9 Dynamic Programming

"Those who cannot remember the past are condemned to repeat it."

Dynamic Programing is all about remembering answers to the sub-problems you've already solved and not solving it again.

Q. Where do we need Dynamic Programming?

- If you are given a problem, which can be broken down into **smaller sub-problems**.
- These smaller sub-problems can still be broken into smaller ones - and if you manage to find out that there are some **over- lappping sub-problems**.
- Then you've encountered a DP problem.

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results.

Example

Writes down "1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 =" on a sheet of paper.

"What's that equal to?"

Counting "Eight!"

Writes down another "1+" on the left.

"What about that?"

"Nine!" " How'd you know it was nine so fast?"

"You just added one more!"

"So you didn't need to recount because you remembered the re were eight! Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Dynamic Programming and Recursion:

- Dynamic programming is basically, recursion plus memoization.
- Recursion allows you to express the value of a function in terms of other values of that function.
- If you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster.
- This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of **Fibonacci numbers**.

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21, 35...

A code for it using pure recursion:

```
int fib (int n) {
if (n < 2)
return 1;
return fib(n-1) + fib(n-2);
}
```

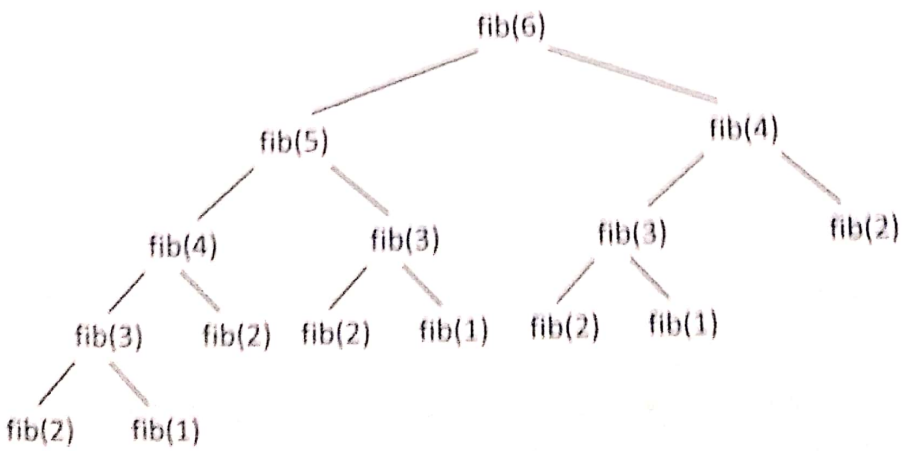
Using Dynamic Programming approach with memoization:

```
void fib () {
fib[0] = 1;
fib[1] = 1;
for (int i = 2; i<n; i++)
fib[i] = fib[i-1] + fib[i-2];
}
```

Q. Are we using a different recurrence relation in the two codes? **No**

Q. Are we doing anything different in the two codes? **Yes**

Let's Visualize :



Here we are running fibonacci function multiple times for the same value of n, which can be prevented using memoization.

Optimization Problems :

Dynamic Programming is typically applied to optimization problems. In such problems there can b any possible solutions. Each solution has a value, and we wish to find a solution with the optima (minimum or maximum) value. We call such a solution an optimal solution to the problem.

❑ MINIMUM STEPS TO ONE

http://www.spoj.com/problems/MST1/

On a positive integer, you can perform any one of the following 3 steps.

1. Subtract 1 from it.
2. If its divisible by 2, divide by 2.
3. If its divisible by 3, divide by 3. Now the question is, given a positive integer n, find the minimum number of steps that takes n to 1.

Cannot apply **Greedy**! (Why ?)

Recursive Solution :

We will define the recurrence relation as

solve(n):

if n = 1

ans = 0

if n > 1

ans = min{1 + solve(n-1), 1 + solve(n/2), 1 + solve(n/3)}

Code :

```
int minStep(int n){
if(n == 1)
return 0;
int subOne = INF, divTwo = INF, divThree = INF;
//If number is greater than or equal to 2, subtract one
if(n >= 2)
subOne = 1 + minStep(n-1);
//If divsible by 2, divide by 2
if(n % 2 == 0)
divTwo = 1 + minStep(n/2);
//If divisible by 3, divide by 3
if(n%3 == 0)
divThree = 1 + minStep(n/3);
//Return the most optimal answer
return min(subOne, min(divTwo, divThree));}
```

Since we are solving a single sub-problem multiple number of times, $T = O(K^N)$

Optimization Problems :

Dynamic Programming is typically applied to optimization problems. In such problems there can be any possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem.

❑ MINIMUM STEPS TO ONE

http://www.spoj.com/problems/MST1/

On a positive integer, you can perform any one of the following 3 steps.

1. Subtract 1 from it.
2. If its divisible by 2, divide by 2.
3. If its divisible by 3, divide by 3. Now the question is, given a positive integer n, find the minimum number of steps that takes n to 1.

Cannot apply **Greedy**! (Why ?)

Recursive Solution :

We will define the recurrence relation as

solve(n):

if n = 1

ans = 0

if n > 1

ans = min{1 + solve(n-1), 1 + solve(n/2), 1 + solve(n/3)}

Code :

```
int minStep(int n){
if(n == 1)
return 0;
int subOne = INF, divTwo = INF, divThree = INF;
//If number is greater than or equal to 2, subtract one
if(n >= 2)
subOne = 1 + minStep(n-1);
//If divsible by 2, divide by 2
if(n % 2 == 0)
divTwo = 1 + minStep(n/2);
//If divisible by 3, divide by 3
if(n%3 == 0)
divThree = 1 + minStep(n/3);
//Return the most optimal answer
return min(subOne, min(divTwo, divThree));}
```

Since we are solving a single sub-problem multiple number of times, $T = O(K^N)$

Can we do better?

The recursion tree for the above solution results in subproblem overlapping. So, we can improve the solution using memoization.

Dynamic Programming Solution :

1. **Top Down DP :** In Top Down, you start building the big solution right away by explaining how you build it from smaller solutions.

Example :

I will be an amazing coder. How?

I will work hard like crazy. How?

I'll practice more and try to improve. How?

I'll start taking part in contests. How?

I'll practicing. How?

I'm going to learn programming.

Code :

```
int minStep(int n){
//if already calculated, return this answer
   if(dp[n] != -1)
         return dp[n];
// Base case
   if(n <= 1)
         return 0;
   int subOne = INF, divTwo = INF, divThree = INF;
//If number is greater than or equal to 2, subtract one
   if(n >= 2)
         subOne = 1 + minStep(n-1);
//If divsible by 2, divide by 2
   if(n % 2 == 0)
         divTwo = 1 + minStep(n/2);
//If divisible by 3, divide by 3
   if(n%3 == 0)
         divThree = 1 + minStep(n/3);
//Return the most optimal answer
   return dp[n] = min(subOne, min(divTwo, divThree));
}
```

2. **Bottom Up DP** : In Bottom Up, you start with the small solutions and then use these small solutions to build up larger solutions.

Example :

I'm going to learn programming.
Then, I will start practicing.
Then, I will start taking part in contests.
Then, I'll practice even more and try to improve.
After working hard like crazy,
I'll be an amazing coder.

Code :

```
void minStep(){
//Base case
dp[1] = 0;
dp[0] = 0;
//Iterate for all possible numbers starting from 2
for(int i = 2;i<=2 * 10000000;i++){
dp[i] = 1 + dp[i-1];
if(i % 2 == 0)
dp[i] = min(dp[i],1 + dp[i/2]);
if(i % 3 == 0)
dp[i] = min(dp[i],1 + dp[i/3]);
}
return;
}
```

❑ **MINIMUM COIN CHANGE**

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

Example :

Input: coins[] = {25, 10, 5}, N = 30
Output: Minimum 2 coins required
We can use one coin of 25 cents and one of 5 cents
Input: coins[] = {9, 6, 5, 1}, N = 13
Output: Minimum 3 coins required
We can use one coin of 6 + 6 + 1 cents coins.

Recursive Solution :

Start the solution with initially sum = N cents and at each iteration find the minimum coins required by dividing the problem in subproblems where we take {C1, C2, ..., CM} coin and decrease the sum N by C[i] (depending on the coin we took). Whenever N becomes 0, this means we have a possible solution. To find the optimal answer, we return the minimum of all answer for which N became 0.

If N == 0, then 0 coins required.

If N > 0

minCoins(N , coins[0..m-1]) = min {1 + minCoins(N-coin[i] ,coins[0....m-1])}

where i varies from 0 to m-1 and coin[i] <= N

Code :

```
int minCoins(int coins[], int m, int N)
{
    // base case
     if (N == 0)
       return 0;
    // Initialize result
    int res = INT_MAX;
    // Try every coin that has smaller value than V
    for (int i=0; i<m; i++)
    {
      if (coins[i] <= N)
        {
           int sub_res = 1 + minCoins(coins, m, N-coins[i]);
            // see if result can minimized
           if (sub_res < res)
               res = sub_res;
        }
    }
  return res;
}
```

Dynamic Programming Solution :

Since same suproblems are called again and again, this problem has Overlapping Subprolems property. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array dp[] and memoizing the computed values in this array.

1. **Top Down DP**

Code :

```
int minCoins(int N, int M)
{
    // if we have already solved this subproblem
    // return memoized result
    if(dp[N] != -1)
     return dp[N];
    // base case
    if (N == 0)
      return 0;
    // Initialize result
    int res = INF;
    // Try every coin that has smaller value than N
     for (int i=0; i<M; i++)
      {
        if (coins[i] <= N)
          {
             int sub_res = 1 + minCoins(N-coins[i], M);
             // see if result can minimized
             if (sub_res < res)
                res = sub_res;
          }
      }
   return dp[N] = res;
}
```

2. Bottom Up DP : ith state of dp : dp[i] : Minimum number of coins required to sum to i cents.

Code :

```
int minCoins(int N, int M)
{
  //Initializing all values to infinity i.e. minimum coins to make any
  //amount of sum is infinite
     for(int i = 0;i<=N;i++)
        dp[i] = INF;
 //Base case i.e. minimum coins to make sum = 0 cents is 0
     dp[0] = 0;
  //Iterating in the outer loop for possible values of sum between 1 to N
  //Since our final solution for sum = N might depend upon any of these
values
    for(int i = 1;i<=N;i++)
    {
        //Inner loop denotes the index of coin array.
        //For each value of sum, to obtain the optimal solution.
       for(int j = 0;j<M;j++)
        {
         //i -> sum
         //j -> next coin index
         //If we can include this coin in our solution
          if(coins[j] <= i)
           {
             //Solution might include the newly included coin.
             dp[i] = min(dp[i], 1 + dp[i - coins[j]]);
           }
        }
   }
 //for(int i = 1;i<=N;i++)cout<<i<<" "<<dp[i]<<endl;
 return dp[N];
}
```

T = O(N*M)

Solution Link: http://pastebin.com/JFNh3LN3

WINE AND MAXIMUM PRICE

Imagine you have a collection of N wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from 1 to N, respectively. The price of the ith wine is pi. (prices of different wines can be different).

Because the wines get better every year, supposing today is the year 1, on year ythe price of the ith wine will be y*pi, i.e. y-times the value that current year.

You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the she lf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).

You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order?

So, for example, if the prices of the wines are (in the order as they are

placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3. The optimal solution would be to sell the wines in the order p1, p4, p3, p2 for a total profit 1 * 1 + 3 * 2 + 2 * 3 + 4 * 4 = 29.

Wrong Solution!

Greedy Approach :

Every year, sell the cheaper of the two(leftmost and right most) available wines.

Let the prices of 4 wines are: 2, 3, 5, 1, 4

At t = 1 year: {2,3,5,1,4} sell p1 = 2 to get cost = 2

At t = 2 years: {3,5,1,4} sell p2 = 3 to get cost = 2 + 2*3 = 8

At t = 3 years: {5,1,4} sell p5 = 4 to get cost = 8 + 4*3 = 20

At t = 4 years: {5,1} sell p4 = 1 to get cost = 20 + 1*4 = 24

At t = 5 years: {5} sell p3 = 5 to get cost = 24 + 5*5 = 49

Greedy approach gives an optimal answer of 49, but if we sell in the order of p1, p5, p4, p2, p3 for a total profit 2 * 1 + 4 * 2 + 1 * 3 + 3 * 4 + 5 * 5 = 50, greedy fails.

Recursive Solution : Here we will try out all the possible options and output the optimal Answer.

```
if(start > end)
return 0
if(start <= end)
return maxPrice(price, start, end, year) = max{price[
start] * year + maxPrice(price, start+1, end, year + 1),
price[end] * year + maxPrice(price, start, end + 1,year + 1)},
```

For each endpoint at every function call, we are doing the following:

- Either we take this end point in the solution, increment/decrement the end point index.
- Or we do not take this end point in the solution.

Since we are doing this for each and every n endpoints (n is number of wines on the table). So, T = O(2^n) which is exponential time complexity.

Code :

```
int maxPrice(int price[], int start, int end, int year)
{
//Base case, stop when star of array becomes more than end.
 if(start > end)
   return 0;
 /Including the wine with starting index in our solution
 int incStart = price[start] * year + maxPrice(price,start + 1, end, year + 1);
 //Including the wine with ending index in our solution
 int incEnd = price[end] * year + maxPrice(price, start, end-1, year + 1);
 //return the most optimal answer
 return max(incStart, incEnd);
}
```

Can we do better?

Yes we can! By carefully observing the recursion tree, we can clearly see that we encounter the property of subproblem overlapping which can be prevented using memoization or dynamic programming.

Top Down DP Code :

```
int maxPrice(int start,int end,int N)
{
   //If we have solved this sub-problem, return the memoized answer
   if(dp[start][end] != 0)
      return dp[start][end];
   //base case
   if(start > end)
     return 0;
   //To get the current year
   int year = N - (end - start + 1) + 1;
   //Including the starting index
```

```
    int incStart = year * price[start] + maxPrice(start+1, end, N);
    //Including the ending index
    int incEnd = year * price[end] + maxPrice(start,end-1,N);
    //memoize this solution and return
    return dp[start][end] = max(incStart,incEnd);
}
```

Solution: http://pastebin.com/Yx1FG3ys

We can also solve this problem using the bottom up dynamic programming approach. Here we will need to create a 2-Dimensional array for memoization where the states i and j in **dp[i][j]** denotes the optimal answer between the **starting index i** and **ending index j**.

According to the definition of states, we define:

```
dp[i][j] = max{current_year * price[i] + dp[i+1][j], curr
ent_year * price[j] + dp[i][j+1]}
```

Bottom Up DP Code :

```
int maxPrice(int start,int end,int N){
 //Initialize the dp array
  for(int i = 0;i<N;i++)
  {
    for(int j = 0;j<N;j++)
      dp[i][j] = 0;
  }
 //Outer loop denotes the starting index for our solution
  for(int i = N-1;i>=0;i--)
  {
  //Inner loop denotes the ending index
    for(int j = 0;j<N;j++)
    {
     //if (start > end), return 0
      if(i > j)
         dp[i][j] = 0;
      else
      {
      //find the current year
         int year = N - (j - i);
```

```
            //using bottom up dp to solve the problem for smaller sub problems
            //and using it to solve the larger problem i.e. dp[i][j]
             dp[i][j] = max(year * price[i] + dp[i+1][j], year * price[j] +
dp[i][j-1]);
        }
    }
 }
 //return the final answer where starting index is start = 0 and ending index
is end = n-1
 return dp[0][N-1];
}
```

Solution: http://pastebin.com/idsyg4aw

T = O(N^2), where N is the number of wines.

❑ PROBLEMS INVOLVING GRIDS

- Finding a minimum-cost path in a grid
- Finding the number of ways to reach a particular position from a given starting point in a 2-D grid and so on.

Finding Minimum-Cost Path in a 2-D Matrix

Problem : Given a cost matrix Cost[][] where Cost[i][j] denotes the Cost of visiting cell with coordinates (i,j), find a min-cost path to reach a cell (x,y) from cell (0,0) under the condition that you can only travel one step right or one step down. (We assume that all costs are positive integers)

It is very easy to note that if you reach a position (i,j) in the grid, you must have come from one cell higher, i.e. (i-1,j) or from one cell to your left , i.e. (i,j-1). This means that the cost of visiting cell (i,j) will come from the following recurrence relation:

Code :

```
MinCost(i,j) = min{ MinCost(i-1,j), MinCost(i,j-1) } + cost[i][j]
```

We now compute the values of the base cases: the topmost row and the leftmost column. For the topmost row, a cell can be reached only from the cell on the left of it. Assuming zero-based index,

```
MinCost(0,j) = MinCost(0,j-1) + Cost[0][j]
MinCost(i,0) = MinCost(i-1,0) + Cost[i][0]
```

i.e. cost of reaching cell (0,j) = Cost of reaching cell (0,j-1) + Cost of visiting cell (0,j) Similarly, cost of reaching cell (i,0) = Cost of reaching cell (i-1,0) + Cost of visiting cell (i,0).

