**Dynamic Problems Property:  
  
1) Overlapping Sub-problems:**

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same sub-problems are needed again and again. In dynamic programming, computed solutions to sub-problems are stored in a table so that these don’t have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) sub-problems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn’t have common sub-problems. If we take example of following recursive program for Fibonacci Numbers, there are many sub-problems which are solved again and again.

1. **Optimal Substructure:**

A given problem has optimal sub-structure problem if optimal solution of the given problem can be obtained by using optimal solution of the sub problems.

**Fibonacci Numbers:**

**Binomial Coefficient:  
  
Base case:  
  
  
  
Recursive case:  
  
**

**Permutation Coefficient:**

**Base case:  
  
**

**Recursive case:**

**  
  
Longest Common Subsequence:**Let’s suppose, we are finding longest common sub-sequence for str1 and str2

Now, str1.length is m and str2.length is n.

LCS[I][j]=LCS[i-1][j-1] if str1[I]==str2[j]

LCS[i][j]=max(LCS[I][j-1],LCS[i-1][j]) if str1[I]!=str2[j]

**Base case:**

LCS[0][j]=0; for 

LCS[I][0]=0; for 

Later, you will know the optimized versions. Like memory saving version: because, only last two rows are required.

**Longest Repeating Subsequence:**a variant of LCS. This is basically LCS(str,str) with one condition. If str[I]==str[j] , i!=j **Ugly Numbers:**

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … shows the first 11 ugly numbers. By convention, 1 is

included.

1 Declare an array for ugly numbers: ugly[n]

2 Initialize first ugly no: ugly[0] = 1

3 Initialize three array index variables i2, i3, i5 to point to

1st element of the ugly array:

i2 = i3 = i5 =0;

4 Initialize 3 choices for the next ugly no:

next\_mulitple\_of\_2 = ugly[i2]\*2;

next\_mulitple\_of\_3 = ugly[i3]\*3

next\_mulitple\_of\_5 = ugly[i5]\*5;

5 Now go in a loop to fill all ugly numbers till 150:

For (i = 1; i < 150; i++ )

{

/\* These small steps are not optimized for good

readability. Will optimize them in C program \*/

next\_ugly\_no = Min(next\_mulitple\_of\_2,

next\_mulitple\_of\_3,

next\_mulitple\_of\_5);

ugly[i] = next\_ugly\_no

if (next\_ugly\_no == next\_mulitple\_of\_2)

{

i2 = i2 + 1;

next\_mulitple\_of\_2 = ugly[i2]\*2;

}

if (next\_ugly\_no == next\_mulitple\_of\_3)

{

i3 = i3 + 1;

next\_mulitple\_of\_3 = ugly[i3]\*3;

}

if (next\_ugly\_no == next\_mulitple\_of\_5)

{

i5 = i5 + 1;

next\_mulitple\_of\_5 = ugly[i5]\*5;

}

}/\* end of for loop \*/

6.return next\_ugly\_no

**What could be the wrong design in this case?**

1 Declare an array for ugly numbers: ugly[n]

2 Initialize first ugly no: ugly[0] = 1

3 Initialize three array index variables i2, i3, i5 to point to

1st element of the ugly array:

i2 = i3 = i5 =0;

4 Initialize 3 choices for the next ugly no:

next\_mulitple\_of\_2 = ugly[i2]\*2;

next\_mulitple\_of\_3 = ugly[i3]\*3

next\_mulitple\_of\_5 = ugly[i5]\*5;

5 Now go in a loop to fill all ugly numbers till 150:

For (i = 1; i < 150; i++ )

{

/\* These small steps are not optimized for good

readability. Will optimize them in C program \*/

next\_ugly\_no = Min(next\_mulitple\_of\_2,

next\_mulitple\_of\_3,

next\_mulitple\_of\_5);

ugly[i] = next\_ugly\_no

if (next\_ugly\_no == next\_mulitple\_of\_2)

{

i2 = i2 + 1;

next\_mulitple\_of\_2 = ugly[i2]\*2;

}

else if (next\_ugly\_no == next\_mulitple\_of\_3)

{

i3 = i3 + 1;

next\_mulitple\_of\_3 = ugly[i3]\*3;

}

else if (next\_ugly\_no == next\_mulitple\_of\_5)

{

i5 = i5 + 1;

next\_mulitple\_of\_5 = ugly[i5]\*5;

}

}/\* end of for loop \*/

6.return next\_ugly\_no

**Why?**

initialize

ugly[] = | 1 |

i2 = i3 = i5 = 0;

