
{
    int h = 3;
    cout << "No. of balanced binary trees"
            " of height h is: "
         << countBT(h) << endl;
}
```

78. Counting pairs when a person can form pair with at most one

Last Updated: 18-05-2020

Consider a coding competition on geeksforgeeks practice. Now their are **n** distinct participants taking part in the competition. A single participant can make pair with at most one other participant. We need count the number of ways in which **n** participants participating in the coding competition.

Examples :

```
Input : n = 2

Output : 2

2 shows that either both participant

can pair themselves in one way or both

of them can remain single.
```

```
Input : n = 3

Output : 4

One way : Three participants remain single

Three More Ways : [(1, 2)(3)], [(1), (2,3)]

and [(1,3)(2)]
```

```
// Number of ways in which participant can take part.
#include<iostream>
using namespace std;

int numberOfWays(int x)
{
    // Base condition
    if (x==0 || x==1)
        return 1;

    // A participant can choose to consider
    // (1) Remains single. Number of people
    //     reduce to (x-1)
    // (2) Pairs with one of the (x-1) others.
    //     For every pairing, number of people
    //     reduce to (x-2).
    else
        return numberOfWays(x-1) +
               (x-1)*numberOfWays(x-2);
}

// Driver code
int main()
{
    int x = 3;
    cout << numberOfWays(x) << endl;
    return 0;
}
```

79. Counts paths from a point to reach Origin

Last Updated: 13-10-2020

You are standing on a point **(n, m)** and you want to go to origin **(0, 0)** by taking steps either **left or down** i.e. from each point you are allowed to move either in **(n-1, m)** or **(n, m-1)**. Find the number of paths from point to origin.

Examples:

```
Input : 3 6

Output : Number of Paths 84
```

```
Input : 3 0

Output : Number of Paths 1
```

```
// CPP program to count total number of
// paths from a point to origin
#include<bits/stdc++.h>
using namespace std;

// Recursive function to count number of paths
int countPaths(int n, int m)
{
    // If we reach bottom or top left, we are
    // have only one way to reach (0, 0)
    if (n==0 || m==0)
        return 1;

    // Else count sum of both ways
    return (countPaths(n-1, m) + countPaths(n, m-1));
}

// Driver Code
int main()
{
    int n = 3, m = 2;
    cout << " Number of Paths " << countPaths(n, m);
    return 0;
}
```

80. Count number of ways to cover a distance

Given a distance ‘dist, count total number of ways to cover the distance with 1, 2 and 3 steps.

Examples:

```
Input: n = 3
Output: 4
Explantion:
Below are the four ways
 1 step + 1 step + 1 step
 1 step + 2 step
 2 step + 1 step
 3 step

Input: n = 4
Output: 7
Explantion:
Below are the four ways
 1 step + 1 step + 1 step + 1 step
 1 step + 2 step + 1 step
 2 step + 1 step + 1 step
 1 step + 1 step + 2 step
```

```
 2 step + 2 step
 3 step + 1 step
 1 step + 3 step
```

```
// A naive recursive C++ program to count number of ways to cover
// a distance with 1, 2 and 3 steps
#include<iostream>
using namespace std;

// Returns count of ways to cover 'dist'
int printCountRec(int dist)
{
    // Base cases
    if (dist<0)      return 0;
    if (dist==0)  return 1;

    // Recur for all previous 3 and add the results
    return printCountRec(dist-1) +
           printCountRec(dist-2) +
           printCountRec(dist-3);
}

// driver program
int main()
{
    int dist = 4;
    cout << printCountRec(dist);
    return 0;
}
```

81. Count ways to divide circle using N non-intersecting chords

Given a number N, find the number of ways you can draw N chords in a circle with 2*N points such that no 2 chords intersect.
Two ways are different if there exists a chord which is present in one way and not in other.

Examples:

```
Input : N = 2

Output : 2

Explanation: If points are numbered 1 to 4 in

clockwise direction, then different ways to

draw chords are:

{(1-2), (3-4)} and {(1-4), (2-3)}
```

```
Input : N = 1

Output : 1

Explanation: Draw a chord between points 1 and 2.
```

```
// cpp code to count ways
// to divide circle using
// N non-intersecting chords.
#include <bits/stdc++.h>
using namespace std;

int chordCnt( int A){

    // n = no of points required
    int n = 2 * A;