Code :

```
int MinCost(int Ro, int Col)
{
  //this bottom-up approach ensures that all the sub-problems needed
  // have already been calculated.
   for(int i = 0;i<Ro;i++)
    {
      for(int j = 0;j<Col;j++)
       {
         //Base Cases
         if(i == 0 && j == 0)
           dp[i][j] = cost[0][0];
         else if(i == 0)
           dp[i][j] = dp[0][j-1] + cost[0][j];
         else if(j == 0)
           dp[i][j] = dp[i-1][0] + cost[i][0];
  //Calculate cost of visiting (i,j) using the
  //recurrence relation discussed above
       else
           dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + cost[i][j];
       }
    }
 //Return the optimum cost of reaching the last cell
  return dp[Ro-1][Col-1];
}
```

- **Finding the number of ways to reach from a starting position to an ending position travelling in specified directions only.**

Problem Statement : Given a 2-D matrix with M rows and N columns, find the number of ways to reach cell with coordinates (i,j) from starting cell (0,0) under the condition that you can only travel one step right or one step down.

This problem is very similar to the previous one. To reach a cell (i,j), one must first reach either the cell (i-1,j) or the cell (i,j-1) and then move one step down or to the right respectively to reach cell (i,j). After convincing yourself that this problem indeed satisfies the optimal sub-structure and overlapping subproblems properties, we try to formulate a bottom-up dynamic programming solution.

Let **NumWays(i,j)** be the number of ways to reach position (i,j). As stated above, number of ways to reach cell (i,j) will be equal to the sum of number of ways of reaching (i-1,j) and number of ways of reaching (i,j-1). Thus, we have our recurrence relation as :

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

Code :

```
int numWays(int Ro, int Col)
{
    //This bottom-up approach ensures that all the sub-problems needed
    // have already been calculated.
   for(int i = 0;i<Ro;i++)
    {
      for(int j = 0;j<Col;j++)
       {
        //Base Cases
        if(i == 0 && j == 0)
          dp[i][j] = 1;
        else if(i == 0)
          dp[i][j] = 1;
        else if(j == 0)
          dp[i][j] = 1;
        //Calculate no. of ways to visit (i,j) using the
        //recurrence relation discussed above
        else
          dp[i][j] = dp[i-1][j] + dp[i][j-1];
       }
    }
     //Return the optimum cost of number of ways
    return dp[Ro-1][Col-1];
}
```

Ques. Finding the number of ways to reach a particular position in a grid from a starting position (given some cells which are blocked)

Problem Statement : Input is three integers M, N and P denoting the number of rows, number columns and number of blocked cells respectively. In the next P lines, each line has exactly 2 integers i and j denoting that the cell (i, j) is blocked.

https://www.codechef.com/problems/CDIIT4

The code below explains how to proceed with the solution. The problem is same as the previous one, except for few extra checks (due to blocked cells).

```
int numWays(int Ro, int Col)
{
 //if the initial block is blocked, we cannot move further
   if(dp[0][0] == -1)
   {
     return 0;
   }
//Number of ways for the first row
   for(int j = 0;j<Col;j++)
   {
     //If any cell is blocked in the first row, we can not visit any
     //cell that are to the right of this blcked cell
      if(dp[0][j] == -1)
        break;
     //There is only one way to reach this cell i.e. from left cell
      dp[0][j] = 1;
  }
 //Number of ways for first column
 for(int i = 0;i<Ro;i++)
{
  //Similar to first row
   if(dp[i][0] == -1)
     break;
  dp[i][0] = 1;
}
//This bottom-up approach ensures that all the sub-problems needed
//have already been calculated.
for(int i = 1;i<Ro;i++)
{
   for(int j = 1;j<Col;j++)
     {
        //If we encounter a blocked cell, do nothing
```

```
            if(dp[i][j] == -1)
               continue;
            //Calculate no. of ways to visit (i,j)
            dp[i][j] = 0;
            //If the cell on the left is not blocked, we can reach
            //(i,j) from (i,j-1)
             if(dp[i][j-1] != -1)
                dp[i][j] = dp[i][j-1] % MOD;
            //If the cell above is not blocked, we can reach
            //(i,j) from (i-1,j)
             if(dp[i-1][j] != -1)
                dp[i][j] = (dp[i][j] + dp[i-1][j]) % MOD;
        }
    }
    //If last cell is blocked, return 0
    if(dp[Ro-1][Col-1] == -1)
       return 0;
    //Return the optimum cost of number of ways
    return dp[Ro-1][Col-1];
}
```

Solution : http://pastebin.com/LeAhedeQ

Another Variant :

Problem Statement : You are given a 2-D matrix A of n rows and m columns where A[i][j] denotes the calories burnt. Two persons, a boy and a girl, start from two corners of this matrix. The boy starts from cell (1,1) and needs to reach cell (n,m). On the other hand, the girl starts from cell (n,1) and needs to reach (1,m). The boy can move right and down. The girl can move right and up. As they visit a cell the amount in the cell A[i][j] is added to their total of calories burnt. You have to maximize the sum of total calories burnt by both of them under the condition that they shall meet only in one cell and the cost of this cell shall not be included in either of their total.

http://codeforces.com/contest/429/problem/B

Let us analyse this problem in steps:

The boy can meet the girl in only one cell.

So, let us assume they meet at cell (i,j).

Boy can come in from left or the top, i.e. (i,j-1) or (i-1,j). Now he can move right or down. That is, the sequence for the boy can be:

(i,j-1)—>(i,j)—>(i,j+1)

(i,j-1)—>(i,j)—>(i+1,1)

(i-1,j)—>(i,j)—>(i,j+1)

(i-1,j)—>(i,j)—>(i+1,j)

Similarly, the girl can come in from the left or bottom, i.e. (i,j-1) or (i+1,j) and she can go up or right. The sequence for girl's movement can be:

(i,j-1)—>(i,j)—>(i,j+1)

(i,j-1)—>(i,j)—>(i-1,j)

(i+1,j)—>(i,j)—>(i,j+1)

(i+1,j)—>(i,j)—>(i-1,j)

Comparing the 4 sequences of the boy and the girl, the boy and girl meet only at one position (i,j). If

Boy: (i,j-1)—>(i,j)—>(i,j+1) and Girl: (i+1,j)—>(i,j)—>(i-1,j)

or

Boy: (i-1,j)—>(i,j)—>(i+1,j) and Girl: (i,j-1)—>(i,j)—>(i,j+1)

Now, we can solve the problem by creating 4 tables:

Boy's journey from start (1,1) to meeting cell (i,j)

Boy's journey from meeting cell (i,j) to end (n,m)

Girl's journey from start (n,1) to meeting cell (i,j)

Girl's journey from meeting cell (i,j) to end (1,n)

The meeting cell can range from 2<= i <= n-1 and 2 <= j <= m-1

See the code below for more details:

```
int maxCalories(int M, int N)
{
  //building boy_start[][] table in bottom up fashion
  //Here boy_start[i][j] -> the max calories that can be burnt if the boy
  //starts from (1,1) and goes up to (i,j)
  for(int i = 1;i<=M;i++)
   {
   for(int j = 1;j<=N;j++)
    {
    boy_start[i][j] = calorie[i][j] + max(boy_start[i-1][j], boy_start[i][j-1]);
    }
   }
//building girl_start[][] table in bottom up fashion
```

```
//Here girl_start[i][j] -> the max calories that can be burnt if the girl
//starts from (M,1) and goes up to (i,j)
 for(int i = M;i>=1;i--)
 {
 for(int j = 1;j<=N;j++)
 {
 girl_start[i][j] = calorie[i][j] + max(girl_start[i+1][j], girl_start[i][j-1]);
 }
 }
//building boy_end[][] table in bottom up fashion which specifies the journey from end
to start
//Here boy_end[i][j] -> the max calories that can be burnt if the boy start
//from end i.e. (M,N) and comes back to (i,j)
 for(int i = M;i>=1;i--)
 {
 for(int j = N;j>=1;j--)
 {
  boy_end[i][j] = calorie[i][j] + max(boy_end[i+1][j], boy_end[i][j+1]);
 }
 }
//building girl_end[][] table in bottom up fashion which specifies the journey from end
to start
//Here girl_end[i][j] -> the max calories that can be burnt if the girl start
//from end i.e. (1,N) and comes back to (i,j)
 for(int i = 1;i<=M;i++)
 {
 for(int j = N;j>=1;j--)
 {
 girl_end[i][j] = calorie[i][j] + max(girl_end[i-1][j], girl_end[i][j+1]);
 }
}
//Iterate over all the possible meeting points i.e. between (2,2) to (M-1,N-1)
//consider this point as the actual meeting point and calculate the max possible answer
int ans = 0;
for(int i = 2;i<M;i++)
{
for(int j = 2;j<N;j++)
{
int ans1 = boy_start[i][j-1]+boy_end[i][j+1]+girl_start[i+1][j] + girl_end[i-1][j]
```

```
    int ans2 = boy_start[i-1][j] + boy_end[i+1][j]+girl_start[i][j-1]+girl_end[i][j+1];
    ans = max(ans, max(ans1, ans2));
    }
   }
 return ans;
}
```

Solution: http://pastebin.com/cnBqeJEP

BUILDING BRIDGES

Problem Statement: Given two array of numbers which denotes the end points of bridges. What is the maximum number of bridges that can be built if ith point of first array must be connected to ith point of second array and two bridges cannot overlap each other.

http://www.spoj.com/problems/BRIDGE/

LONGEST INCREASING SUBSEQUENCE

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, 9, 3, 5, 4, 11, 7, 8} is 4 and LIS is {3, 4, 7, 8}.

Recurrence relation :

```
L(i) = 1 + max{ L(j) } where 0 < j < i and arr[j] < arr[i];
or
L(i) = 1, if no such j exists.
return max(L(i)) where 0 < i < n
```

Code :

```
int LIS(int n)
{
  int i,j,res = 0;
  /* Initialize LIS values for all indexes */
  for (i = 0; i < n; i++ )
   lis[i] = 1;
  /* Compute optimized LIS values in bottom up manner */
  for (i = 1; i < n; i++ )
   for (j = 0; j < i; j++ )
     if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
       lis[i] = lis[j] + 1;
```

```
    /* Pick maximum of all LIS values */
    for (i = 0; i < n; i++ )
     if (res < lis[i])
       res = lis[i];
   return res;
}
```

T = O(n^2)

We can also print the Longest Increasing Subsequence as:

```
void print_lis(int n)
{
//denotes the current LIS
//initially it is equal to the LIS of the whole sequence
 int cur_lis = LIS(n);
//denotes the previous element printed
//to print the LIS, previous element printed must always be larger than current
//element (if we are printing the LIS backwards)
//Initially set it to infinity
 int prev = INF;
 for(int i = n-1;i>=0;i--)
 {
  //find the element upto which the LIS equal to the cur_LIS
  //and that element is less than the previous one printed
  if(lis[i] == cur_lis && arr[i] <= prev)
  {
   cout<<arr[i]<<" ";
   cur_lis--;
   prev = arr[i];
  }
  if(!cur_lis)
   break;
 }
return;
}
```

In this problem we will use the concept of LIS. Two bridges will not cut each other if both their end points are either in non-increasing or non-decreasing order. To find the solution we will first pair the endpoints i.e. ith point in 1st sequence is paired with ith point in 2nd sequence. Then sort the points w.r.t 1st point in the pair and apply LIS on second point of the pair.

Voila! You have a solution :)

Example: First consider the pairs: (3,6), (5, 4), (8, 1), (10, 2), sort it according to the first element of the pairs (in this case are already sorted) and compute the lis on the second element of the pairs, thus compute the LIS on 6 4 1 2, that is 1 2. Therefore the non overlapping bridges we are looking for are (8, 1) and (10, 2)

Code :

```
int maxBridges(int n)
{
 //for a single point there is always a bridge
 if(n == 1)
  return 1;
 //Sort the points according to the first sequence
  sort(points.begin(), points.end());
 //Apply LIS on the second sequence
 int i,j,res = 0;
 //Initialize LIS values for all indexes
 for (i = 0; i < n; i++ )
  lis[i] = 1;
 //Compute optimized LIS values in bottom up manner
 for (i = 1; i < n; i++ )
  for (j = 0; j < i; j++ )
   if ( points[i].second >= points[j].second && lis[i] < lis[j] + 1)
     lis[i] = lis[j] + 1;
 //Pick maximum of all LIS values
  for (i = 0; i < n; i++ )
   if (res < lis[i])
     res = lis[i];
return res;
}
```

Solution:http://pastebin.com/UdiitgasV

In this problem we will use the concept of LIS. Two bridges will not cut each other if both their end points are either in non-increasing or non-decreasing order. To find the solution we will first pair the endpoints i.e. ith point in 1st sequence is paired with ith point in 2nd sequence. Then sort the points w.r.t 1st point in the pair and apply LIS on second point of the pair.

Voila! You have a solution :)

Example: First consider the pairs: (2,6), (5, 4), (8, 1), (10, 2), sort it according to the first element of the pairs (in this case are already sorted) and compute the lis on the second element of the pairs, thus compute the LIS on 6 4 1 2, that is 1 2. Therefore the non overlapping bridges we are looking for are (8, 1) and (10, 2).

Code :

```
int maxBridges(int n)
{
 //for a single point there is always a bridge
 if(n == 1)
  return 1;
 //Sort the points according to the first sequence
  sort(points.begin(), points.end());
 //Apply LIS on the second sequence
 int i,j,res = 0;
 //Initialize LIS values for all indexes
 for (i = 0; i < n; i++ )
  lis[i] = 1;
 //Compute optimized LIS values in bottom up manner
 for (i = 1; i < n; i++ )
  for (j = 0; j < i; j++ )
   if ( points[i].second >= points[j].second && lis[i] < lis[j] + 1)
     lis[i] = lis[j] + 1;
 //Pick maximum of all LIS values
  for (i = 0; i < n; i++ )
   if (res < lis[i])
     res = lis[i];
return res;
}
```

Solution:http://pastebin.com/UdHtgasV

❑ LONGEST COMMON SUBSEQUENCE

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", '"acefg", ... etc are subsequences of "abcdefg". So a string of length n has 2^n different possible subsequences.

Example :

LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4

Recurrence Relation :

LCS(str1, str2, m, n) = 0, if m = 0 or n = 0 //Base Case

LCS(str1, str2, m, n) = 1 + LCS(str1, str2, m-1, n-1), if str1[m] = str2[n]

LCS(str1, str2, m, n) = max{LCS(str1, str2, m-1, n), LCS(str1, str2, m, n-1)}, otherwise

LCS can take value between 0 and min(m,n).