First iteration

ugly[1] = Min(ugly[i2]\*2, ugly[i3]\*3, ugly[i5]\*5)

= Min(2, 3, 5)

= 2

ugly[] = | 1 | 2 |

i2 = 1, i3 = i5 = 0 (i2 got incremented )

Second iteration

ugly[2] = Min(ugly[i2]\*2, ugly[i3]\*3, ugly[i5]\*5)

= Min(4, 3, 5)

= 3

ugly[] = | 1 | 2 | 3 |

i2 = 1, i3 = 1, i5 = 0 (i3 got incremented )

Third iteration

ugly[3] = Min(ugly[i2]\*2, ugly[i3]\*3, ugly[i5]\*5)

= Min(4, 6, 5)

= 4

ugly[] = | 1 | 2 | 3 | 4 |

i2 = 2, i3 = 1, i5 = 0 (i2 got incremented )

Fourth iteration

ugly[4] = Min(ugly[i2]\*2, ugly[i3]\*3, ugly[i5]\*5)

= Min(6, 6, 5)

= 5

ugly[] = | 1 | 2 | 3 | 4 | 5 |

i2 = 2, i3 = 1, i5 = 1 (i5 got incremented )

Fifth iteration

ugly[4] = Min(ugly[i2]\*2, ugly[i3]\*3, ugly[i5]\*5)

= Min(6, 6, 10)

= 6

ugly[] = | 1 | 2 | 3 | 4 | 5 | 6 |

i2 = 3, i3 = 2, i5 = 1 (i2 and i3 got incremented )

**(Otherwise, problem will be generated in steps like this)**

Will continue same way till I < 150

**Longest Increasing Subsequence:**

**Problem Statement:**

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, 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}.

**Solution:**

Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

Then, L(i) can be recursively written as:

L(i) = 1 + max( L(j) ) where 0 < j < i and arr[j] < arr[i]; or

L(i) = 1, if no such j exists.

**Maximum length of chain of Pairs:**a standard variant of LIS. Following is a simple two step:  
  
1) sort given pairs in increasing order of first or smaller element.

2)Then perform modified LIS.

**Coin Change Problem:**

To count total number of solutions, we can divide solutions in two sets:

1. Solution that do not contain Sm
2. Solutions that contains at least One Sm

**Count(s[],m,n)=count(s[],m-1,n)+count(s[],m,n-Sm)**

Now, n is the value to be reached in this recursive solution.

m is the current coin index.

**Longest Common Substring:**

**Subset Sum Problem:**

Given a set of non negative integers ans value sum. Determine if there is a subset of the given set with sum equal to given number.

**Solution:**

**Two choices:**

Include the last element. Recur for n=n-1 sum=sum-set[n-1]

Exclude the last element. Recur for n=n-1  
  
subset sum problem is a different variant of coin change problem.

**Maximum Numbers Of Edits Require To Convert String 1 to String 2:  
  
Problem statement:**

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

Insert

Remove

Replace

**Solution:**

What are the sub-problems in this case?

The idea is process all characters one by one staring from either from left or right sides of both strings.

Let we traverse from right corner, there are two possibilities for every pair of character being traversed.

m: Length of str1 (first string)

n: Length of str2 (second string)

If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.

Else (If last characters are not same), we consider all operations on ‘str1’, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.

Insert: Recur for m and n-1

Remove: Recur for m-1 and n

Replace: Recur for m-1 and n-1

**Tiling Problem:**

**There are two tiling problems which are easy:  
  
First,** Given a “2 x n” board and tiles of size “2 x 1”, count the number of ways to tile the given board using the 2 x 1 tiles. A tile can either be placed horizontally i.e., as a 1 x 2 tile or vertically i.e., as 2 x 1 tile.

**Second,** Count number of ways to fill a “n x 4” grid using “1 x 4” tiles

I can do both.

**Space Optimization Solution of LCS:**

Track last two rows (I and I-1th) for each iteration in LCS.

**LCS of three strings:**

**Size of array after repeated deletion of LIS:**

You have to do LIS in every step.

**Maximum Bitonic Sequence:  
  
Problem statement:**

Given an array of integers. A subsequence of arr[] is called Bitonic if it is first increasing, then decreasing.

**Solution:**

We construct two arrays MSIBS[] and MSDBS[]. MSIBS[i] stores the sum of the Increasing subsequence ending with arr[i]. MSDBS[i] stores the sum of the Decreasing subsequence starting from arr[i]. Finally, we need to return the max sum of MSIBS[i] + MSDBS[i] – Arr[i].

**Count All Possible Paths From Top Left To Bottom Right of a m\*n matrix:**

**Condition:** From each cell you can either move only to right or down.

I can do it.