    // dp array containing the sum
    int dpArray[n + 1]={ 0 };
    dpArray[0] = 1;
    dpArray[2] = 1;
    for (int i=4;i<=n;i+=2){
        for (int j=0;j<i-1;j+=2){

          dpArray[i] +=
            (dpArray[j]*dpArray[i-2-j]);
        }
    }

    // returning the required number
    return dpArray[n];
}
// Driver function
int main()
{

    int N;
    N = 2;
cout<<chordCnt( N)<<'\n';
    N = 1;
cout<<chordCnt( N)<<'\n';
    N = 4;
cout<<chordCnt( N)<<'\n';
    return 0;
}
```

82. Count the number of ways to tile the floor of size n x m using 1 x m size tiles

Given a floor of size n x m and tiles of size 1 x m. The problem is to count the number of ways to tile the given floor using 1 x m tiles. A tile can either be placed horizontally or

vertically.
Both n and m are positive integers and 2 < = m.

Examples:

```
Input : n = 2, m = 3

Output : 1

Only one combination to place
two tiles of size 1 x 3 horizontally
on the floor of size 2 x 3.

Input :  n = 4, m = 4
Output : 2
1st combination:
All tiles are placed horizontally
2nd combination:
All tiles are placed vertically.
```

```
// C++ implementation to count number of ways to
// tile a floor of size n x m using 1 x m tiles
#include <bits/stdc++.h>

using namespace std;

// function to count the total number of ways
int countWays(int n, int m)
{

    // table to store values
    // of subproblems
    int count[n + 1];
    count[0] = 0;

    // Fill the table upto value n
    for (int i = 1; i <= n; i++) {

        // recurrence relation
        if (i > m)
            count[i] = count[i - 1] + count[i - m];

        // base cases and for i = m = 1
        else if (i < m || i == 1)
            count[i] = 1;

        // i = = m
        else
            count[i] = 2;
    }

    // required number of ways
    return count[n];
}
```

```
// Driver program to test above
int main()
{
    int n = 7, m = 4;
    cout << "Number of ways = "
         << countWays(n, m);
    return 0;
}
```

83. 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***

Examples :

```
Input :  m = 2, n = 2;

Output : 2

There are two paths

(0, 0) -> (0, 1) -> (1, 1)

(0, 0) -> (1, 0) -> (1, 1)


Input :  m = 2, n = 3;

Output : 3

There are three paths

(0, 0) -> (0, 1) -> (0, 2) -> (1, 2)

(0, 0) -> (0, 1) -> (1, 1) -> (1, 2)

(0, 0) -> (1, 0) -> (1, 1) -> (1, 2)
```

```
// A C++  program to count all possible paths
// from top left to bottom right

#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;
}
```

84. Largest Sum Contiguous Subarray

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

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

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

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

        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}

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

85. Smallest sum contiguous subarray

Given an array containing **n** integers. The problem is to find the sum of the elements of the contiguous subarray having the smallest(minimum) sum.
Examples:

```
Input : arr[] = {3, -4, 2, -3, -1, 7, -5}

Output : -6

Subarray is {-4, 2, -3, -1} = -6


Input : arr = {2, 6, 8, 1, 4}
Output : 1
```

```
// C++ implementation to find the smallest sum
// contiguous subarray
#include <bits/stdc++.h>

using namespace std;

// function to find the smallest sum contiguous subarray
int smallestSumSubarr(int arr[], int n)
{
    // to store the minimum value that is ending
    // up to the current index
    int min_ending_here = INT_MAX;

    // to store the minimum value encountered so far
    int min_so_far = INT_MAX;

    // traverse the array elements
    for (int i=0; i<n; i++)
    {
        // if min_ending_here > 0, then it could not possibly
        // contribute to the minimum sum further
        if (min_ending_here > 0)
            min_ending_here = arr[i];

        // else add the value arr[i] to min_ending_here
        else
            min_ending_here += arr[i];

        // update min_so_far
        min_so_far = min(min_so_far, min_ending_here);
    }

    // required smallest sum contiguous subarray value
    return min_so_far;
}


// Driver program to test above
int main()
{
    int arr[] = {3, -4, 2, -3, -1, 7, -5};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    cout << "Smallest sum: "
         << smallestSumSubarr(arr, n);
    return 0;
}
```

86. Size of array after repeated deletion of LIS

Given an array arr[0..n-1] of the positive element. The task is to print the remaining elements of arr[] after repeated deletion of LIS (of size greater than 1). If there are multiple LIS with the same length, we need to choose the LIS that ends first.

Examples:

```
Input : arr[] = {1, 2, 5, 3, 6, 4, 1}

Output : 1

Explanation :

{1, 2, 5, 3, 6, 4, 1} - {1, 2, 5, 6} = {3, 4, 1}

{3, 4, 1} - {3, 4} = {1}


Input : arr[] = {1, 2, 3, 1, 5, 2}

Output : -1

Explanation :

{1, 2, 3, 1, 5, 2} - {1, 2, 3, 5} = {1, 2}

{1, 2} - {1, 2} = {}


Input : arr[] = {5, 3, 2}

Output : 3
```

```
/* C++ program to find size of array after repeated
  deletion of LIS */
#include <bits/stdc++.h>
using namespace std;

// Function to construct Maximum Sum LIS
vector<int> findLIS(vector<int> arr, int n)
{
    // L[i] - The Maximum Sum Increasing
    // Subsequence that ends with arr[i]
    vector <vector<int> > L(n);

    // L[0] is equal to arr[0]
    L[0].push_back(arr[0]);

    // start from index 1
```

```
    for (int i = 1; i < n; i++)
    {
        // for every j less than i
        for (int j = 0; j < i; j++)
        {
            /* L[i] = {MaxSum(L[j])} + arr[i]
            where j < i and arr[j] < arr[i] */
            if (arr[i] > arr[j] && (L[i].size() < L[j].size()))
                L[i] = L[j];
        }

        // L[i] ends with arr[i]
        L[i].push_back(arr[i]);
    }

    // set lis =  LIS
    // whose size is max among all
    int maxSize = 1;
    vector<int> lis;
    for (vector<int> x : L)
    {
        // The > sign makes sure that the LIS
        // ending first is chose.
        if (x.size() > maxSize)
        {
            lis = x;
            maxSize = x.size();
        }
    }

    return lis;
}

// Function to minimize array
void minimize(int input[], int n)
{
    vector<int> arr(input, input + n);

    while (arr.size())
    {
        // Find LIS of current array
        vector<int> lis = findLIS(arr, arr.size());

        // If all elements are in decreasing order
        if (lis.size() < 2)
            break;

        // Remove lis elements from current array. Note
        // that both lis[] and arr[] are sorted in
        // increasing order.
        for (int i=0; i<arr.size() && lis.size()>0; i++)
        {
            // If first element of lis[] is found
            if (arr[i] == lis[0])
            {
```

```
                // Remove lis element from arr[]
                arr.erase(arr.begin()+i) ;
                i--;

                // Erase first element of lis[]
                lis.erase(lis.begin()) ;
            }
        }
    }

    // print remaining element of array
    int i;
    for (i=0; i < arr.size(); i++)
        cout  << arr[i] << " ";

    // print -1 for empty array
    if (i == 0)
        cout << "-1";
}

// Driver function
int main()
{
    int input[] = { 3, 2, 6, 4, 5, 1 };
    int n = sizeof(input) / sizeof(input[0]);

    // minimize array after deleting LIS
    minimize(input, n);

    return 0;
}
```

87. Remove array end element to maximize the sum of product

Given an array of **N** positive integers. We are allowed to remove element from either of the two ends i.e from the left side or right side of the array. Each time we remove an element, score is increased by value of element * (number of element already removed + 1). The task is to find the maximum score that can be obtained by removing all the element.

Examples:

```
Input : arr[] = { 1, 3, 1, 5, 2 }.

Output : 43

Remove 1 from left side (score = 1*1 = 1)

then remove 2, score = 1 + 2*2 = 5
```

then remove 3, score = 5 + 3*3 = 14

then remove 1, score = 14 + 1*4 = 18

then remove 5, score = 18 + 5*5 = 43.

Input : arr[] = { 1, 2 }

Output : 5.