Code :

```
int lcs( char *str1, char *str2, int m, int n )
{
//Following steps build L[m+1][n+1] in bottom up fashion. Note
//that L[i][j] contains length of LCS of str1[0..i-1] and str2[0..j-1]
for (int i=0; i<=m; i++)
 {
  for (int j=0; j<=n; j++)
   {
    if (i == 0 || j == 0)
     LCS[i][j] = 0;
    else if (str1[i-1] == str2[j-1])
      LCS[i][j] = LCS[i-1][j-1] + 1;
    else
      LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1]);
   }
 }
//Print LCS
```

```
  int i = m, j = n, index = LCS[m][n];
 //array containing the final LCS
  char *lcs_arr = new char[index];
  while (i > 0 && j > 0)
   {
   // If current character in str1[] and str2[] are same, then
   // current character is part of LCS
    if (str1[i-1] == str2[j-1])
     {
      // Put current character in result
       lcs_arr[index-1] = str1[i-1];
      // reduce values of i, j and index
        i--;
        j--;
        index--;
     }
      // If not same, then find the larger of two and
     // go in the direction of larger value
     else if (LCS[i-1][j] > LCS[i][j-1])
       i--;
     else
       j--;
 }
// Print the lcs
 cout << "LCS of " << str1 << " and " << str2 << " is "<< lcs_arr << endl;
//L[m][n] contains length of LCS for str1[0..n-1] and
 str2[0..m-1]
 return LCS[m][n];
}
```

T = O(mn)

SHORTEST COMMON SUPERSEQUENCE OF TWO STRINGS

A supersequence is defined as the shortest string that has both str1 and str2 as subsequences.

str1 = "apple", str2 = "les"

"apples"

str1 = "AGGTAB", str2 = "GXTXAYB"

"AGGXTXAYB"

http://www.spoj.com/problems/ADFRUITS/

Approach:

- First we will find the LCS of the two strings.
- We will insert the non-LCS characters in the LCS found above in their original order.

```
void lcs( string str1, string str2, int m, int n )
{
 //Following steps build L[m+1][n+1] in bottom up fashion. Note
 //that L[i][j] contains length of LCS of str1[0..i-1] and str2[0..j-1]
 for (int i=0; i<=m; i++)
 {
   for (int j=0; j<=n; j++)
   {
    if (i == 0 || j == 0)
     LCS[i][j] = 0;
    else if (str1[i-1] == str2[j-1])
     LCS[i][j] = LCS[i-1][j-1] + 1;
   else
     LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1]);
   }
 }
int i = m, j = n, index = LCS[m][n];
//array containing the final LCS string lcs_arr;
while (i > 0 && j > 0)
 {
 // If current character in str1[] and str2[] are same, then
 // current character is part of LCS
```

```
    if (str1[i-1] == str2[j-1])
    {
     // Put current character in result
      lcs_arr += str1[i-1];
     // reduce values of i, j and index
      i--;
      j--;
      index--;
    }
// If not same, then find the larger of two and
// go in the direction of larger value
    else if (LCS[i-1][j] > LCS[i][j-1])
      i--;
    else
      j--;
 }
//Print LCS
//cout<<lcs_arr<<endl;
//Use LCS to get the final answer
 i = m-1, j = n-1;
//Use to make the final array
 int l = 0,k = 0;
//contains the length of longest common subsequence of the two strings
 index = LCS[m][n];
//string containing the final answer
 string ans = "";
while(i>=0 || j>=0)
{
 //If we have not covered the full LCS array
 if(l < index)
 {
   //Find the non-LCS characters of A and put them in a array
```

```
      //in reverse order
      while(i >= 0 && str1[i] != lcs_arr[l])
         ans += str1[i--];
    //Find the non-LCS charactes of B and put them in a array
    //in reverse order
      while(j >= 0 && str2[j] != lcs_arr[l])
        ans += str2[j--];
    //Now both the last value of str1 and str2 are equal
    //to the last value of LCS string
        ans += lcs_arr[l++];
       j--;
       i--;
  }
 //If we have exhausted all of our lcs_array and now we
 //just have to merge the non-lcs characters of both the strings
  else
  {
     while(i >= 0)
       ans += str1[i--];
     while(j >= 0)
       ans += str2[j--];
  }
}
for(int i = ans.length()-1;i>=0;i--)
   cout<<ans[i];
cout<<endl;
}
```

Solution: http://pastebin.com/EB9U8PNu

EDIT DISTANCE

Problem Statement: Given two strings str1 and str2 and below operations can be performed on str1. Find min number of edits(operations) required to convert str1 to str2.

- Insert
- Remove
- Replace

All the above operations are of equal cost.

http://www.spoj.com/problems/EDIST/

Example :

str1 = "cat"

str2 = "cut"

Replace 'a' with 'u', min number of edits = 1

str1 = "sunday"

str2 = "saturday"

Last 3 characters are same, we only need to replace "un" with "atur".

Replace n—>r and insert 'a' and 't' before 'u', min number of edits = 3

Recurrence Relation :

```
if str1[m] = str2[n]
        editDist(str1, str2, m, n) = editDist(str1, str2, m-1, n-1)
else
        editDist(str1, str2, m, n) = 1 + min{editDist(str1, str2, m-1, n) //Remove
        editDist(str1, str2, m, n-1) //Insert
        editDist(str1, str2, m-1, n-1) //Replace
}
```

Transform "Sunday" to "Saturday" :

Last 3 are same, so ignore. We will transform Sun —> Satur

(Sun, Satur) —> (Su, Satu) //Replace 'n' with 'r', cost= 1

(Su, Satu) —> (S, Sat) //Ignore 'u', cost = 0

(S, Sat) —> (S,Sa) //Insert 't', cost = 1

(S,Sa) —> (S,S) //Insert 'a', cost = 1

(S,S) —> ('','') //Ignore 'S', cost = 0

('', '') —> return 0

Dynamic Programming :

```
int editDist(string str1, string str2){
int m = str1.length();
int n = str2.length();
// dp[i][j] -> the minimum number of edits to transform str1[0...i-1] to
str2[0...j-1]
//Fill up the dp table in bottom up fashion
for(int i = 0;i<=m;i++)
{
for(int j = 0;j<=n;j++)
{
//If both strings are empty
 if(i == 0 && j == 0)
   dp[i][j] = 0;
//If first string is empty, only option is to
//insert all characters of second string
//So number of edits is the length of secondstring
 else if(i == 0)
   dp[i][j] = j;
//If second string is empty, only option is to
//remove all characters of first string
//So number of edits is the length of first string
 else if(j == 0)
   dp[i][j] = i;
//If the last character of the two strings are
//same, ignore this character and recur for the
//remaining string
 else if(str1[i-1] == str2[j-1])
   dp[i][j] = dp[i-1][j-1];
//If last character is different, we need at least one
//edit to make them same. Consider all the possibilities
//and find minimum
 else
   dp[i][j] = 1 + min(min(
              dp[i-1][j], //Remove
```

```
                dp[i][j-1]}, //Insert
                dp[i-1][j-1] //Replace
                 };

    }
}
//Return the most optimal solution
return  dp[m][n];

}
```

Solution: http://pastebin.com/ZVqfmHHJ

MIXTURES

Problem Statement : Given n mixtures on a table, where each mixture has one of 100 different colors (0 - 99). When mixing two mixtures of color 'a' and 'b', resulting mixture have the color (a + b) mod 100 and amount of smoke generated is a*b. Find the minimum amount of smoke that we can get when mixing all mixtures together, given that we can only mix two adjacent mixtures.

http://www.spoj.com/problems/MIXTURES/

The first thing to notice here is that, if we mix mixtures i...j into a single mixture, irrespective of the steps taken to achieve this, the final color of the mixture is same and equal to sum(i,j) = sum(color(i)..color(j)) mod 100.

So we define dp(i,j) as the most optimum solution where least amount of smoke is produced while mixing the mixtures from i...j into a single mixture. For achieving this, at the previous steps, we would have had to combine the two mixtures which are resultants of ranges i...k and k+1...j where i <= k <= j.

So it's about splitting the mixture into 2 subsets and each subset into 2 more subsets and so on such that smoke produced is minimized. Hence the recurrence relation will be:

dp(i,j) = min(k: i <= k < j) {dp(i,k) + dp(k+1,j) + sum(i,k)*sum(k+1,j)}

Code :

```
int  minSmoke(int  n)
{
  //Building the cummulative sum array
   sum[0] = col[0];
   for(int  i = 1;i<n;i++)
      sum[i] = (sum[i-1] + col[i]);
  //dp[i][j] -> min smoke produced after mixing {color(i).....color(j)}
  //Note color after mixing {color(i).....color(j)} is sum[i...j] mod M
  //Building the dp in bottom up fashion
```

```
        for(int i = n-1;i>=0;i--)
        {
          for(int j = 0;j<n;j++)
          {
            //Base Case
            //if i and j are equal then we have a single mixture and
            //hence no smoke is produced
            if(i == j)
            {
             dp[i][i] = 0;
              continue;
            }
           dp[i][j] = LONG_MAX;
           for(int k = i;k<j;k++)
           {
            int color_left = (sum[k] - sum[i-1]) % 100;
            int color_right = (sum[j] - sum[k]) % 100;
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + (color_left *
color_right));
           }
        }
      }
     //return the most optimal answer
      return dp[0][n-1];
    }
```

0-1 KNAPSACK PROBLEM

For each item you are given its weight and its value. You want to maximize the total value of all the items you are going to put in the knapsack such that the total weight of items is less than knapsack's capacity. What is this maximum total value?

http://www.spoj.com/problems/KNAPSACK/

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1. Maximum value obtained by n-1 items and W weight (excludingnth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

Recurrence Relation :

```
//Base case
//If we have explored all the items all we have reached the maximum capacity of Knapsack
if (n=0 or W=0)
 return 0
//If the weight of nth item is greater than the capacity of knapsack, we cannot include //this item
if (wieght[n] > W)
 return solve(n-1, W)
otherwise
 return max{ solve(n-1, W), //We have not included the item
solve(n-1, W-weight[n]) //We have included the item in the knapsack
}
```

If we build the recursion tree for the above relation, we can clearly see that the **property of overlapping sub-problems** is satisfied. So, we will try to solve it using dynamic programming.

Let us define the dp solution with states i and j as
dp[i,j] —> max value that can be obtained with objects u
pto index i and knapsack capacity of j.

The most optimal solution to the problem will be **dp[N][W]** i.e. max value that can be obtained upto index N with max capacity of W.

Code :

```
int knapsack(int N, int W)
{
 for(int i = 0;i<=N;i++)
  {
   for(int j = 0;j<=W;j++)
    {
     //Base case
```

```
        //When no object is to be explored or our knapsack's capacity is 0
        if(i == 0 || j == 0)
          dp[i][j] = 0;
        //When the wieght of the item to be considered is more than the
        //knapsack's capacity, we will not include this item
        if(wt[i-1] > j)
           dp[i][j] = dp[i-1][j];
        else
           dp[1][j] = max(
                      //If we include this item, we get a value of val[i-1] but
the
                      //capacity of the knapsack gets reduced by the weight of
that
                     //item.
                      val[i-1] + dp[i-1][j - wt[i-1]],
                      //If we do not include this item, max value will be the
                      //solution obtained by taking objects upto index i-1,
capacity
                      //of knapsack will remain unchanged.
                     dp[i-1][j]);
      }
    }
  return dp[N][W];
}
```

Time Complexity := O(NW)

Space Complexity := O(NW)

Can we do better?

If we observe carefully, we can see that the dp solution with states (i,j) will depend on state (i-1, j) or (i-1, j-wt[i-1]). In either case the solution for state (i,j) will lie in the **i-1**th row of the memoization table. So at every iteration of the index, we can copy the values of current row and use only this row for building the solution in next iteration and no other row will be used. Hence, at any iteration we will be using only a **single** row to build the solution for current row. Hence, we can reduce the space complexity to just **O(W)**.

Space-Optimized DP Code :

```
int knapsack(int N, int W)
{
 for(int  j = 0;j<=W;j++)
       dp[0][j] = 0;
 for(int  i = 0;i<=N;i++)
 {
   for(int  j = 0;j<=W;j++)
   {
   //Base case
   //When no object is to be explored or our knapsack's capacity is 0
    if(i == 0 || j == 0)
       dp[1][j] = 0;
   //When the wieght of the item to be considered is more than the
   //knapsack's capacity, we will not include this item
    if(wt[i-1] > j)
       dp[1][j] = dp[0][j];
    else
       dp[1][j] = max(
                  //If we include this item, we get a value of val[i-1] but the
                  //capacity of the knapsack gets reduced by the weight of that
                 //item.
                   val[i-1] + dp[0][j - wt[i-1]],
                 //If we do not include this item, max value will be the
                  //solution obtained by taking objects upto index i-1, capacity
                  //of knapsack will remain unchanged.
                  dp[0][j]);
   }
       //Here we are copying value of current row into the previous row,
        //which will be used in building the solution for next iteration ofrow.
    for(int  j = 0;j<=W;j++)
       dp[0][j] = dp[1][j];
  }
return  dp[1][W];
}
```

Time Complexity: O(N*W)

Space Complexity: O(W)

ROD CUTTING PROBLEM

You are given a rod of size n > 1, it can be cut into any number of pieces k. Price for each piece of size i is represented as p(i) and maximum revenue from a rod of size i is r(i) (could be split into multiple pieces). Find r(n) for the rod of size n.

Example :

Let the length of the rod is 6.

Price of size is given below

Length	1	2	3	4	5	6
Price(p)	2	5	8	9	10	11

Sol. Possible number cut on given rod :

Rod(6)	1*6	1*4,2	1*3, 3	1,2,3	1*2,4	1,5	2*3	2,4	3,3
Price(p)	12	13	14	15	13	12	15	13	16

Max Revenue : 16 (3,3)

Approach : Here we have to generate all the configurations of different pieces and find the highest priced configuration. We can get the best price(maximum revenue) by making a cut at different positions and comparing the values obtained after a cut.

Optimal Substructure Property :

Let the maximum revenue be r(n).

then, r(n) = max [p(n), p(1) + r(n - 1), p(2)+r(n-2),....., p(n-1)+r(1)]

$r(n) = \max[p(i) + r(n-i)] \; \forall \; 1 \le i \le (n)$

if RodCut(n) be the function which give the value of r(n) then

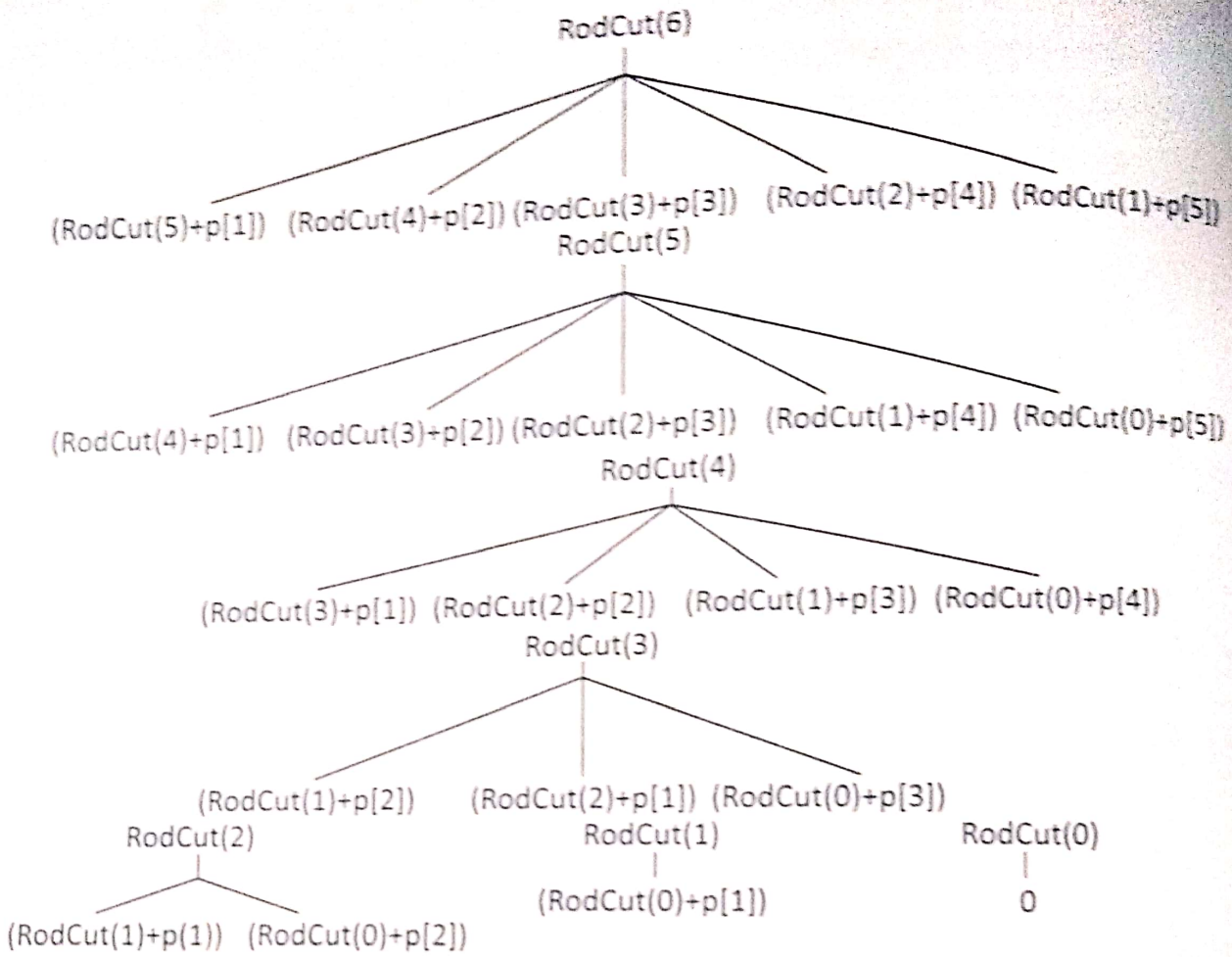
RodCut(n) = $\max[p(i) + \text{RodCut}(i)]] \; \forall \; 1 \le i \le (n)$

Code :

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b)
{
      return (a>b)?a:b;
}
```