**Trick:** consider each cell as the rightmost bottom most cell during calculation.

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

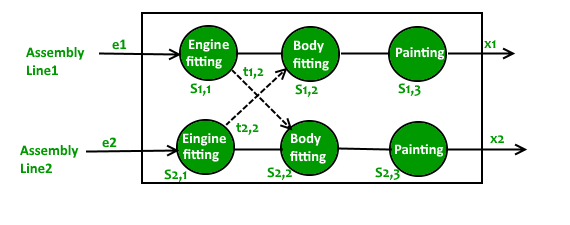
If indices are in proper range.

**Minimum jump to reach end:**

I can do it.

**Assembly Line Scheduling:**

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



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

Two assembly lines, 1 and 2, each with stations from 1 to n.

A car chassis must pass through all stations from 1 to n in order(in any of the two assembly lines). i.e. it cannot jump from station i to station j if they are not at one move distance.

The car chassis can move one station forward in the same line, or one station diagonally in the other line. It incurs an extra cost ti, j to move to station j from line i. No cost is incurred for movement in same line.

The time taken in station j on line i is ai, j.

Si, j represents a station j on line i.

Breaking the problem into smaller sub-problems:

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

If the minimum time taken by the chassis to leave station Si, j-1 is known, the minimum time taken to leave station Si, j can be calculated quickly by combining ai, j and ti, j.

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

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

**Base cases:**

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

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

T1(1) = Entry time in Line 1 + Time spent in station S1,1

T1(1) = e1 + a1,1

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

T2(1) = e2 + a2,1

**Recursive Relations:**

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

The car chassis at station S1,j can come either from station S1, j-1 or station S2, j-1.

Case #1: Its previous station is S1, j-1

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

T1(j) = Minimum time taken to leave station S1, j-1 + Time spent in station S1, j

T1(j) = T1(j-1) + a1, j

Case #2: Its previous station is S2, j-1

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

T1(j) = Minimum time taken to leave station S2, j-1 + Extra cost incurred to change the assembly line + Time spent in station S1, j

T1(j) = T2(j-1) + t2, j + a1, j

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

T1(j) = min((T1(j-1) + a1, j), (T2(j-1) + t2, j + a1, j))

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

T2(j) = min((T2(j-1) + a2, j), (T1(j-1) + t1, j + a2, j))

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

Tmin = min(Time taken to leave station Si,n + Time taken to exit the car factory)

Tmin = min(T1(n) + x1, T2(n) + x2)

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

**Problem statement:**

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.

I can do that.

**Maximum Path Sum that starts with any cell of the 0th row and ends with any cell of the (n-1)th row.**Given a N X N matrix Mat[N][N] of positive integers. There are only three possible moves from a cell (i, j)

(i+1, j)

(i+1, j-1)

(i+1, j+1)  
  
I can do it.  
  
**Count Even Length Binary Sequences with same sum of first and second half bits.**

Now, let’s say the length of the first half is n. So, the whole string is 2\*n.

The idea is to first and last bits of first half and second half respectively, and reduce the length of a half as (n-1). there are three possibilities when we fix first and last bits:

1. First and last bits are the same, remaining (n-1) bits on both sides should also have the same sum.
2. First bit is 1 and last bit is 0. Sum of remaining n-1 bits on the left side should be 1 less than the sum of n-1 bits on the right side.
3. First bit is 0 and last bit is 1. Sum of remaining n-1 bits on the left side should be 1 more than the sum of n-1 bits on the right side.  
     
   **Count(n,diff)=2\*count(n-1,diff)+count(n-1,diff-1)+count(n-1,diff+1)**

**Base cases:**if n=1 (I.e. total string length is 2\*1. and diff 0, return 2(0 0 or 1 1)

If n=1 and diff is 1, return 1

If n=1 and diff is -1, return 1

If [diff]>n, return 0.

**Nth Catalan Number:**

Co=1



**Cutting A Rod:**

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces.

**cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}**

**How To Reconstruct The Solution:**

Now, let us write the pseudo code of the original solution:

**Input:** Price array (price[0…n]), n (the number of elements)

Let, r[0…n] be a new array. Initialize r[i] for all 

for all i=0 to n

result\_for\_curr\_rod\_size=INT\_MIN;

for i=1 to j

result\_for\_curr\_rod\_size=max(result\_for\_curr\_rod\_size,price[i]+r[j-i]);

r[j]=result\_for\_curr\_rod\_size;

return r[n]

Now, if we introduce another array s[0..n] which contains the cut position for every rod size

**Input:** Price array (price[0…n]), n (the number of elements)

Let, r[0…n] and s[0..n] be new arrays. Initialize r[i] for all 

for all i=0 to n

result\_for\_curr\_rod\_size=INT\_MIN;

for i=1 to j

If(result\_for\_curr\_rod\_size<price[i]+r[j-i])

{

result\_for\_curr\_rod\_size=price[i]+r[j-i];

s[j]=i;

}

r[j]=result\_for\_curr\_rod\_size;

return r[n]

Now, using s array you can always do that.