```
// CPP program to find maximum score we can get
// by removing elements from either end.
#include <bits/stdc++.h>
#define MAX 50
using namespace std;

int solve(int dp[][MAX], int a[], int low, int high,int turn)
{
    // If only one element left.
    if (low == high)
        return a[low] * turn;

    // If already calculated, return the value.
    if (dp[low][high] != 0)
        return dp[low][high];

    // Computing Maximum value when element at
    // index i and index j is to be chosed.
    dp[low][high] = max(a[low] * turn + solve(dp, a,low + 1, high, turn + 1),
                        a[high] * turn + solve(dp, a,low, high - 1, turn+1));

    return dp[low][high];
}

// Driven Program
int main()
{
    int arr[] = { 1, 3, 1, 5, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    int dp[MAX][MAX];
    memset(dp, 0, sizeof(dp));

    cout << solve(dp, arr, 0, n - 1, 1) << endl;
    return 0;
}
```

88. Ways to sum to N using array elements with repetition allowed

Given a set of m distinct positive integers and a value ‘N’. The problem is to count the total number of ways we can form ‘N’ by doing sum of the array elements. Repetitions and different arrangements are allowed.

Examples :

```
Input : arr = {1, 5, 6}, N = 7

Output : 6


Explanation:- The different ways are:

1+1+1+1+1+1+1
1+1+5
1+5+1
5+1+1
1+6
6+1

Input : arr = {12, 3, 1, 9}, N = 14
Output : 150
```

```
// C++ implementation to count ways
// to sum up to a given value N
#include <bits/stdc++.h>

using namespace std;

// function to count the total
// number of ways to sum up to 'N'
int countWays(int arr[], int m, int N)
{
    int count[N + 1];
    memset(count, 0, sizeof(count));

    // base case
    count[0] = 1;

    // count ways for all values up
    // to 'N' and store the result
    for (int i = 1; i <= N; i++)
        for (int j = 0; j < m; j++)

            // if i >= arr[j] then
            // accumulate count for value 'i' as
            // ways to form value 'i-arr[j]'
            if (i >= arr[j])
                count[i] += count[i - arr[j]];

    // required number of ways
    return count[N];

}
```

```
// Driver code
int main()
{
    int arr[] = {1, 5, 6};
    int m = sizeof(arr) / sizeof(arr[0]);
    int N = 7;
    cout << "Total number of ways = "
        << countWays(arr, m, N);
    return 0;
}
```

89. Unique paths in a Grid with Obstacles

Given a grid of size m * n, let us assume you are starting at (1, 1) and your goal is to reach (m, n). At any instance, if you are on (x, y), you can either go to (x, y + 1) or (x + 1, y).
Now consider if some obstacles are added to the grids. How many unique paths would there be?
An obstacle and empty space are marked as 1 and 0 respectively in the grid.

Examples:

```
Input: [[0, 0, 0],
        [0, 1, 0],
        [0, 0, 0]]
Output : 2
There is only one obstacle in the middle.
```

```
// C++ code to find number of unique paths
// in a Matrix
#include<bits/stdc++.h>
using namespace std;

int uniquePathsWithObstacles(vector<vector<int>>& A)
{

    int r = A.size(), c = A[0].size();

    // create a 2D-matrix and initializing
    // with value 0
    vector<vector<int>> paths(r, vector<int>(c, 0));

    // Initializing the left corner if
    // no obstacle there
    if (A[0][0] == 0)
        paths[0][0] = 1;

    // Initializing first column of
    // the 2D matrix
```

```
    for(int i = 1; i < r; i++)
    {
        // If not obstacle
        if (A[i][0] == 0)
            paths[i][0] = paths[i-1][0];
    }

    // Initializing first row of the 2D matrix
    for(int j = 1; j < c; j++)
    {

        // If not obstacle
        if (A[0][j] == 0)
            paths[0][j] = paths[0][j - 1];
    }

    for(int i = 1; i < r; i++)
    {
        for(int j = 1; j < c; j++)
        {

            // If current cell is not obstacle
            if (A[i][j] == 0)
                paths[i][j] = paths[i - 1][j] +
                              paths[i][j - 1];
        }
    }

    // Returning the corner value
    // of the matrix
    return paths[r - 1];
}

// Driver code
int main()
{
   vector<vector<int>> A = { { 0, 0, 0 },
                             { 0, 1, 0 },
                             { 0, 0, 0 } };

   cout<<uniquePathsWithObstacles(A)<<\n";
}
```

90. Number of n-digits non-decreasing integers

Given an integer n > 0, which denotes the number of digits, the task to find total number of n-digit positive integers which are non-decreasing in nature.
A non-decreasing integer is a one in which all the digits from left to right are in non-decreasing form. ex: 1234, 1135, ..etc.
Note : Leading zeros also count in non-decreasing integers such as 0000, 0001, 0023, etc are also non-decreasing integers of 4-digits.
Examples :

Input : n = 1

Output : 10

Numbers are 0, 1, 2, ...9.

Input : n = 2

Output : 55

Input : n = 4

Output : 715