```
int RodCut(int p[], int n)
{
      int max_revenue = INT_MIN;
      if(n<=0)
            return 0;
      else
      {
            for(int i=1;i<=(n);i++)
            {
                                          max_revenue =
max(max_revenue,p[i]+RodCut(p,n-i));
            }
      }
      return max_revenue;
}
int main()
{
      /* code */
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {
            int n;
            cin>>n;
            int p[n+1];
            p[0]=0;
            for(int i=1;i<=n;i++)
            {
                cin>>p[i];
            }
            cout<<"Maximum Revenue of given rod of length "<< n << " =
"<<RodCut(p,n)<<"\n";
      }
      return 0;
}
```

Overlapping Subproblems :



Time Complexity : Exponential Time Complexity

A rod of length n can have n – 1 exactly cut positions. We can choose any the k cut (without repe
anywhere we want so that for each such k the number of different choices is

$$\binom{n-1}{k}$$

when we sum up over ll possibilities for 0 £ k £ n – 1 then

$$\sum_{k=0}^{n-1}\binom{n-1}{k} = \sum_{k=0}^{n-1}\frac{(n-1)!}{k! * (n-1-k)!} = 2^{n-1}$$

Top Down Method :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
```

```
int max(int x, int y)
{
	return (x>y)?x:y;
}
int RodCut(int p[], int n)
{
	int Revenue[n+1];
    Revenue[0]=0;
    for(int i=1;i<=n;i++)
    {
	int best_price = INT_MIN;
	for(int j=1;j<=i;j++)
	{
		best_price = max(best_price, p[j]+Revenue[i-j]);
	}
	Revenue[i]=best_price;
    }
    return Revenue[n];
}
int main()
{
	/* code */
	int t;
	cin>>t;
	for(int i=0;i<t;i++)
	{
		int n;
		cin>>n;
		int p[n+1];
		p[0]=0;
		for(int i=1;i<=n;i++)
		{
			cin>>p[i];

		}
		cout<<"Maximum Revenue of given rod of length "<< n << " =
"<<RodCut(p,n)<<"\n";
	}
	return 0;
}
int Revenue[MAX];
int max(int a, int b)
```

```
    {
        return (a>b)?a:b;
    }
    int RodCut(int p[], int n)
    {
        int max_revenue = INT_MIN;
        if(n<=0)
                return 0;
            else if(Revenue[n]==-1)
        {
            for(int i=1;i<=(n);i++)
            {
                max_revenue = max(max_revenue,p[i]+RodCut(p,n-i));
            }
            Revenue[n] = max_revenue;
            return Revenue[n];
          }
           else
           return Revenue[n];
}
int main()
{
    /* code */
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        int p[n+1];
        p[0]=0;
        for(int i=1;i<=n;i++)
        {
                                cin>>p[i];
                                // Revenue[i]=p[i];
```

```
        }
        for(int i=0;i<=n;i++)
                Revenue[i] = -1;


        cout<<"Maximum Revenue of given rod of length "<< n << " =
"<<RodCut(p,n)<<"\n";
    }
    return 0;
}
```

Bottom Up :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int max(int x, int y)
{
    return (x>y)?x:y;
}
int RodCut(int p[], int n)
{
    int Revenue[n+1];
   Revenue[0]=0;
   for(int i=1;i<=n;i++)
  {
    int best_price = INT_MIN;
    for(int j=1;j<=i;j++)
    {
        best_price = max(best_price, p[j]+Revenue[i-j]);
    }
    Revenue[i]=best_price;
  }
  return Revenue[n];
}
int main()
```

```
{
        /* code */
        int t;
        cin>>t;
        for(int i=0;i<t;i++)
        {
            int n;
            cin>>n;
            int p[n+1];
            p[0]=0;
            for(int i=1;i<=n;i++)
            {
                                    cin>>p[i];
            }
            cout<<"Maximum Revenue of given rod of length "<< n <<" =
"<<RodCut(p,n)<<"\n";
        }
        return 0;
}
```

❑ LONGEST PALINDROME SUBSEQUENCE

Given a sequence of character, find the length of the longest palindromic subsequence in it.

Ex. if the given sequence is "BBABCBCAB", then the output should be 7 as "BABCBAB" is the longest palindromic subsequence in it.

Note : For given sequence "BBABCBCAB", subsquence "BBBBB" and "BBCBB" are also palindromic subsequnce but these are not longest.

Approach : The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence.

Optimal substructure Property :

Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].

i.e. $X = A_1 A_2 A_3 \ldots\ldots\ldots\ldots A_{n-1} A_n$

Recursive Relation :

$$L[i, j] = \begin{cases} 2 + L[i+1, j-1], & \text{if } X[i] = X[j] \\ \max(L[i, j-1], m[i+1, j]), & \text{if } X[i] \neq [j] \end{cases}$$

Base Cases :

$$L[i, j] = \begin{cases} 1, & \text{if } i = j \\ 2, & \text{if } i = j-1, X[i] = X[j] \end{cases}$$

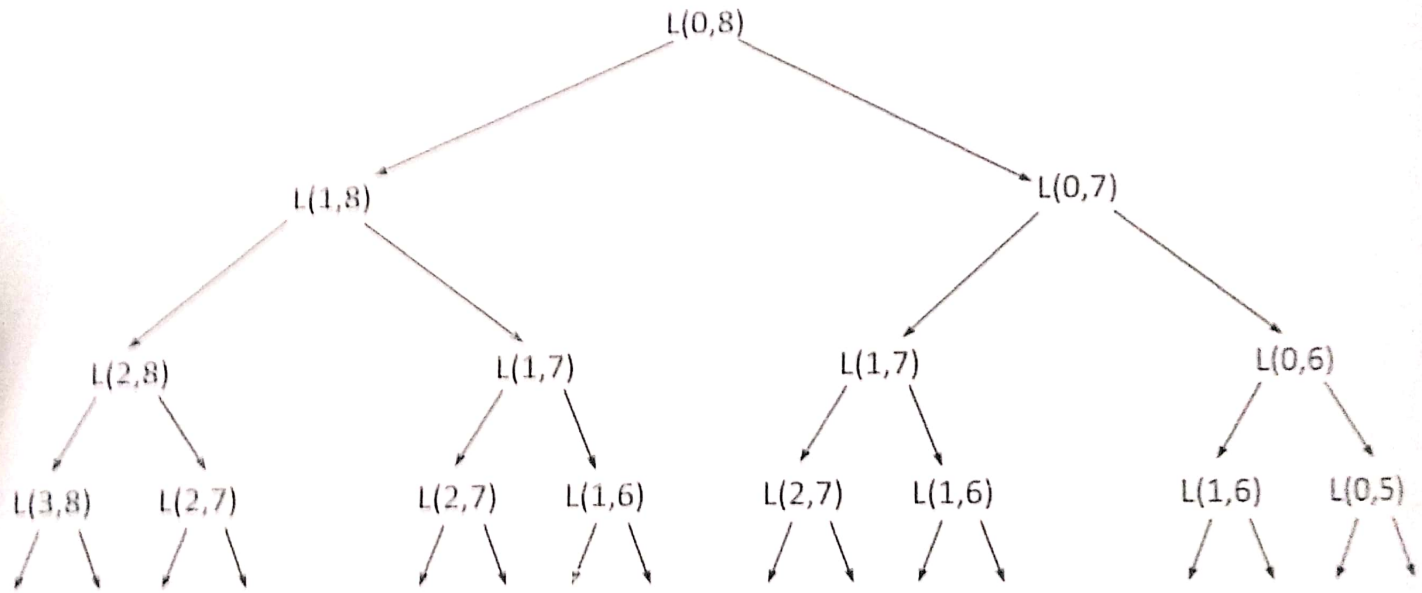
tart the code with i = 0 and j = n – 1;

ode :

```
nclude <iostream>
nclude <bits/stdc++.h>
efine MAX 200
ing namespace std;
t SubSeq(char seq[], int i, int j)

    int length_SubSeq =0;
 if(i==j)
    return 1;
 else if(i==(j-1) && seq[i]==seq[j])
 {
          return 2;
 }
  else if(seq[i]==seq[j])
    length_SubSeq = SubSeq(seq,i+1,j-1)+2;
 else
    length_SubSeq = max(SubSeq(seq,i+1,j), SubSeq(seq, i, j-1));
 return length_SubSeq;
}
int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {

     int n;
```

```
        cin>>n;
                char seq[n];
        for(int j=0;j<n;j++)
        {
       cin>>seq[j];
        }
          cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,0,n-
1)<<"\n";
      }
}
```

Recursion Tree (Overlapping Subproblems) :



Top Down (Memoization) :

```
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int length[MAX][MAX];
int SubSeq(char seq[], int i, int j)
{
      if(length[i][j]!=-1)
                  return length[i][j];
            else
                {
  if(i==j)
```

```
    length[i][j]=1;
  else if(seq[i]==seq[j] && i==(j-1))
  {
     length[i][j]=2;
  }
  else if(seq[i]==seq[j])
    length[i][j] = SubSeq(seq,i+1,j-1)+2;
  else
    length[i][j] = max(SubSeq(seq,i+1,j), SubSeq(seq, i, j-1));
 }

  return length[i][j];
}
int main()
{
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {


       int n;
       cin>>n;
            char seq[n];
       for(int j=0;j<n;j++)
       {
     cin>>seq[j];
       }
       for(int j=0;j<n;j++)
       {
          for(int k=0;k<n;k++)
            length[j][k]=-1;

       }
     cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,0,n-1)<<"\n";
     }
}
```

Bottom Up (Tabulation) :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int SubSeq(char seq[], int n)
{
      int length[n][n];
      for(int i=0;i<n;i++)
            length[i][i]=1;
      for(int k=2;k<=n;k++)
      {
            for(int i=0;i<=(n-k);i++)
            {
                  int j = k+i-1;
                  if(k==2 && seq[i]==seq[j])
                  {
                              length[i][j]=2;
                  }
                  else
                  {
                        if(seq[i]==seq[j])
                        {
                              length[i][j] = 2+length[i+1][j-1];
                        }
                        else
                        {
                              length[i][j]=max(length[i][j-1],length[i+1][j]);
                        }
                  }
            }
      }
      return length[0][n-1];
}
int main()
{
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {
      int n;
```

```
cin>>n;
        char seq[n];
for(int j=0;j<n;j++)
{
cin>>seq[j];
 }
   cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,n)<<"\n";
}
```

MATRIX CHAIN MULTIPLICATION

We are given a sequence(chain) (A_1 and A_2 , ... , A_n) of n matrices to be multiplied. Our work is to find the most efficient way to multiply these matrices together.

Since matrix multiplication is associative, So we have many option to multiply a chain of matrices. ex. Let the chain of matrices is (A_1, A_2, A_3, A_4) and we need to find $A_1 . A_2 . A_3 . A_4$.

There are multiple ways to find this.

1. $((A_1 . A_2) . (A_3 . A_4))$ 2. $((A_1 (A_2 . A_3)) . A_4))$ 3. $(A_1 . ((A_2 . A_3)) . A_4))$
4. $(((A_1 . A_2) . A_3) . A_4)$ 5. $(A_1 . (A_2 . (A_3 . A_4)))$

Multiplication Cost :

Let the size of the matrix is given $A_1 = 40 * 20$, $A_2 = 20 * 30$, $A_3 = 30 * 10$, $A_4 = 10 * 30$

Then multiplication cost for 1st multiplication would be :

(A_1, A_2) : Cost $= (40 * 20 * 30)$, Matrix Size $= 40 * 30$

(A_3, A_4) : Cost $= (30 * 10 * 30)$, Matrix Size $= 30 * 30$

Total Cost $= (40 * 10 * 30) + (30 * 10 * 30) + (40 * 30 * 30) = 70200$ size of Resultant Matrix = $40 * 30$

Total Cost = (40 * 20 * 30) + (30 * 10 * 30) + (40 * 30 * 30) = 70200 Size of Resultant Matrix = 40 * 30

Similarly cost for other multiplication would be given below :

	Multiplication	Cost
1.	$((A_1, A_2), (A_3, A_4))$	$(40 * 20 * 30) + (30 * 10 * 30) + (40 * 30 * 30) = 70200$
2.	$((A_1, (A_2, A_3)), A_4))$	$20 * 30 * 10 + 40 * 20 * 10 + 40 * 10 * 30 = 26000$
3.	$((A_1, (A_2, A_3)), A_4))$	$20 * 30 * 10 + 20 * 10 * 30 + 40 * 20 * 30 = 36000$
4.	$(((A_1, A_2), A_3), A_4)$	$40 * 20 * 30 + 40 * 30 * 10 + 40 * 10 * 30 = 48000$
5.	$(A_1, (A_2, (A_3, A_4)))$	$30 * 10 * 30 + 20 * 30 * 30 + 40 * 20 * 30 = 51000$

Approach :

Approach towards the solution is to place parentheses at all possible places, calculate the cost for each placement and return the minimum value.

Suppose p[n + 1] is an array contain the size of each matrix. i.e.

size of A_i = p[i − 1] * p[i], $\forall$ 1 d ≤ i d ≤ n

Let m[i,j] be the minimum number scalar multiplication needed to compute the multiplication of matrix A_i to A_j (i.e. $A_i \cdot A_{i+1} \cdot A_{i+2} \cdot \cdots \cdot A_j$).

So m[1,n] would be the lowest cost to compute the multiplication of matrix to (i.e. $A_1 \cdot A_2 \cdot A_3 \cdot \cdots \cdot A_n$).

A recursive Solution :

We can split the product the product $A_i \cdot A_{i+1} \cdot A_{i+2} \cdot \cdots \cdot A_j$ between A_k and A_{k+1} where i ≤ k < j then m[i,j] equals the minimum cost for computing the subproducts $A_{i \ldots k}$ and $A_{k+1 \ldots j}$, plus the cost of multiplying these two matrices together.

- Size of $A_{i \ldots k}$ = p[i-1]*p[k] and size of $A_{k+1 \ldots j}$ = p[k]*p[j]
- then multiplication cost of $A_{i \ldots k} * A_{k+1 \ldots j}$ = p[i − 1] * p[k] * p[j]
- $A_{i \ldots k}$ = minimum cost to multiply the matrices $(A_i \cdot A_{i+1} \cdot A_{i+2} \cdot \cdots \cdot A_k)$ = m[i, k]
- $A_{k+1 \ldots j}$ = minimum cost to multiply the matrices= m[k + 1, j]
- m[i, j] = m[i, k] + m[k + 1, j] + p[i − 1] * p[k] * p[j]

  m[i,j] = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]

Possible value of k is j − i i.e. k = i, i + 1, i + 2, ..., j − 1

We need to check them all values of k to find the best(minimum cost i.e optimal solution).

if i==j

  m[i, j] = 0;

else

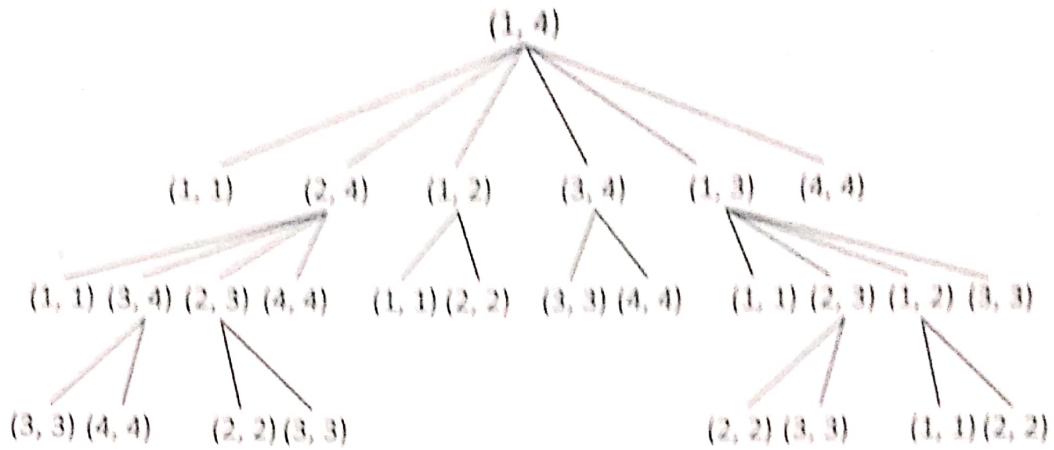
  m[i,j] = min(m[i, k] + m[k + 1, j] + p[i − 1] * p[k] * p[j]), $\forall$ i ≤ k < j

Code :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int min(int x, int y)
{
      return (x>y)?y:x;
}
```