**Matrix-chain multiplication:**

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ....

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

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

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

// See the Cormen book for details of the following algorithm

#include<stdio.h>

#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

int MatrixChainOrder(int p[], int n)

{

/\* For simplicity of the program, one extra row and one

extra column are allocated in m[][]. 0th row and 0th

column of m[][] are not used \*/

int m[n][n];

int i, j, k, L, q;

/\* m[i,j] = Minimum number of scalar multiplications needed

to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where

dimension of A[i] is p[i-1] x p[i] \*/

// cost is zero when multiplying one matrix.

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

m[i][i] = 0;

// L is chain length.

for (L=2; L<n; L++)

{

for (i=1; i<n-L+1; i++)

{

//i is the starting matrix index in matrix chain

j = i+L-1;

//j is the ending index based on length

m[i][j] = INT\_MAX;

for (k=i; k<=j-1; k++)

{

// q = cost/scalar multiplications

q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];

if (q < m[i][j])

m[i][j] = q;

}

}

}

return m[1][n-1];

}

int main()

{

int arr[] = {1, 2, 3, 4};

int size = sizeof(arr)/sizeof(arr[0]);

printf("Minimum number of multiplications is %d ",

MatrixChainOrder(arr, size));

return 0;

}

**Find number of solutions of a linear equation of n variables:**

We can solve this problem recursively. The idea is to subtract first coefficient from rhs and then recur for remaining value of rhs.

If rhs = 0 countSol(coeff, 0,n-1,rhs) = 1

Where 0 is the starting indexing index. n-1 is the ending index.

**countSol(coeff,0,n-1,rhs)**=where coeff[I]<=rhs and I varies from 0 to n-1.

**Solution:**

// A Dynamic programming based C++ program to find number of

// non-negative solutions for a given linear equation

#include<bits/stdc++.h>

using namespace std;

// Returns counr of solutions for given rhs and coefficients

// coeff[0..n-1]

int countSol(int coeff[], int n, int rhs)

{

// Create and initialize a table to store results of

// subproblems

int dp[rhs+1];

memset(dp, 0, sizeof(dp));

dp[0] = 1;

// Fill table in bottom up manner

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

for (int j=coeff[i]; j<=rhs; j++)

//because, we can subtract any value between 0 to coeff[i]

dp[j] += dp[j-coeff[i]];

return dp[rhs];

}

// Driver program

int main()

{

int coeff[] = {2, 2, 5};

int rhs = 4;

int n = sizeof(coeff)/sizeof(coeff[0]);

cout << countSol(coeff, n, rhs);

return 0;

}

**Find maximum length Snake sequence:  
  
problem statement:**

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

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

**Solution:**

I can do it.

**Choice of Area:**

**Problem statement:**

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.

I can do it. Observing all the constraints closely I can make a solution.

**Minimum cost to fill given weight in a bag**

**Problem statement:**

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 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 infinite supply of all available packet types.

**Solution:**

This problem is can be reduced to 0-1 Knapsack Problem. So in cost array, we first ignore those packets which are not available i.e; cost is -1 and then traverse the cost array and create two array val[] for storing cost of ‘i’ kg packet of orange and wt[] for storing weight of corresponding packet. Suppose cost[i] = 50 so weight of packet will be i and cost will be 50.

Algorithm :

Create matrix min\_cost[n+1][W+1], where n is number of distinct weighted packets of orange and W is maximum capacity of bag.

Initialize 0th row with INF (infinity) and 0th Column with 0.

Now fill the matrix

if wt[i-1] > j then min\_cost[i][j] = min\_cost[i-1][j] ;

if wt[i-1] >= j then min\_cost[i][j] = min(min\_cost[i-1][j], val[i-1] + min\_cost[i][j-wt[i-1]]);

**If min\_cost[n][W]==INF then output will be -1 because this means that we cant not make make weight W by using these weights else output will be min\_cost[n][W].**

Now, that’s why we set min\_cost[i][0]=INF for all i=1 to n

It’s just a marker.