```
// C++ program for counting n digit numbers with
// non decreasing digits
#include <bits/stdc++.h>
using namespace std;

// Returns count of non- decreasing numbers with
// n digits.
int nonDecNums(int n)
{
    /* a[i][j] = count of all possible number
       with i digits having leading digit as j */
    int a[n + 1][10];

    // Initialization of all 0-digit number
    for (int i = 0; i <= 9; i++)
        a[0][i] = 1;

    /* Initialization of all i-digit
      non-decreasing number leading with 9*/
    for (int i = 1; i <= n; i++)
        a[i][9] = 1;

    /* for all digits we should calculate
      number of ways depending upon leading
      digits*/
    for (int i = 1; i <= n; i++)
        for (int j = 8; j >= 0; j--)
            a[i][j] = a[i - 1][j] + a[i][j + 1];

    return a[n][0];
}

// driver program
int main()
{
    int n = 2;
    cout << "Non-decreasing digits = "
         << nonDecNums(n) << endl;
    return 0;
```

```
}
```

91. Number of ways to arrange N items under given constraints

We are given N items which are of total K different colors. Items of the same color are indistinguishable and colors can be numbered from 1 to K and count of items of each color is also given as k1, k2 and so on. Now we need to arrange these items one by one under a constraint that the last item of color i comes before the last item of color (i + 1) for all possible colors. Our goal is to find out how many ways this can be achieved.

Examples:

```
Input : N = 3
        k1 = 1    k2 = 2
Output : 2
Explanation :
Possible ways to arrange are,
k1, k2, k2
k2, k1, k2


Input : N = 4
        k1 = 2    k2 = 2
Output : 3
Explanation :
Possible ways to arrange are,
k1, k2, k1, k2
k1, k1, k2, k2
k2, k1, k1, k2
```

```
// C++ program to find number of ways to arrange
// items under given constraint
#include <bits/stdc++.h>
using namespace std;

// method returns number of ways with which items
// can be arranged
int waysToArrange(int N, int K, int k[])
{
    int C[N + 1][N + 1];
    int i, j;
```

```
    // Calculate value of Binomial Coefficient in
    // bottom up manner
    for (i = 0; i <= N; i++) {
        for (j = 0; j <= i; j++) {

            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

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

    // declare dp array to store result up to ith
    // colored item
    int dp[K];

    // variable to keep track of count of items
    // considered till now
    int count = 0;

    dp[0] = 1;

    // loop over all different colors
    for (int i = 0; i < K; i++) {

        // populate next value using current value
        // and stated relation
        dp[i + 1] = (dp[i] * C[count + k[i] - 1][k[i] - 1]);
        count += k[i];
    }

    // return value stored at last index
    return dp[K];
}

// Driver code to test above methods
int main()
{
    int N = 4;
    int k[] = { 2, 2 };

    int K = sizeof(k) / sizeof(int);
    cout << waysToArrange(N, K, k) << endl;
    return 0;
}
```

92. Probability of reaching a point with 2 or 3 steps at a time

A person starts walking from position X = 0, find the probability to reach exactly on X = N if she can only take either 2 steps or 3 steps. Probability for step length 2 is given i.e. P, probability for step length 3 is 1 – P.

Examples :

```
Input : N = 5, P = 0.20

Output : 0.32

Explanation :-
There are two ways to reach 5.
2+3 with probability = 0.2 * 0.8 = 0.16
3+2 with probability = 0.8 * 0.2 = 0.16
So, total probability = 0.32.
```

```
// CPP Program to find probability to
// reach N with P probability to take
// 2 steps (1-P) to take 3 steps
#include <bits/stdc++.h>
using namespace std;