```
int Matrix_Chain(int p[], int i,int j)
{
      int mult_cost = INT_MAX;
      if(i==j)
            return 0;
      else
      {
      for(int k=i;k<j;k++)
     {
      mult_cost = min(mult_cost,
                  Matrix_Chain(p,i,k)+Matrix_Chain(p,k+1,j)+p[i-1]*p[k]*p[j]);
     }
      }
      return mult_cost;
}
int main()
{
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {
            int n;
            cin>>n;
            int p[n+1];
            for(int j=0;j<=n;j++)
            {
                  cin>>p[j];
            }
     cout<<"Minimum cost for matrix multiplication is :"<<"
"<<Matrix_Chain(p,1,n)<<"\n";
   }
}
```

Recursive Tree Diagram (Overlapping Subproblems) :



Top Down (Memoization):

Code :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int mult[MAX][MAX];
int min(int x, int y)
{
      return (x>y)?y:x;
}
int Matrix_Chain(int p[], int i,int j)
{
      if(mult[i][j]!=-1)
            return mult[i][j];
      else
      {
            int mult_cost = INT_MAX;
          if(i==j)
              mult[i][j]=0;
          else
           {
          for(int k=i;k<j;k++)
           {
```

```
            mult_cost = min(mult_cost,
                  Matrix_Chain(p,i,k)+Matrix_Chain(p,k+1,j)+p[i-1]*p[k]*p[j]);
        }
         mult[i][j]=mult_cost;
        }
         return mult[i][j];
    }
    //return  mult_cost;
}
int main()
{
      int t;
      cin>>t;
      for(int  i=0;i<t;i++)
      {
            int n;
            cin>>n;
            int  p[n+1];
            for(int  j=0;j<=n;j++)
            {
                  cin>>p[j];
            }
            for(int  j=0;j<MAX;j++)
            {
                  for(int  k=0;k<MAX;k++)
                  {
                        mult[j][k]=-1;
                  }
            }
      cout<<"Minimum cost for matrix multiplication is :"<<"
"<<Matrix_Chain(p,1,n)<<"\n";
   }
}
```

Bottom Up (Tabulation) :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int min(int x, int y)
{
      return (x>y)?y:x;
}
int Matrix_Chain(int p[], int n)
{
      int mult_cost[n][n];
      for(int i=1;i<n;i++)
      {
            mult_cost[i][i]=0;
      }
      for(int l=2;l<n;l++)
      {
            for(int i=1;i<(n-l+1);i++)
            {
                  int j=i+l-1;
                  mult_cost[i][j] = INT_MAX;
                  for(int k=i;k<j;k++)
                  {
                        mult_cost[i][j] = min(mult_cost[i][j],
                              mult_cost[i][k]+mult_cost[k+1][j]+p[i-
1]*p[k]*p[j]);
                  }
            }
      }
      return mult_cost[1][n-1];
}
int main()
{
      int t;
```

```
    cin>>t;
    for(int  i=0;i<t;i++)
    {
          int n;
          cin>>n;
          int p[n+1];
          for(int  j=0;j<=n;j++)
          {
                cin>>p[j];
          }

    cout<<"Minimum  cost  for  matrix  multiplication  is  :"<<"
"<<Matrix_Chain(p,n+1)<<"\n";
   }
}
```

DO IT YOURSELVES

- ❑ Follow Coding Blocks tutorials on Dynamic Programming with Bitmasks.
- ❑ Try the Dynamic Programming Section problems on HackerBlocks.

10 MO's Algorithm

Query SQRT Decomposition

Mo's algorithm is a generic idea. It applies to the following class of problems :

You are given array Arr of length N and Q queries. Each q uery is represented by two numbers L and R, and it asks you to compute some function with subarray Arr[L..R] as its argument.

Let's Start with a Simple Problem :

Given an array of size **N**. All elements of array <= **N**. You need to answer **M** queries. Each query is of the form **L, R**. You need to answer the count of values in range **[L, R]** which are repeated at least **3** times.

Example: Let the array be {1, 2, 3, 1, 1, 2, 1, 2, 3, 1} (zero indexed)

Query: L = 0, R = 4. Answer = **1**. Values in the range [L, R] = {1, 2, 3, 1, 1} only 1 is repeated at least 3 times.

Query: L = 1, R = 8. Answer = **2**. Values in the range [L, R] = {2, 3, 1, 1, 2, 1, 2, 3} 1 is repeated 3 times and 2 is repeated 3 times.

Number of elements repeated at least 3 times = Answer = **2**.

Now explain a simple solution which takes $O(N^2)$

```
for each query:
answer = 0
count[] = 0
for i in {l..r}:
    count[array[i]]++
    if count[array[i]] == 3:
        answer++
```

Slight Modification to above algorithm. It still runs in $O(N^2)$.

```
add(position):
  count[array[position]]++
  if count[array[position]] == 3:
    answer++

remove(position):
  count[array[position]]--
```

```
    if count[array[position]] == 2:
      answer-

currentL = 0
currentR = 0
answer = 0
count[] = 0
for each query:
    // currentL should go to L, currentR should go to R
    while currentL <= L:
        remove(currentL)
        currentL++
    while currentL >= L:
      add(currentL)
      currentL-
    while currentR <= R:
        add(currentR)
        currentR++
    while currentR >= R:
      remove(currentR)
      currentR-
    output answer
```

Initially we always looped from **L to R**, but now we are changing the positions from **previous query to adjust to current query.**

If previous query was **L=3, R=10**, then we will have **currentL=3** and **currentR=10** by the end of that query. Now if the next query is **L=5, R=7**, then we move the **currentL** to **5** and **currentR** to **7**.

add function means we are adding the element at position to our current set. And updating answer accordingly.

remove function means we are deleting the element at position from our current set. **And updating answer accordingly.**

MO's algorithm is just an order in which we process the queries. We were given M queries, we will re-order the queries in a particular order and then process them. Clearly, this is an offline algorithm. Each query has L and R, we will call them opening and closing. Let us divide the given input array into Sqrt(N) blocks. Each block will be N / Sqrt(N) = Sqrt(N) size. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.

In this algorithm we will process the queries of **1st block**. Then we process the queries of **2nd block** and so on... finally **Sqrt(N)'th** block. We already have an ordering, queries are ordered in the ascending order of its block. There can be many queries that belong to the same block.

Let's focus on how we query and answer **block 1**. We will similarly do for all blocks. All of these queries have their opening in **block 1**, but their closing can be in any block including **block 1**. Now let us reorder these queries in ascending order of their **R value**. We do this for all the blocks.

How does the final order look like?

All the queries are first ordered in ascending order of their block number (block number is the block in which its opening falls). Ties are ordered in ascending order of their R value.

For example consider following queries and assume we have 3 blocks each of size 3.

{0, 3} {1, 7} {2, 8} {7, 8} {4, 8} {4, 4} {1, 2}

Let us re-order them based on their block number.

{0, 3} {1, 7} {2, 8} {1, 2} {4, 8} {4, 4} {7, 8}

Now let us re-order ties based on their R value.

{1, 2} {0, 3} {1, 7} {2, 8} {4, 4} {4, 8} {7, 8}

Now we use the same code stated in previous section and solve the problem. Above algorithm is correct as we did not do any changes but just reordered the queries.

Proof for complexity of above algorithm – O(Sqrt(N) * N) :

We are done with MO's algorithm, it is just an ordering.

It turns out that the O(N^2) code we wrote works in O(Sqrt(N) * N) time if we follow the above order.

With just reordering the queries we reduced the complexity from O(N^2) to O(Sqrt(N) * N), and that too with out any further modification to code.

Have a look at our code above, **the complexity over all queries is determined by the 4 while loops.** First 2 while loops can be stated as "**Amount moved by left pointer in total**", second 2 while loops can be stated as "**Amount moved by right pointer**". **Sum of these two will be the over all complexity.**

Let us talk about the right pointer first.

For each block, the queries are sorted in increasing order, so clearly the right pointer **(currentR)** moves in increasing order. During the start of next block the pointer possibly be at extreme end will move to least R in next block. That means for a given block, the amount moved by right pointer is **O(N)**.

We have **O(Sqrt(N))** blocks, so the total is **O(N * Sqrt(N))**.

Let us see how the left pointer moves.

For each block, the left pointer of all the queries fall in the same block, as we move from query to query the left pointer might move but as previous L and current L fall in the same block, the moment is **O(Sqrt(N)) (Size of the block)**. In each block the amount left pointer movies is **O(Q * Sqrt(N))** where **Q** is number of queries falling in that block. Total complexity is **O(M * Sqrt(N))** for all blocks.

So, total complexity is **O((N + M) * Sqrt(N)) = O(N * Sqrt(N))**.

This Algorithm is Offline :

That means we cannot use it when we are forced to stick to given order of queries. That also means we cannot use this when there are update operations. Not just that, there is one important possible limitation: We should be able to write the functions add and remove. There will be many cases where add is trivial but remove is not. One such example is where we want maximum in a range. As we add elements, we can keep track of maximum. But when we remove elements it is not trivial.

❑ DQUERY (Spoj)

Find number of distinct elements in a range l to r:

```
struct query {
    int l;
    int r;
    int in;
} q[200005];
```

How'll we sort query according to above defined way:

```
bool compare(query x, query y) {
    if (x.l / BLOCK != y.l / BLOCK) {
        // different blocks, so sort by block.
        return x.l / BLOCK < y.l / BLOCK;
    }
    // same block, so sort by R value
    return x.r < y.r;
}
```

Add and Remove :

```
void add(int cur) {
    cnt[arr[cur]]++;
    if (cnt[arr[cur]] == 1) {
        tans++;
    }
```

```
}
void remove(int cur) {
    cnt[arr[cur]]--;
    if (cnt[arr[cur]] == 0) {
        tans--;
    }
}
```

Main :

```
int main() {
    fastRead_int(n);
    loop(i, 0, n) {
        fastRead_int(arr[i]);
    }

    fastRead_int(t);
    loop(i, 0, t) {
        fastRead_int(a);
        fastRead_int(b);
        a--, b--;
        q[i].l = a, q[i].r = b, q[i].in = i;
    }

    //structure sorted
    sort(q, q + t, compare);
    int curL = 0, curR = 0;
    loop(i, 0, t) {

        while (curL < q[i].l) {
            remove(curL);
            curL++;
        }
        while (curL > q[i].l) {
            add(curL - 1);
            curL--;
        }
        while (curR <= q[i].r) {
```

```
            add(curR);
            curR++;
        }
        while (curR > q[i].r + 1) {
            remove(curR - 1);
            curR--;
        }
        ans[q[i].in] = tans;

    }
    for (int i = 0; i < t; ++i) {
        printf("%d\n", ans[i]);
    }
    return 0;
}
```

❑ POWERFUL ARRAY (CODE FORCES, 86-D)

An array of positive integers a_1, a_2, ..., a_n is given. Let us consider its arbitrary subarray al, al + 1 ..., ar, where **1 ≤ l ≤ r ≤ n**. For every positive integer s denote by **Ks** the number of occurrences of s into the subarray. We call the power of the subarray the sum of products **Ks·Ks·s** for every positive integer **s**. The sum contains only finite number of nonzero summands as the number of different values in the array is indeed finite.

You should calculate the power of **t** given subarrays.

First line contains two integers n and t **(1 ≤ n, t ≤ 200000)** — the array length and the number of queries correspondingly.

Second line contains n positive integers ai **(1 ≤ ai ≤ 10^6)** — the elements of the array.

Next **t** lines contain two positive integers **l, r (1 ≤ l ≤ r ≤ n)** each — the indices of the left and the right ends of the corresponding subarray.

Code :

```
ll arr[200005];
ll ans[200005];
ll cnt[1000005];

ll tans;

struct query {
```

```
    int l;
    int r;
    int in;
} q[200005];

bool compare(query x, query y) {
   if (x.l / BLOCK != y.l / BLOCK) {
        // different blocks, so sort by block.
        return x.l / BLOCK < y.l / BLOCK;
    }
    // same block, so sort by R value
    return x.r < y.r;
}

void add(int cur) {
    cnt[arr[cur]]++;
    if(cnt[arr[cur]]>0)
        tans += arr[cur] * (2 * cnt[arr[cur]] - 1);
}

void remove(int cur) {
    cnt[arr[cur]]--;
    if(cnt[arr[cur]]>=0)
        tans -= arr[cur] * (2 * cnt[arr[cur]] + 1);
}

int main() {
    //ios_base::sync_with_stdio(false);
    //cin.tie(NULL);

    fastRead_int(n), fastRead_int(t);

    loop(i, 0, n) {
        fastRead_lint(arr[i]);
    }
    loop(i, 0, t) {
        fastRead_int(a), fastRead_int(b);
```

```
        a--, b--;
        q[i].l = a, q[i].r = b, q[i].in = i;
    }

    //structure sorted
    sort(q, q + t, compare);
    int curL = 0, curR = 0;
    loop(i, 0, t) {

        while (curL < q[i].l) {
            remove(curL);
            curL++;
        }
        while (curL > q[i].l) {
            add(curL-1);
            curL--;
        }
        while (curR <= q[i].r) {
            add(curR);
            curR++;
        }
        while (curR > q[i].r + 1) {
            remove(curR - 1);
            curR--;
        }
        ans[q[i].in] = tans;
    }

    for (int i = 0; i < t; ++i) {
        prl(ans[i]);
        nl;
    }

    return 0;
}
```

TIME TO TRY

Tree and Queries :

You have a rooted tree consisting of n vertices. Each vertex of the tree has some color. We will assume that the tree vertices are numbered by integers from 1 to n. Then we represent the color of vertex v as cv. The tree root is a vertex with number 1.

In this problem you need to answer to m queries. Each query is described by two integers vj,kj. The answer to query vj,kj is the number of such colors of vertices x, that the subtree of vertex vj contains at least kj vertices of color x.

INPUT :

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

OUTPUT :

2
2
1
0
1

11 Graph Algorithms

Introduction :

Graphs are mathematical structures that represent pairwise relationships between objects. A graph is a flow structure that represents the relationship between various objects. It can be visualized by using the following two basic components :

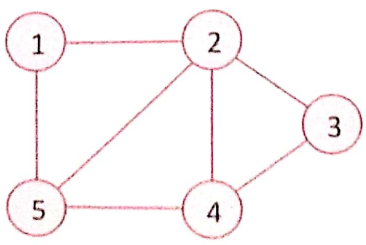
- **Nodes :** These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- **Edges :** Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

Some Real Life Applications:

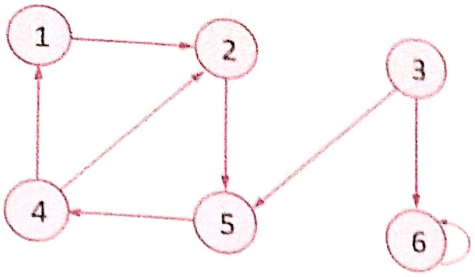
- **Google Maps :** To find a route based on shortest route/Time.
- **Social Networks :** Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
- **Web Search :** Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks, each page is a vertex and the link between two pages is an edge.
- **Recommendation System :** On eCommerce websites relationship graphs are used to show recommendations.

Types of Graphs :

- **Undirected :** An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.

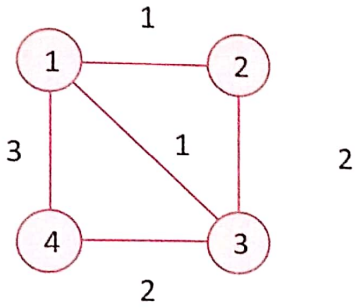


- **Directed :** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.

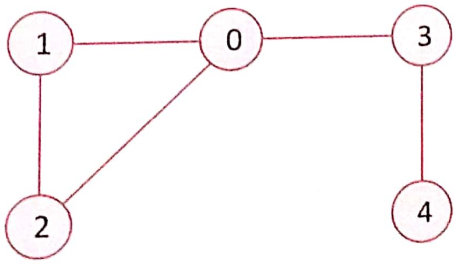


- **Weighted :** In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:
  - 1 -> 2 -> 3
  - 1 -> 3
  - 1 -> 4 -> 3

Therefore the total cost of each path will be as follows: The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



- **Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

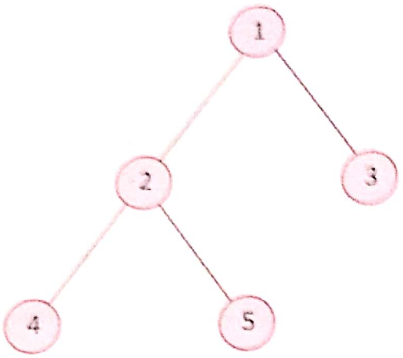


Cycle presents in the above graph (0->1->2)

A tree is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

Note: A root node has no parent.

A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



(Tree : There is no any cycle or self loop in this graph)

Graph Representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

1. Adjacency Matrix

An adjacency matrix is a V*V binary matrix A. Element $A_{i,j}$ is 1 if there is an edge from vertex i to vertex j else $A_{i,j}$ is 0.

Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in $A_{i,j}$ the weight or cost of the edge will be stored.

In an undirected graph, if $A_{i,j} = 1$, then $A_{j,i} = 1$.

In a directed graph, if $A_{i,j} = 1$, then may or may not be 1.

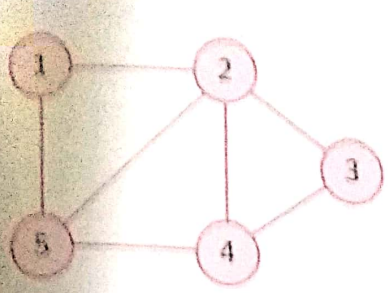
Adjacency matrix provides constant time access (O(1)) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is $O(V^2)$.

2. Adjacency List

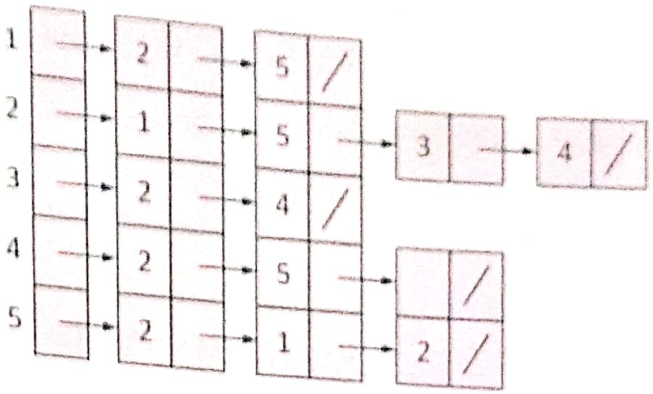
The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array A_i is a list, which contains all the vertices that are adjacent to vertex i.

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list A_i then vertex i will be in list A_j.

The space complexity of adjacency list is O(V + E)because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.



(a)

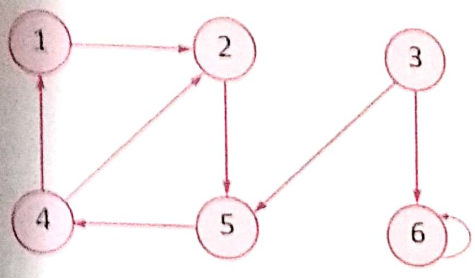


(b)

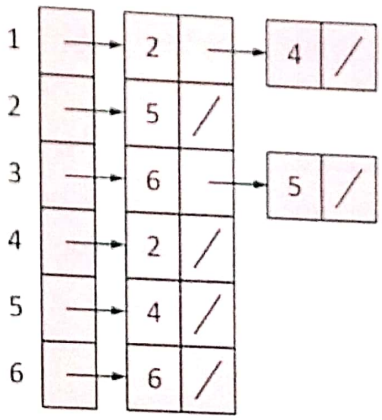
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Two representations of an undirected graph.
(a) An undirected graph G having five vertices and seven edges
(b) An adjacency-list representation of G
(c) The adjacency-matrix representation of G



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Code for Adjacency list representation of a graph :