Now, min\_cost[0][j] for all j=1 to w has a different meaning. INF marker also separates some cases from others.

**Maximum weight path ending at any element of last row in a matrix:**

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).

Down Move : (i+1, j)

Diagonal Move : (i+1, j+1)

I can do that.

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

I can do that.

**Maximum subsequence sum such that no three are consecutive**

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

I can do that.

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

I can do that.

**Path with maximum average value:**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 top left cell move only right or down and ends on bottom right cell. We want to find a path with maximum average over all existing paths. Average is computed as total cost divided by number of cells visited in path.

**Solution:**

This problem is quite easy. If we know this trick.

One interesting observation is, the only allowed moves are down and right, we need N-1 down moves and N-1 right moves to reach destination (bottom rightmost). So any path from from top left corner to bottom right corner requires 2N – 1 cells. In average value, denominator is fixed and we need to just maximize numerator. Therefore we basically need to to find maximum sum path.

**Friends Pairing Problem**

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.

**Solution:**

Let f(n) be the number of ways people can remain single or pair up.

For nth person, there are two choices:  
  
i) nth person remains single, We recur for f(n-1)

ii) nth person pairs up. Now, he can pair up with any of the (n-1) persons remaining. And the rest of the n-2 people can either remain single or pair in f(n-2) ways.

Hence, f(n)=f(n-1)+(n-1)\*f(n-2)  
  
**Maximum sum of disjoint pairs with specific difference**

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.

First we sort the given array in increasing order. Once array is sorted, we traverse the array. For every element, we try to pair it with its previous element first. Why do we prefer previous element? Let arr[i] can be paired with arr[i-1] and arr[i-2] (i.e. arr[i] – arr[i-1] < K and arr[i]-arr[i-2] < K). Since the array is sorted, value of arr[i-1] would be more than arr[i-2]. Also, we need to pair with difference less than k, it means if arr[i-2] can be paired, then arr[i-1] can also be paired in a sorted array.

Now observing the above facts, we can formulate our dynamic programming solution as below,

Let dp[i] denotes the maximum disjoint pair sum we can achieve using first i elements of the array. Assume currently we are at i’th position, then there are two possibilities for us.

Pair up i with (i-1)th element, i.e.

dp[i] = dp[i-2] + arr[i] + arr[i-1]

Don't pair up, i.e.

dp[i] = dp[i-1]

**Minimum steps to minimize n as per given condition:**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.

**Gold Mine Problem**

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.

I can do it.

**Maximum path sum in a triangle.**

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.

I can do it.

**Find number of endless points**

Given a binary N x N matrix, we need to find the total number of matrix positions from which there is an endless path. Any position (i, j) is said to have an endless path if and only if all of the next positions in its row(i) and its column(j) should have value 1. If any position next to (i,j) either in row(i) or in column(j) will have 0 then position (i,j) doesn’t have any endless path.

**Advance Approach (Dynamic programming):**

We can easily say that if there is a zero at any position, then it will block path for all the positions left to it and top of it.

Also, we can say that any position (i,j) will have an endless row if (i,j+1) will have an endless row and value of (i,j) is 1.

Similarly, we can say that any position (i,j) will have an endless column if (i+1,j) will have an endless column and value of (i,j) is 1.

**Subset with sum divisible by m**

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

This problem is a variant of subset sum problem. In subset sum problem we check if given sum subset exist or not, here we need to find if there exist some subset with sum divisible by m or not. Seeing input constraint, it looks like typical DP solution will work in O(nm) time. But in tight time limits in competitive programming, the solution may work. Also auxiliary space is high for DP table, but here is catch.

**If n > m there will always be a subset with sum divisible by m (which is easy to prove with pigeonhole principle). So we need to handle only cases of n <= m .**

For n <= m we create a boolean DP table which will store the status of each value from 0 to m-1 which are possible subset sum (modulo m) which have been encountered so far.

Now we loop through each element of given array arr[], and we add (modulo m) j which have DP[j] = true and store all the such (j+arr[i])%m possible subset sum in a boolean array temp, and at the end of iteration over j, we update DP table with temp. Also we add arr[i] to DP I.e. DP[arr[i]%m] = true.

In the end if DP[0] is true then it means YES there exist a subset with sum which is divisible by m, else NO.

// C++ program to check if there is a subset

// with sum divisible by m.