// Returns probability to reach N
float find_prob(int N, float P)
{
    double dp[N + 1];
    dp[0] = 1;
    dp[1] = 0;
    dp[2] = P;
    dp[3] = 1 - P;
    for (int i = 4; i <= N; ++i)
        dp[i] = (P)*dp[i - 2] + (1 - P) * dp[i - 3];

    return dp[N];
}

// Driver code
int main()
{
    int n = 5;
    float p = 0.2;
    cout << find_prob(n, p);
    return 0;
}
```

93. Value of continuous floor function : F(x) = F(floor(x/2)) + x

Given an array of positive integers. For every element x of array, we need to find the value of continuous floor function defined as **F(x) = F(floor(x/2)) + x**, where F(0) = 0.

Examples :-

```
Input : arr[] = {6, 8}

Output : 10 15


Explanation : F(6) = 6 + F(3)
                   = 6 + 3 + F(1)
                   = 6 + 3 + 1 + F(0)
                   = 10

Similarly F(8) = 15
```

```
#include <bits/stdc++.h>

#define max 10000
using namespace std;

int dp[max];

void initDP()
    {
         for (int i = 0; i < max; i++)
             dp[i] = -1;
    }

// function to return value of F(n)
int func(int x)
    {
        if (x == 0)
            return 0;
        if (dp[x] == -1)
            dp[x] = x + func(x / 2);

        return dp[x];
    }

void printFloor(int arr[], int n)
    {
        for (int i = 0; i < n; i++)
            cout << func(arr[i]) << " ";
    }

// Driver code
int main()
    {
        // call the initDP() to fill DP array
        initDP();
```

```
        int arr[] = { 8, 6 };
        int n = sizeof(arr) / sizeof(arr[0]);

        printFloor(arr, n);

        return 0;
    }
```

94. Number of decimal numbers of length k, that are strict monotone

We call decimal number a monotone if:

Write a program which takes positive number n on input and returns number of decimal numbers of length n that are strict monotone. Number can't start with 0.

Examples :

```
Input : 2

Output : 36

Numbers are 12, 13, ... 19, 23

24, ... 29, .... 89.


Input : 3

Output : 84
```

```
// CPP program to count numbers of k
// digits that are strictly monotone.
#include <cstring>
#include <iostream>

int static const DP_s = 9;

int getNumStrictMonotone(int len)
{
    // DP[i][j] is going to store monotone
    // numbers of length i+1 considering
    // j+1 digits (1, 2, 3, ..9)
    int DP[len][DP_s];
    memset(DP, 0, sizeof(DP));

    // Unit length numbers
    for (int i = 0; i < DP_s; ++i)
        DP[0][i] = i + 1;

    // Building dp[] in bottom up
    for (int i = 1; i < len; ++i)
        for (int j = 1; j < DP_s; ++j)
```

```
            DP[i][j] = DP[i - 1][j - 1] + DP[i][j - 1];

    return DP[len - 1][DP_s - 1];
}

// Driver code
int main()
{
    std::cout << getNumStrictMonotone(2);
    return 0;
}
```

95. Different ways to sum n using numbers greater than or equal to m

Given two natural number **n** and **m**. The task is to find the number of ways in which the numbers that are greater than or equal to m can be added to get the sum n.
Examples:

```
Input : n = 3, m = 1

Output : 3

Following are three different ways

to get sum n such that each term is

greater than or equal to m

1 + 1 + 1, 1 + 2, 3


Input : n = 2, m = 1

Output : 2

Two ways are 1 + 1 and 2
```

```
// CPP Program to find number of ways to
// which numbers that are greater than
// given number can be added to get sum.
#include <bits/stdc++.h>
#define MAX 100
using namespace std;

// Return number of ways to which numbers
// that are greater than given number can
// be added to get sum.
int numberofways(int n, int m)
{
    int dp[n+2][n+2];
    memset(dp, 0, sizeof(dp));

    dp[0][n + 1] = 1;
```

```
    // Filling the table. k is for numbers
    // greater than or equal that are allowed.
    for (int k = n; k >= m; k--) {

        // i is for sum
        for (int i = 0; i <= n; i++) {

            // initializing dp[i][k] to number
            // ways to get sum using numbers
            // greater than or equal k+1
            dp[i][k] = dp[i][k + 1];

            // if i > k
            if (i - k >= 0)
                dp[i][k] = (dp[i][k] + dp[i - k][k]);
        }
    }

    return dp[n][m];
}

// Driver Program
int main()
{
    int n = 3, m = 1;
    cout << numberofways(n, m) << endl;
    return 0;
}
```