```
#include<iostream>
#include<list>
using namespace std;
class Graph{
    int V;
    list<int> *l;
public:
    Graph(int v){
        V = v;
        //Array of Linked Lists
        l = new list<int>[V];
    }
    void addEdge(int u,int v,bool bidir=true){
```

```
            l[u].push_back(v);
            if(bidir){
                  l[v].push_back(u);
            }
      }
      void printAdjList(){
            for(int i=0;i<V;i++){
                        cout<<i<<"->";
                        //l[i] is a linked list
                        for(int vertex: l[i]){
                              cout<<vertex<<",";
                        }
                        cout<<endl;
            }
      }
};
int main(){
      // Graph has 5 vertices number from 0 to 4
      Graph g(5);
      g.addEdge(0,1);
      g.addEdge(0,4);
      g.addEdge(4,3);
      g.addEdge(1,4);
      g.addEdge(1,2);
      g.addEdge(2,3);
      g.addEdge(1,3);
      g.printAdjList();
return 0;
}
```

Graph Traversal :

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS) :

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

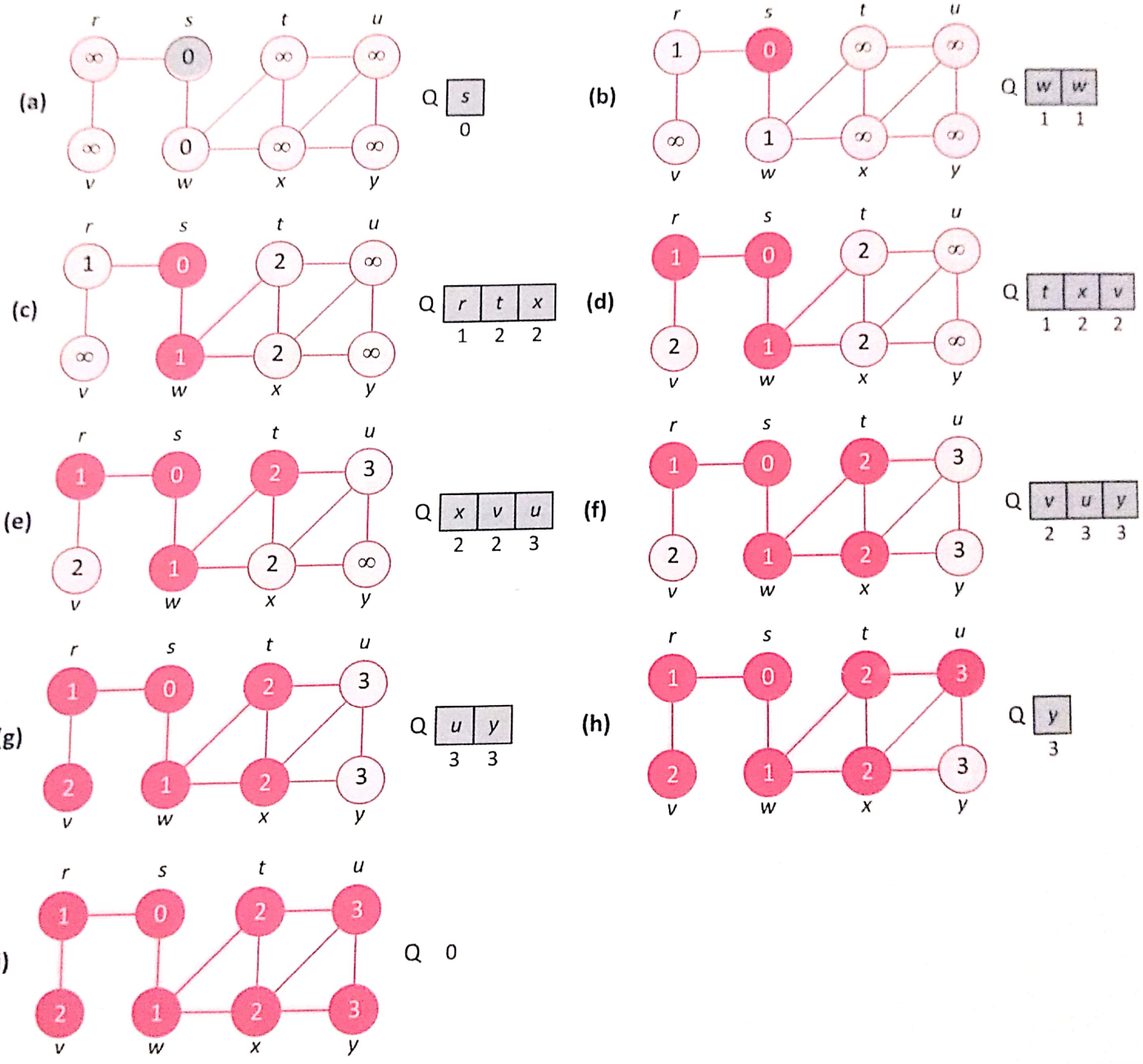
Breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is *discovered* the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If (u, v) " E and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the *predecessor* or *parent* of v in the breadth-first tree.

Algorithm :

```
BFS(G, s)
    for each vertex u ∈ V [G] - {s}
        do color[u] ← WHITE
           d[u] ← ∞
           π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← 0
    ENQUEUE(Q, s)
    while Q ≠
        do u ← DEQUEUE(Q)
           for each ∈ Adj[u]
               do if color[v] = WHITE
                  then color[v] ← GRAY
```

```
        d[v] ← d[u] + 1
        π[v] ← u
        ENQUEUE(Q, v)

color[u] ← BLACK
```



The operation of BFS on an undirected graph. Tree edges are shown shaded as they are prduced by BFS. Within each vertex vertex u is shown d[u]. The queue Q is shown at the beginning of each iteration of the while loop of lines 10-18. Vertex distances are shown next to vertices in the queue.

Code :

```
#include<iostream>
#include<map>
#include<list>
#include<queue>
using namespace std;

template<typename T>
class Graph{

    map<T,list<T> > adjList;

public:
    Graph(){

    }
    void addEdge(T u, T v,bool bidir=true){

        adjList[u].push_back(v);
        if(bidir){
            adjList[v].push_back(u);
        }
    }

    void print(){
```

Time Complexity :

Time complexity and space complexity of above algorithm is O(V + E) and O(V).

Application of BFS

1. **Shortest Path and Minimum Spanning Tree for unweighted graph** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

Code for finding shortest path using BFS :

```
#include<iostream>
#include<map>
#include<list>
```

```
#include<queue>
using namespace std;
template<typename T>
class Graph{
        map<T,list<T> > adjList;
public:
        Graph(){
 }
        void addEdge(T u, T v,bool bidir=true){
                adjList[u].push_back(v);
        if(bidir){
                adjList[v].push_back(u);
        }
    }

    void print(){
        //Iterate over the map
        for(auto i:adjList){
                cout<<i.first<<"->";
        //i.second is LL
        for(T entry:i.second){
                cout<<entry<<",";
        }
        cout<<endl;
    }
}
voidbfs(T src){
    queue<T> q;
    map<T,int>dist;
    map<T,T> parent;
    for(auto i:adjList){
            dist[i.first] = INT_MAX;
    }
    q.push(src);
```

```
        dist[src] = 0;
        parent[src] = src;
        while(!q.empty()){
            T node = q.front();
            cout<<node<<" ";
            q.pop();
            // For the neigbours of the current node, find out the nodes which
        are not visited
            for(intneigbour :adjList[node]){
                if(dist[neigbour]==INT_MAX){
                    q.push(neigbour);
                    dist[neigbour] = dist[node] + 1;
                    parent[neigbour] = node;
                }
            }
        }
        //Print the distance to all the nodes
        for(auto i:adjList){
            T node = i.first;
            cout<<"Dist of "<<node<<" from "<<src<<" is "<<dist[node]<<endl;
        }
    }
};
intmain(){
    Graph<int> g;
    g.addEdge(0,1);
    g.addEdge(1,2);
    g.addEdge(0,4);
    g.addEdge(2,4);
    g.addEdge(2,3);
    g.addEdge(3,5);
    g.addEdge(3,4);
    g.bfs(0);
}
```

2. **Peer to Peer Networks :** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines :** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
4. **Social Networking Websites :** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5. **GPS Navigation systems :** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network :** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **In Garbage Collection :** Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8. **Cycle detection in undirected graph :** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
9. **Ford–Fulkerson algorithm :** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).
10. **To test if a graph is Bipartite :** We can either use Breadth First or Depth First Traversal.
11. **Path Finding :** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12. **Finding all nodes within one connected component :** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

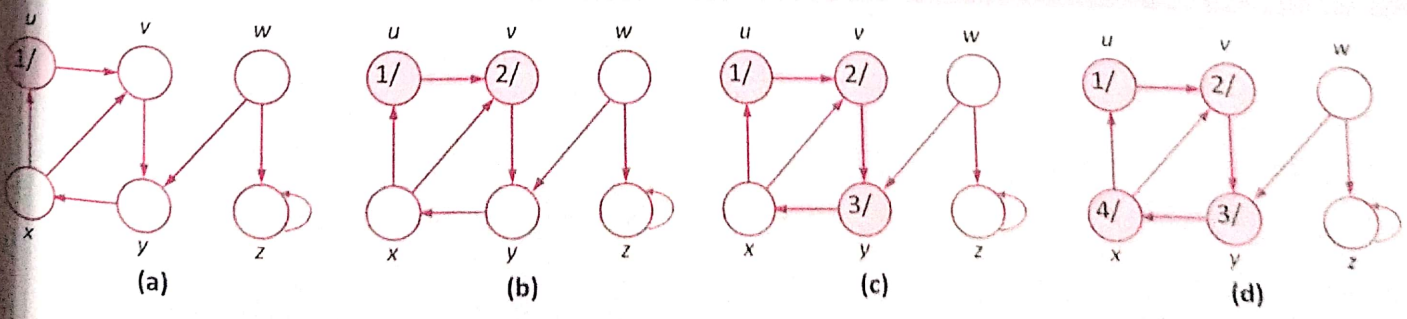
- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

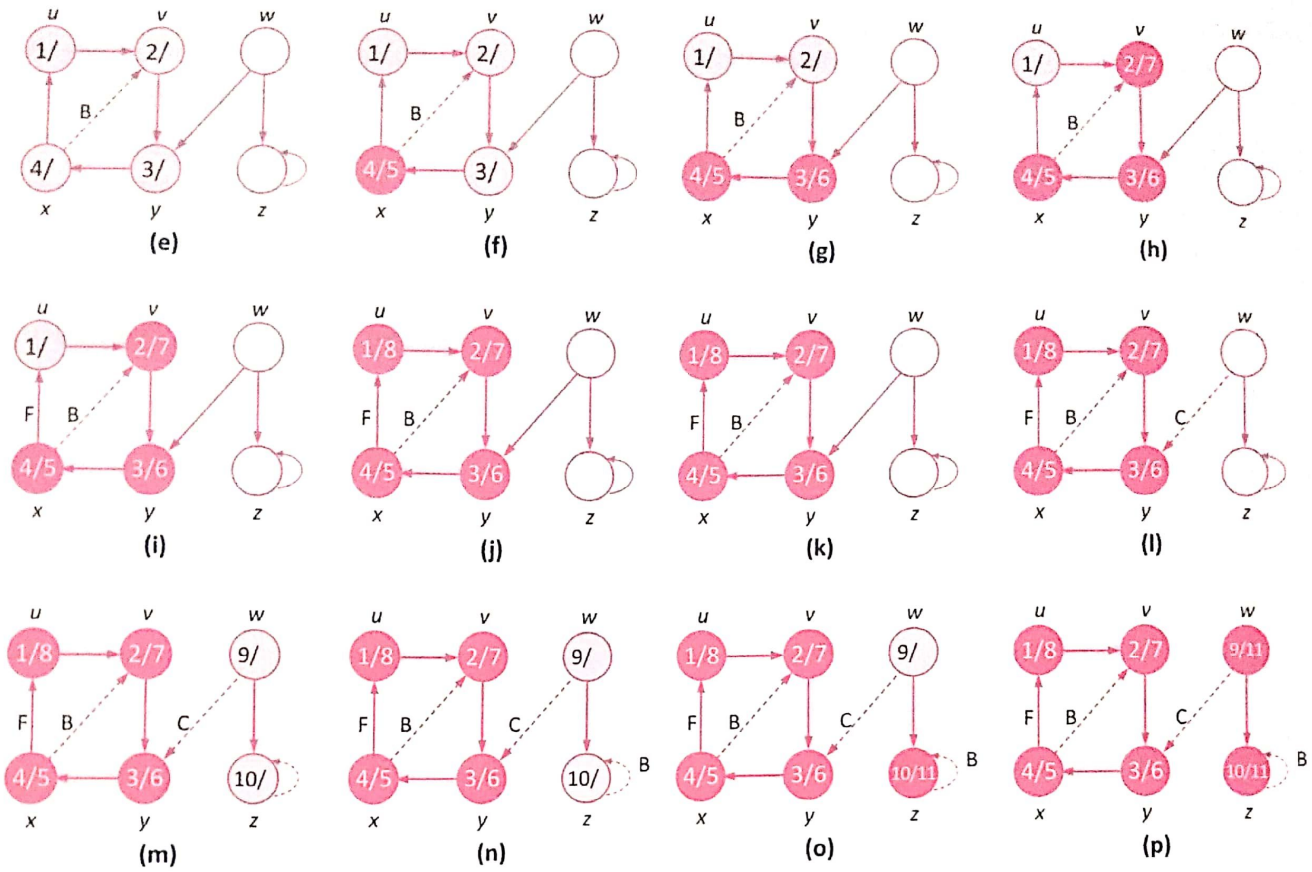
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Algorithm :

```
DFS(G)
for each vertex u ∈ V[G]
    do color [u] ← WHITE
       p[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
DFS-VISIT(u)
color[u] ← GRAY          White vertex u has just been discovered
time ← time + 1
d[u] ← time
for each v Î Adj[u]          Explore edge (u, v)
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] BLACK          Blacken u; it is finished
f[u] ← time ← time + 1
```





The progress of the depth-first-search algorithm DFS on a directed path. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

Code :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;

template<typename T>
class Graph{

    map<T,list<T> > adjList;

public:
    Graph(){

    }
    void addEdge(T u, T v,bool bidir=true){
        adjList[u].push_back(v);
```

```
            if(bidir){
                adjList[v].push_back(u);
            }
        }

        void print(){

            //Iterate over the map
            for(auto i:adjList){
                cout<<i.first<<"->";

                //i.second is LL
                for(T entry:i.second){
                    cout<<entry<<",";
                }
                cout<<endl;
            }
        }
        void dfsHelper(T node,map<T,bool> &visited){
            //Whenever to come to a node, mark it visited
            visited[node] = true;
            cout<<node<<" ";
            //Try to find out a node which is neigbour of current node and not
 yet visited
            for(T neighbour: adjList[node]){
                if(!visited[neighbour]){
                    dfsHelper(neighbour,visited);
                }
            }
        }
        void dfs(T src){
            map<T,bool> visited;
            dfsHelper(src,visited);

        }
};
int main(){

    Graph<int> g;
    g.addEdge(0,1);
    g.addEdge(1,2);
    g.addEdge(0,4);
```

```
        g.addEdge(2,4);
        g.addEdge(2,3);
        g.addEdge(3,4);
        g.addEdge(3,5);
        g.dfs(0);


    return 0;
    }
```

Time Complexity :

Time complexity of above algorithm is O(V + E).