#include <bits/stdc++.h>

using namespace std;

// Returns true if there is a subset

// of arr[] with sum divisible by m

bool modularSum(int arr[], int n, int m)

{

if (n > m)

return true;

// This array will keep track of all

// the possible sum (after modulo m)

// which can be made using subsets of arr[]

// initialising boolean array with all false

bool DP[m];

memset(DP, false, m);

// we'll loop through all the elements of arr[]

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

{

// anytime we encounter a sum divisible

// by m, we are done

if (DP[0])

return true;

// To store all the new encountered sum (after

// modulo). It is used to make sure that arr[i]

// is added only to those entries for which DP[j]

// was true before current iteration.

bool temp[m];

memset(temp,false,m);

// For each element of arr[], we loop through

// all elements of DP table from 1 to m and

// we add current element i. e., arr[i] to

// all those elements which are true in DP

// table

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

{

// if an element is true in DP table

if (DP[j] == true)

{

if (DP[(j+arr[i]) % m] == false)

// We update it in temp and update

// to DP once loop of j is over

temp[(j+arr[i]) % m] = true;

}

}

// Updating all the elements of temp

// to DP table since iteration over

// j is over

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

if (temp[j])

DP[j] = true;

// Also since arr[i] is a single element

// subset, arr[i]%m is one of the possible

// sum

DP[arr[i]%m] = true;

}

return DP[0];

}

Now, check this algorithm. It turns O(n2) into O(nm)

**Now, how to prove that if there is n elements in an array and n>m there is a subset (not subarray) whose sum will be divisible by m?**

Read the small proof:  
  
How can I show that for any set of 5 integers, there is at least one subset of 3 integers whose sum is divisible by 3?

One way to approach this is as follows: Let  be given. Note that for every , there exists some q and some  such that .

Now let  and  be given such that for all () What we're trying to prove is that for every possible realization of this set of five variables we can find at least one combination of  such that a,b,c are all distinct and in {1,2,3,4,5,6} and that v6 is divisible by three.

Assume that there exist a combination of three vi ( )such that all three vi are equal. Without loss of generality we can assume these are . Then we have that Where me make use of the assumption that v1=v2=v3. Clearly the sum q1+q2+q3 is divisible by three.

Now assume there exists no combination of three vi  such that all three are equal. It is trivial to show that now we have at least one vi for every possible realization of vi. Thus, without loss of generality we can assume v1=0, v2=1, v3=2. We now obtain:

q1+q2+q3=3(x1+x2+x3)+v1+v2+v3=3(x1+x2+x3)+3=3(x1+x2+x3+1).

Which is clearly divisible by three. This completes the proof.

Note that this proof is only applicable for natural numbers. But the extension to integers shouldn't cause too much trouble.

**Another way of proving that:**Suppose the numbers are x1,x2,x3,x4, and x5. For each i, let ri be the unique number ri∈{0,1,2} such that ri≡xi(mod3). Then we need to show that every possible instance of the list r1,r2,r3,r4,r5 contains three numbers whose sum is congruent to 0 modulo 3.

The first thing to notice is that all three numbers 0,1, and 2 cannot occur in the list at the same time, since 0+1+2≡0(mod3)

So you must have a list of 5 numbers chosen from one of the sets {0,1},{0,2}, or {1,2}. By the pigeon-hole principle one of those numbers must occur at least three times. Since 0+0+0≡1+1+1≡2+2+2≡0(mod3), then there will always exist a subset of three numbers whose sum is a multiple of three.

Now, we can also prove when a list of 5 numbers are chosen from one of the sets {0},{1},{2} we can find one subset (it will be much more easier)

**Bell Numbers (Number of ways to Partition a Set)**

Given a set of n elements, find number of ways of partitioning it.

I**nput:** n = 2

**Output:** Number of ways = 2

**Explanation:** Let the set be {1, 2}

{ {1}, {2} }

{ {1, 2} }

**Input:** n = 3

**Output:** Number of ways = 5

**Explanation:** Let the set be {1, 2, 3}

{ {1}, {2}, {3} }

{ {1}, {2, 3} }

{ {2}, {1, 3} }

{ {3}, {1, 2} }

{ {1, 2, 3} }.

Now, let’s consider the second example:

n = 3

**Output:** Number of ways = 5

**Explanation:** Let the set be {1, 2, 3}

{ {1}, {2}, {3} }

{ {1}, {2, 3} }

{ {2}, {1, 3} }

{ {3}, {1, 2} }

{ {1, 2, 3} }.

Among them, these are the ways a group of 3 elements is partitioned into 2 sets (3)

{ {1}, {2, 3} }

{ {2}, {1, 3} }

{ {3}, {1, 2} }

Among them, this is the ways a group of 3 elements is partitioned into 1 set:

{1,2,3}

Among them, this is the ways a group of 3 elements is partitioned into 3 set:

{ {1}, {2}, {3} }

**Now, what is Bell number?**

Now, let S(n,k) be the number of ways in which a group of n elements are partitioned into k sets.