Application of DFS :

1. For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
2. **Detecting cycle in a graph :** A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edge.

Code for detect a cycle in a graph :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;
template<typename T>
class Graph{
    map<T,list<T> > adjList;
public:
    Graph(){
    }
    void addEdge(T u, T v,bool bidir=true){
        adjList[u].push_back(v);
        if(bidir){
            adjList[v].push_back(u);
        }
    }
    bool isCyclicHelper(T node,map<T,bool> &visited,map<T,bool> &inStack){
```

```
//Processing the current node - Visited, InStack
visited[node] = true;
InStack[node] = true;
```

3. **Path Finding** : We can specialize the DFS algorithm to find a path between two given vertices u and z.
   (i) Call DFS(G, u) with u as the start vertex.
   (ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
   (iii) As soon as destination vertex z is encountered, return the path as the contents of the stack.

4. **Topological Sorting** : In the field of computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering.

Code for printing Topological sorting of DAG :

```
#include<iostream>
#include<map>
#include<list>
#include<queue>
using namespace std;
template<typename T>
class Graph{
    map<T,list<T> > adjList;
public:
    Graph(){
    }
    void addEdge(T u, T v,bool bidir=true){
        adjList[u].push_back(v);
        if(bidir){
            adjList[v].push_back(u);
        }
    }
    void topologicalSort(){
```

```
queue<T> q;
map<T,bool> visited;
map<T,int> indegree;
for(auto i:adjList){
        //i is pair of node and its list
        T node = i.first;
        visited[node] = false;
        indegree[node] = 0;
}
//Init the indegrees of all nodes
for(auto i:adjList){
      T u = i.first;
      for(T v: adjList[u]){
      indegree[v]++;
      }
}
//Find out all the nodes with 0 indegree
for(auto i:adjList){
      T node = i.first;
      if(indegree[node]==0){
            q.push(node);
      }
}
//Start with algorithm
while(!q.empty()){
      T node = q.front();
      q.pop();
      cout<<node<<"—>";
      for(T neighbour:adjList[node]){
            indegree[neighbour]—;
      if(indegree[neighbour]==0){
            q.push(neighbour);
      }
}
```

```
        }
    }
};

int main(){
    Graph<string> g;
    g.addEdge("English","ProgrammingLogic",false);
    g.addEdge("Maths","Programming Logic",false);
    g.addEdge("Programming Logic","HTML",false);
    g.addEdge("Programming Logic","Python",false);
    g.addEdge("Programming Logic","Java",false);
    g.addEdge("Programming Logic","JS",false);
    g.addEdge("Python","WebDev",false);
    g.addEdge("HTML","CSS",false);
    g.addEdge("CSS","JS",false);
    g.addEdge("JS","WebDev",false);
    g.addEdge("Java","WebDev",false);
    g.addEdge("Python","WebDev",false);
    g.topologicalSort();

return0;
}
```

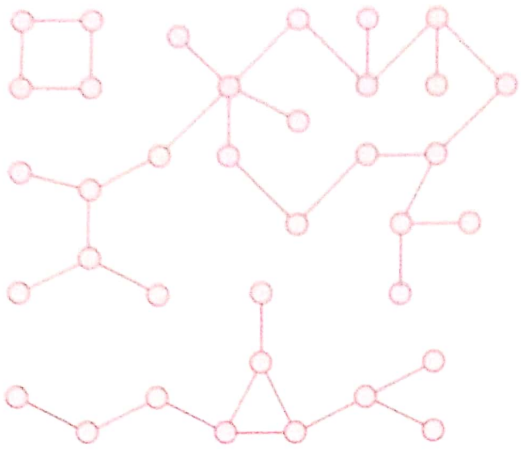
5. **To test if a graph is bipartite :** We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black.

6. **Finding Strongly Connected Components of a graph :** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

7. **Solving puzzles with only one solution,** such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

8. **For finding Connected Components :** In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Example :



In the above picture, there are three connected components.

Code for finding connected components :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;

template<typename T>
class Graph{
      map<T,list<T> > adjList;
public:
      Graph(){
      }
      void addEdge(T u, T v,bool bidir=true){
             adjList[u].push_back(v);
             if(bidir){
                    adjList[v].push_back(u);
             }
       }
       voidprint(){
            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";
                  //i.second is LL
```

```
            for(T entry:i.second){
                cout<<entry<<",";
            }
            cout<<endl;
        }
    }
    void dfsHelper(T node,map<T,bool> &visited){
        //Whenever to come to a node, mark it visited
        visited[node] = true;
        cout<<node<<" ";
        //Try to find out a node which is neigbour of current node and not yet visited
        for(T neighbour: adjList[node]){
            if(!visited[neighbour]){
                dfsHelper(neighbour,visited);
            }
        }
    }
    void dfs(T src){
        map<T,bool> visited;
        int component = 1;
        dfsHelper(src,visited);
        cout<<endl;
        for(auto i:adjList){
            T city = i.first;
            if(!visited[city]){
                dfsHelper(city,visited);
                component++;
            }
        }
    cout<<endl;
            cout<<"The current graph had "<<component<<" components";
        }
};
int main(){
    Graph<string> g;
```

```
        g.addEdge("Amritsar","Jaipur");
        g.addEdge("Amritsar","Delhi");
        g.addEdge("Delhi","Jaipur");
        g.addEdge("Mumbai","Jaipur");
        g.addEdge("Mumbai","Bhopal");
        g.addEdge("Delhi","Bhopal");
        g.addEdge("Mumbai","Bangalore");
        g.addEdge("Agra","Delhi");
        g.addEdge("Andaman","Nicobar");
        g.dfs("Amritsar");
return 0;
}
```

Minimum Spanning Tree

What is a Spanning Tree?

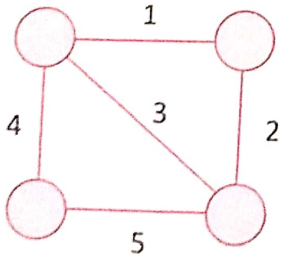
Given an undirected and connected graph G=(V,E) , a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

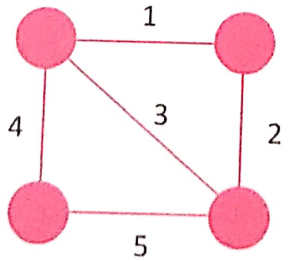
The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

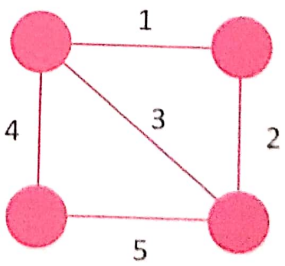


Undirected Graph



Spanning Tree

Cost = 11(= 4 + 5 + 2)



Minimum Spanning Tree

Cost = 7(= 4 + 1 + 2)

There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is "**Disjoint Sets**":

Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first.

Note : Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Algorithm :

```
MST-KRUSKAL(G, w)
A ← ∅
For each vertex v ∈ V[G]
    Do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) ∈ E, taken in nondecreasing order by weight
    do if FIND-SET(u)≠ FIND-SET(v)
          then A ← A ∪ {(u, v)}
               UNION(u, v)
return A
```

Here **A** is the set which contains all the edges of minimum spanning tree.

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is "**Disjoint Sets**":

Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

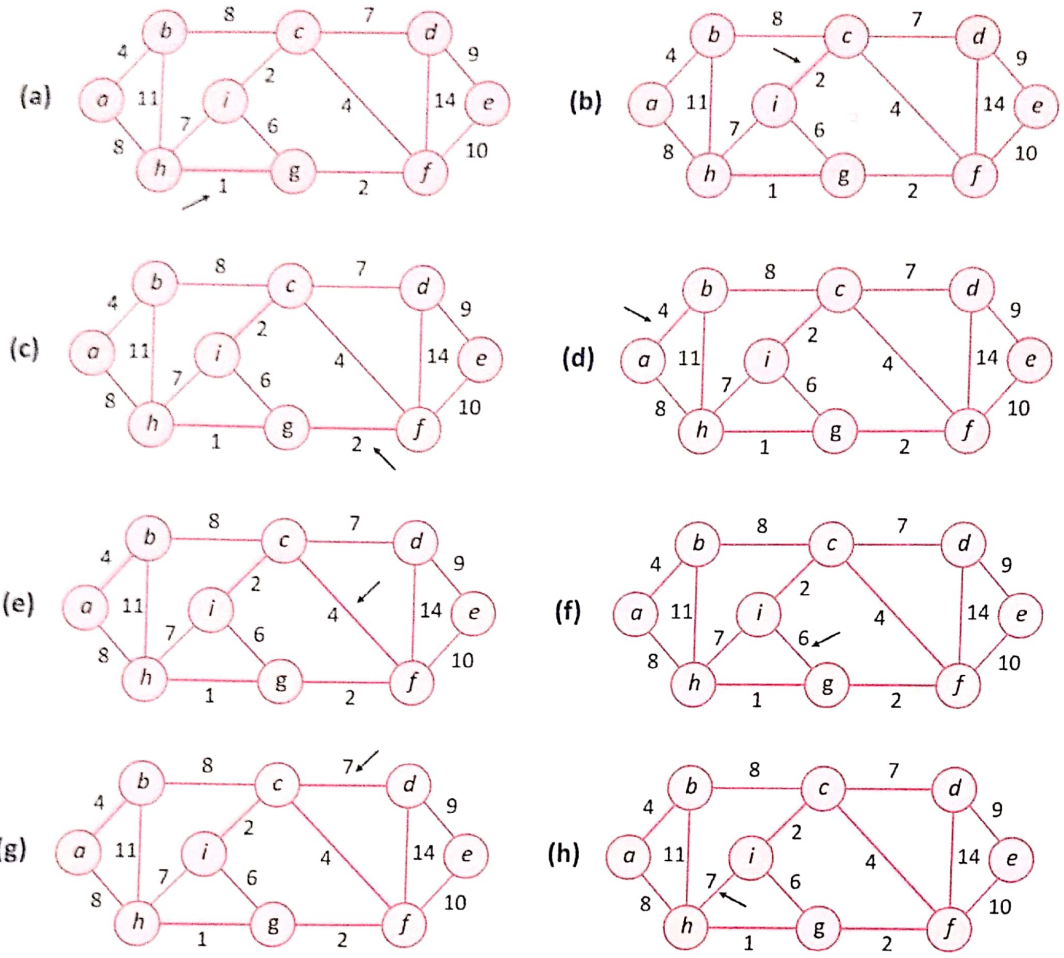
In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first.

Note : Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Algorithm :

```
MST-KRUSKAL(G, w)
A ← ∅
For each vertex v ∈ V[G]
    Do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) ∈ E, taken in nondecreasing order by weight
    do if FIND-SET(u)≠ FIND-SET(v)
          then A ← A ∪ {(u, v)}
               UNION(u, v)
return A
```

Here A is the set which contains all the edges of minimum spanning tree.



The execution of Kruskal's algorithm on the graph from figure. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Code :

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
    for(int i = 0;i < MAX;++i)
```

```
        id[i] = i;
}

int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}

void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
        // Selecting edges one by one in increasing order from the beginning
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
        }
```

```
    }
    return minimumCost;
}

int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in the ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity :

The total running time complexity of kruskal algorithm is O(V log E).

Prim's Algorithm :

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps :

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

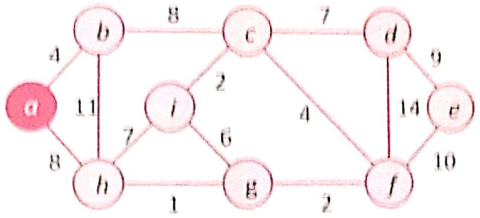
Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex.

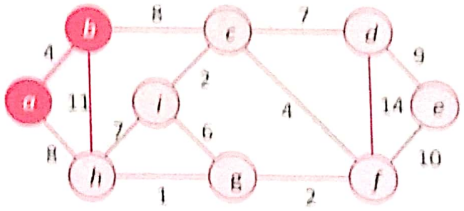
```
MST-PRISM(G, w, r)
    for each u ∈ V[G]
        do key[u] ← ∞
           p[u] ← NIL
    key[r] ← 0
    Q ← V[G]
    while Q ≠ 0
        do u ← EXTRACT-MIN(Q)
           for each v ∈ Adj[u]
               do if v ∈ Q and w(u, v) < key[v]
                     then π[v] ← u
                          key[v] ← w(u, v)
```

The prims algorithm works as shown in below graph.

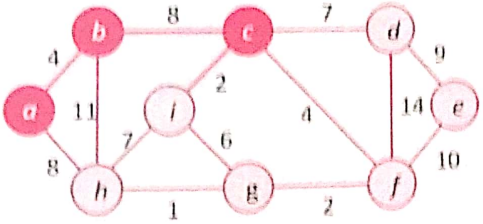
(a)



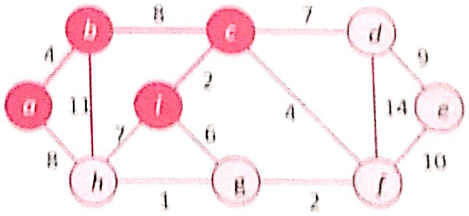
(b)



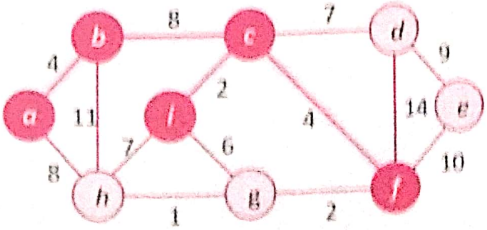
(c)



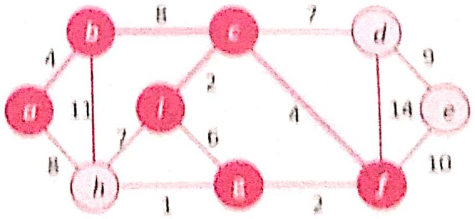
(d)



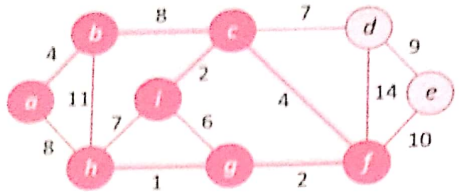
(e)



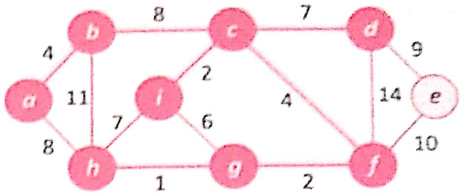
(f)



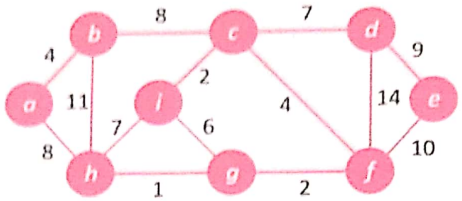
(g)



(h)



(i)



Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>
using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
   priority_queue<PII, vector<PII>, greater<PII> > Q;
  int y;
  long long minimumCost = 0;
  PII p;
  Q.push(make_pair(0, x));
  while(!Q.empty())
  {
      // Select the edge with minimum weight
      p = Q.top();
      Q.pop();
      x = p.second;
      // Checking for cycle
```

```
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}
int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
    }
    // Selecting 1 as the starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Shortest Path Algorithms

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

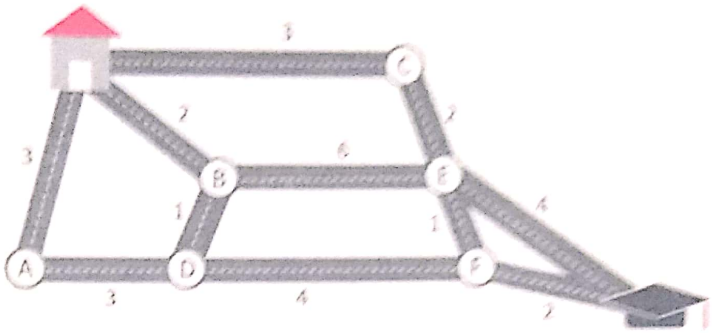
This problem could be solved easily using (BFS) if all edge weights were (1), but here weights can take any value. There are some algorithm discussed below which work to find Shortest path between two vertices.