Now, bell number is 

Now, S(n+1, k) = k\*S(n, k) + S(n, k-1)

ow does above recursive formula work?

When we add a (n+1)th element to k partitions, there are two possibilities.

1) It is added as a single element set to existing partitions, i.e, S(n, k-1)

2) It is added to all sets of every partition, i.e., k\*S(n, k)

A better method is Bell’s triangle:

// If this is first column of current row 'i'

If j == 0

// Then copy last entry of previous row

// Note that i'th row has i entries

Bell(i, j) = Bell(i-1, i-1)

// If this is not first column of current row

Else

// Then this element is sum of previous element

// in current row and the element just above the

// previous element

Bell(i, j) = Bell(i-1, j-1) + Bell(i, j-1)

**Base case:**

Bell[0][0]=1

**A Sample Bell Triangle:**

1

1 2

2 3 5

5 7 10 15

15 20 27 37 52

**Subset Sum Problems Print All Given Set:**

**Solution:**// C++ program to count all subsets with

// given sum.

#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.**

**//this two can be in any order**

**//because, both can lead to different solution**

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;

}

**0-1 knapsack Problem:**

I can do that.

**Length Of The Longest Substring without repeating characters:**I can do that.

Map and dp.

**Longest Bitonic Sequence:  
  
Partition Problem:  
  
Remember the formula:**

S(n+1,k)=k\*s(n,k)+s(n,k-1)

And, how it is generated:

**Box stacking Problem:**You are given a set of n types of rectangular 3-D boxes, where the ith box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.

2) Sort the above generated 3n boxes in decreasing order of base area.

3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.

MSH(i) = Maximum possible Stack Height with box i at top of stack

MSH(i) = { Max ( MSH(j) ) + height(i) } where j < i and width(j) > width(i) and depth(j) > depth(i).

If there is no such j then MSH(i) = height(i)

1. To get overall maximum height, we return max(MSH(i)) where 0 < i < n.

**Egg Dropping Puzzle:**

The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

…..An egg that survives a fall can be used again.  
…..A broken egg must be discarded.  
…..The effect of a fall is the same for all eggs.  
…..If an egg breaks when dropped, then it would break if dropped from a higher floor.  
…..If an egg survives a fall then it would survive a shorter fall.  
…..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. **Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?**

The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

**Solution:**

k=number of floors.

n=number of eggs.  
eggDrop(n,k) or eggDrop[n][k]: Minimum no of trials needed to find the critical floor in this worst case.

eggDrop(n,k)=1+{min(max(eggDrop(n-1,x-1), eggDrop(n,k-x)) x in 1,2,…k}

egggDrop[n][k]=1+min(max(eggDrop[n-1][x-1], eggDrop[n][k-x]) for x in 1,2,…k)

eggDrop(n-1,x-1): if the egg breaks while the egg drops from xth floor, then we strictly need to check for lower floors with remaining eggs.

eggDrop(n,k-x): If an egg is dropped from xth floor and is not broken, we need to test it from remaining k-x floors.

**Base cases:**

eggDrop[i][0]=1 Independent of all I.

eggDrop[i][1]=1 (I.e. minimum number of trials to know whether 1st floor is safe for egg dropping is 1 and irrespective of i ()

**Bellman–Ford Algorithm:**

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed Dijkstra’s algorithm for this problem. Dijksra’s algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap).

**Advantages:**

Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs. (if their is a negative edge, Bellman Ford can detect it. It wont calculate the shortest path in that case) Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems.

**Disadvantages:**But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

**Input:** Graph and a source vertex src

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

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

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

…..a) Do following for each edge u-v

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

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

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

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

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

// A C++ program for Bellman-Ford's single source

// shortest path algorithm.

#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph

struct Edge

{

int src, dest, weight;

};

// a structure to represent a connected, directed and

// weighted graph

struct Graph

{

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

int V, E;

// graph is represented as an array of edges.

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[E];

return graph;

}

// A utility function used to print the solution

void printArr(int dist[], int n)

{

printf("Vertex Distance from Source\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

// The main function that finds shortest distances from src to

// all other vertices using Bellman-Ford algorithm. The function

// also detects negative weight cycle

void BellmanFord(struct Graph\* graph, int src)

{

int V = graph->V;

int E = graph->E;

int dist[V];

// Step 1: Initialize distances from src to all other vertices

// as INFINITE

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

dist[i] = INT\_MAX;

dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times. A simple shortest

// path from src to any other vertex can have at-most |V| - 1

// edges

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

{

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

{

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

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

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

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

}

}

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

// guarantees shortest distances if graph doesn't contain

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

// is a cycle.

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

{

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

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

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

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

printf("Graph contains negative weight cycle");

}

printArr(dist, V);

return;

}

// Driver program to test above functions

int main()

{

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

BellmanFord(graph, 0);

return 0;

}

Now, note the following things:

**Calculate shortest path from source to other vertices take O(VE)**

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

{

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

{

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

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

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

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

}

}

**Now, detecting the presence of negative edge weight cycle takes O( E )**for (int i = 0; i < E; i++)

{

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

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

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

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

printf("Graph contains negative weight cycle");

}

**Floyd Warshall Algorithm**The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

**Largest Independent Set Problem:**Given a Binary Tree, find size of the Largest Independent Set(LIS) in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.

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

(sum of LISS for all children of X) }

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

#include <stdio.h>

#include <stdlib.h>

// A utility function to find max of two integers

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

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

right child \*/

struct node

{

int data;

int liss;

struct node \*left, \*right;

};

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

// a given binary tree

int LISS(struct node \*root)

{

if (root == NULL)

return 0;

if (root->liss)

return root->liss;

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

return (root->liss = 1);

// Calculate size excluding the current node

int liss\_excl = LISS(root->left) + LISS(root->right);

// Calculate size including the current node

int liss\_incl = 1;

if (root->left)

liss\_incl += LISS(root->left->left) + LISS(root->left->right);

if (root->right)

liss\_incl += LISS(root->right->left) + LISS(root->right->right);

// Maximum of two sizes is LISS, store it for future uses.

root->liss = max(liss\_incl, liss\_excl);

return root->liss;

}

// A utility function to create a node

struct node\* newNode(int data)

{

struct node\* temp = (struct node \*) malloc( sizeof(struct node) );

temp->data = data;

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

temp->liss = 0;

return temp;

}

// Driver program to test above functions

int main()

{

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

struct node \*root = newNode(20);

root->left = newNode(8);

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

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

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

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

root->right = newNode(22);

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

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

return 0;

}

**Note that:** an additional field is maintained to ease the dynamic programming calculation.

**Minimum insertions to form a palindrome:**

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.

Before we go further, let us understand with few examples:

ab: Number of insertions required is 1. bab

aa: Number of insertions required is 0. aa

abcd: Number of insertions required is 3. dcbabcd

abcda: Number of insertions required is 2. adcbcda which is same as number of insertions in the substring bcd(Why?).

abcde: Number of insertions required is 4. edcbabcde

Let the input string be str[l……h]. The problem can be broken down into three parts:

1. Find the minimum number of insertions in the substring str[l+1,…….h].

2. Find the minimum number of insertions in the substring str[l…….h-1].

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

**Recursive Solution**

The minimum number of insertions in the string str[l…..h] can be given as:

minInsertions(str[l+1…..h-1]) if str[l] is equal to str[h]

min(minInsertions(str[l…..h-1]), minInsertions(str[l+1…..h])) + 1 otherwise

**Minimum number of deletions to make a string palindrome**

// C++ implementation to find minimum number

// of deletions to make a string palindromic

#include <bits/stdc++.h>

using namespace std;

// Returns the length of the longest

// palindromic subsequence in 'str'

int lps(string str)

{

int n = str.size();

// Create a table to store

// results of subproblems

int L[n][n];

// Strings of length 1

// are palindrome of length 1

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

L[i][i] = 1;

// Build the table. Note that the

// lower diagonal values of table

// are useless and not filled in

// the process. c1 is length of substring

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

{

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

{

int j = i + cl - 1;

**//j is the end character of currently considering substring**

if (str[i] == str[j] && cl == 2)

L[i][j] = 2;

**//two characters are correctly placed**

else if (str[i] == str[j])

L[i][j] = L[i + 1][j - 1] + 2;

else

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

}

}

// length of longest palindromic subseq

return L[0][n-1];

**//we are doing it in such a way because, we are finally going to return n-len or**

**//n-L[0][n-1]**

}

// function to calculate minimum

// number of deletions

int minimumNumberOfDeletions(string str)

{

int n = str.size();

// Find longest palindromic subsequence

int len = lps(str);

// After removing characters other than

// the lps, we get palindrome.

return (n - len);

**//check the final result it’s n-len Not, n**

}

// Driver Code

int main()

{

string str = "geeksforgeeks";

cout << "Minimum number of deletions = "

<< minimumNumberOfDeletions(str);

return 0;

}