1. Single Source Shortest Path Algorithm :

In this kind of problem, we need to find the shortest path of single vertices to all other vertices.

Example :

Find the shortest path from home to school in the following graph:



A weighted graph representing roads from home to school

The shortest path, which could be fund using Dijkstra's algorithm, is

Home → B → D → F → School

There are two algorithm works to find Single source shortest path from a given graph.

A. Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Pseudo Code :

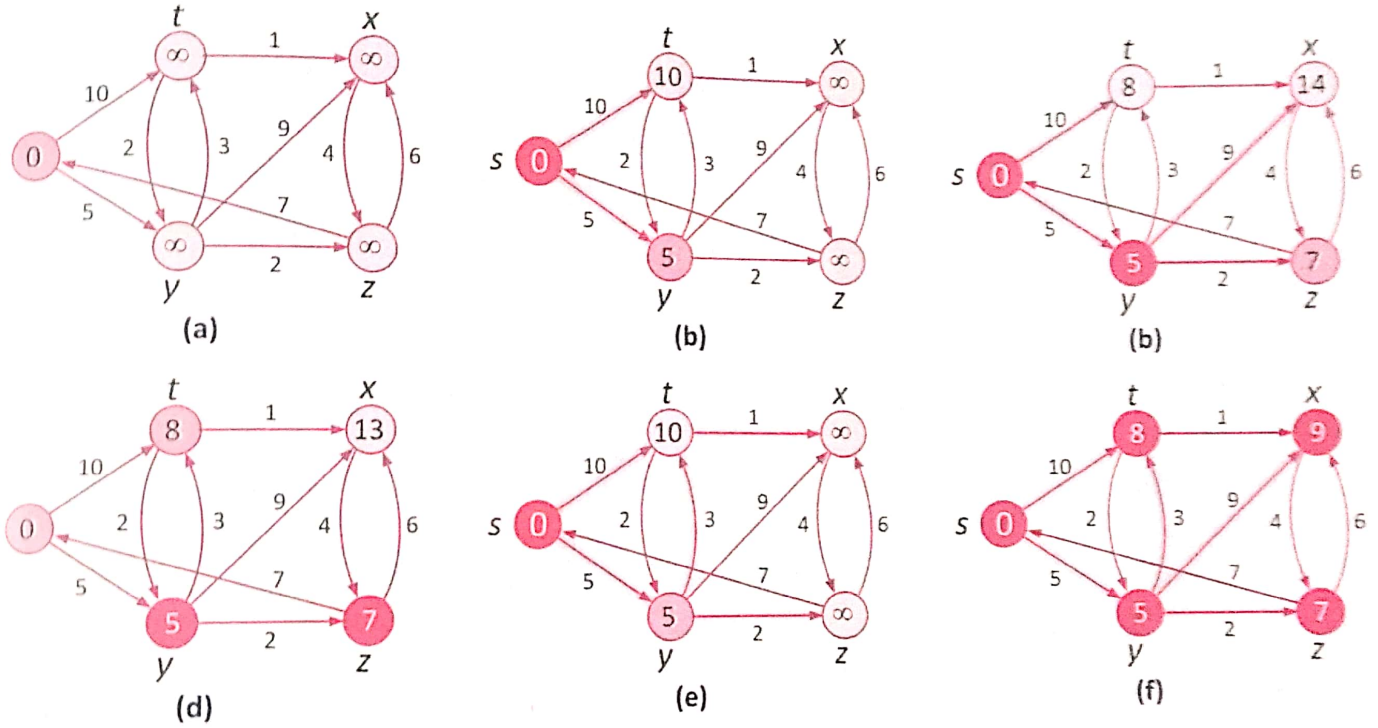
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅
```

```
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

For relaxing an edge of given graph, Algorithm works likes this :

```
RELAX(u, v, w)
    if d[v] > d[u] + w(u, v)
       then d[v] ← d[u] + w(u, v)
            π[v] ← u
```

Example :



he execution of Dijkstra's algorithm. The source *s* is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set *S*, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop. The shaded vertex has the minimum *d* value and is chosen as vertex *u* . **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex *u* in line 5 of the next iteration. The *d* and À values shown in part (f) are the final values.

Code :

```
#include<bits/stdc++.h>
using namespace std;
template<typename T>
class Graph{
    unordered_map<T, list<pair<T,int> > > m;
```

```
public:
    void addEdge(T u,T v,int dist,bool bidir=true){
        m[u].push_back(make_pair(v,dist));
        if(bidir){
            m[v].push_back(make_pair(u,dist));
        }
    }
    void printAdj(){
        //Let try to print the adj list
        //Iterate over all the key value pairs in the map
        for(auto j:m){
            cout<<j.first<<"->";
            //Iterater over the list of cities
            for(auto l: j.second){
                cout<<"("<<l.first<<","<<l.second<<")";
            }
            cout<<endl;
        }
    }
    void dijsktraSSSP(T src){
        unordered_map<T,int> dist;
        //Set all distance to infinity
        for(auto j:m){
            dist[j.first] = INT_MAX;
        }
        //Make a set to find a out node with the minimum distance
        set<pair<int, T> > s;
        dist[src] = 0;
        s.insert(make_pair(0,src));
        while(!s.empty()){
            //Find the pair at the front.
            auto p =   *(s.begin());
            T node = p.second;
            int nodeDist = p.first;
            s.erase(s.begin());
```

```
            //Iterate over neighbours/children of the current node
            for(auto childPair: m[node]){
                if(nodeDist + childPair.second < dist[childPair.first]){
                    //In the set updation of a particular is not possible
                    // we have to remove the old pair, and insert the new pair
to  simulation updation
                    T dest = childPair.first;
                    auto f = s.find( make_pair(dist[dest],dest));
                    if(f!=s.end()){
                        s.erase(f);
                    }
                    //Insert the new pair
                    dist[dest] = nodeDist + childPair.second;
                    s.insert(make_pair(dist[dest],dest));
                }
            }
        }
        //Lets print distance to all other node from src
        for(auto d:dist){
            cout<<d.first<<" is located at distance of  "<<d.second<<endl;
        }
    }
};
int main(){
    Graph<int> g;
    g.addEdge(1,2,1);
    g.addEdge(1,3,4);
    g.addEdge(2,3,1);
    g.addEdge(3,4,2);
    g.addEdge(1,4,7);
    //g.printAdj();
   // g.dijsktraSSSP(1);
    Graph<string> india;
    india.addEdge("Amritsar","Delhi",1);
    india.addEdge("Amritsar","Jaipur",4);
    india.addEdge("Jaipur","Delhi",2);
```

```
        india.addEdge("Jaipur","Mumbai",8);
        india.addEdge("Bhopal","Agra",2);
        india.addEdge("Mumbai","Bhopal",3);
        india.addEdge("Agra","Delhi",1);
       //india.printAdj();
        india.dijsktraSSSP("Amritsar");


return 0;
}
```

Time Complexity : Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to $O(V + E \log V)$.

B. Bellman Ford's Algorithm

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n – 1 edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle ?

There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

- The outer loop traverses from 0 to n – 1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

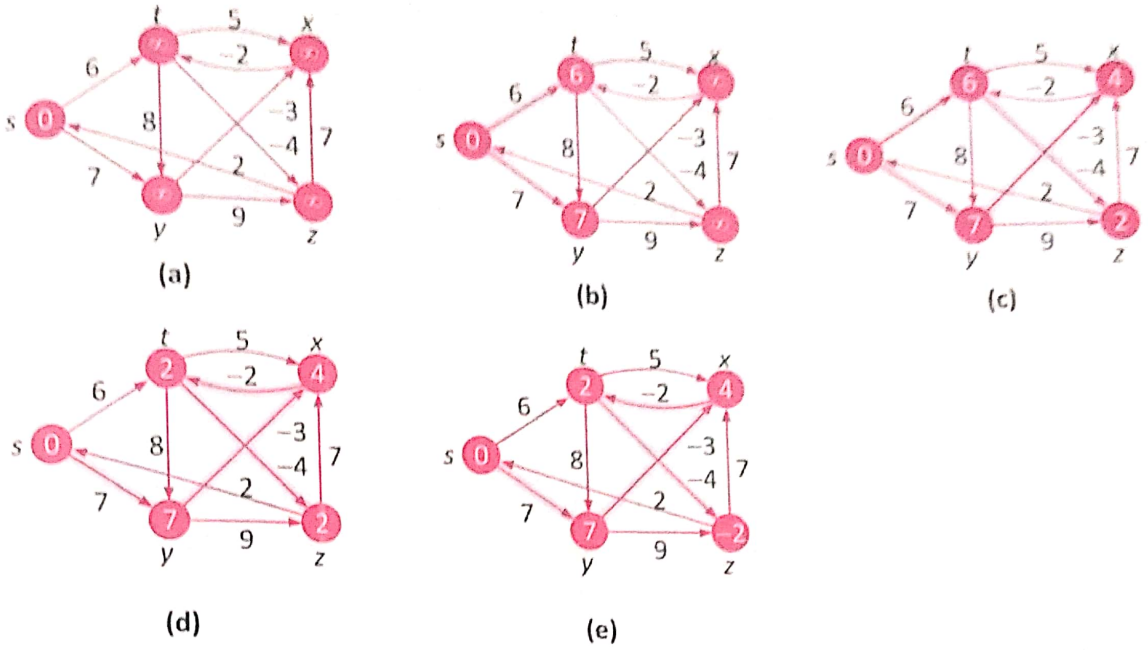
This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Pseudo Code :

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| - 1
      do for each edge (u, v) ∈ E[G]
            do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
```

```
    do if d[v] > d[u] + w(u, v)
          then return FALSE
return TRUE
```

Example :



All-Pairs Shortest Paths :

The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be solved using n applications of Dijkstra's algorithm at each vertex if graph doesn't contains negative weight. We use two algorithms for finding All-pair shortest path of a graph.

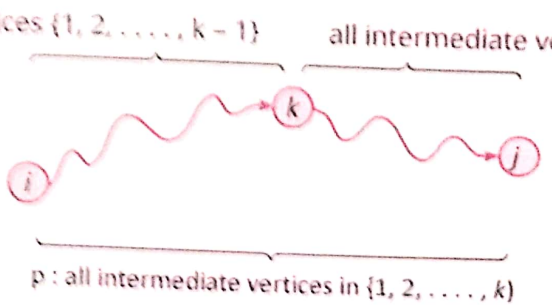
1. Floyd-Warshall's Algorithm
2. Johnson's Algorithm

Floyd-Warshall's Algorithm

Here we use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the ***Floyd-Warshall algorithm***. Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in $O(V^3)$, where V is the number of vertices in a graph.

The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1,2,\ldots,k\}$ of vertices for some k. For any pair of vertices i, $j - V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. The Floyd- Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p. There would two possibilities :

1. If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1,2,\ldots,k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1,2,\ldots,k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.
2. If k is an intermediate vertex of path p, then we break p down into $i \rightarrow k \rightarrow j$ as $p1$ is a shortest path from i to k *and* $p2$ is a shortest path from k to j. *Since* $p1$ is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. Because vertex k is not an intermediate vertex of path $p1$, we see that $p1$ is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly, $p2$ is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1,2,\ldots,k-1\}$.



Path P is a shortest path from vertices i to vertex j, and k is the highest=numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set {1, 2, ... , k – 1}. The same holds for path p_2 from vertex k to vertex j.

Let $d_{ij}^{(k)}$ be the weight of shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if} \quad k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if} \quad k \geq 1 \end{cases}$$

The Algorithm Steps:

For a graph with V vertices:

- Initialize the shortest paths between any 2 vertices with Infinity.
- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all V vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices (i , j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be:

Min (dist[i][k] + dist[k][j] , dist[i][j])

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k , j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

Constructing a shortest path

For construction of the shortest path, we can use the concept of predecessor matrix π to construct the path. We can compute the predecessor matrix π "on-line" just as the Floyd-Warshall algorithm computes the matrices $D(k)$. Specifically, we compute a sequence of matrices $\pi^{(k)}$ where $\pi^{(k)}$ is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1,2,\ldots,k\}$. We can give a recursive formulation of $\pi^{(k)}$ as

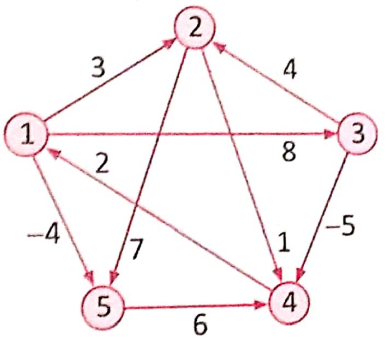
For k = 0 :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NILL} & \text{if} \quad i = j \text{ or } w_{ij} = \infty \\ i & \text{if} \quad i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

And For 1<=k<=V

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if} \quad d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ i & \text{if} \quad d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Consider the below graph and find distance (D) and parents (π) matrix for this graph



Computed distance (D) and parents (π) matrix for the above graph :

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Implemented Code :

```
#include<bits/stdc++.h>
#include<iostream>
#include<unordered_map>
#define INF 99999
#define V 5
using namespace std;
template<typename T>
class Graph
{
   unordered_map<T, list<pair<T, int> > > m;
   int graph[V][V];
   int parent[V][V];

public:
 //Adjacency list representation of the graph
 void addEdge(T u, T v, int dist,bool bidir = false)
 {
      m[u].push_back(make_pair(v,dist));
     /*if(bidir){
```

```
        m[v].push_back(make_pair(u,dist));
    }*/

}
//Print Adjacency list
void printAdj()
{
    for(auto j:m)
    {
        cout<<j.first<<"->";
        for(auto l: j.second)
        {
            cout<<"("<<l.first<<","<<l.second<<")";
        }
        cout<<endl;
    }

}
void matrix_form(int u, int v , int w)
{
  graph[u-1][v-1] = w;
  parent[u-1][v-1] = u;
  return;
}

//Adjacency matrix representation of the graph
 void matrix_form2()
{
    for(int i=0;i<V;i++)
    {
        for(int j=0;j<V;j++)
        {
            if(i==j)
            {
                graph[i][j]=0;
                parent[i][j]=0;
            }
```

```
            else
            {
                graph[i][j]=INF;
                parent[i][j]=0;
            }
        }
    }
    return;
}
//Print Adjacency matrix
 void print_matrix()
{
     for(int i=0;i<V;i++)
    {
         for(int j=0;j<V;j++)
        {
             if(graph[i][j]==INF)
             cout<<"INF"<<"    ";
            else
             cout<<graph[i][j]<<"     ";
        }
         cout<<endl;
    }

}

//Print predecessor matrix
 void printParents(int p[][V])
 {

     cout<<"Parents Matrix"<<"\n";
   for (int i = 0; i < V; i++)
   {
        for (int j = 0; j < V; j++)
       {
            if(p[i][j]!=0)
```

```
            cout<<p[i][j]<<"    ";
            else
            {
            cout<<"NIL"<<"  ";
            //cout<<p[i][j]<<"    ";
            }
        }
          cout<<endl;
    }
 }
 //All pair shortest path matrix i.e D
 void printSolution(int dist[][V])
{

            cout<<"Following matrix shows the shortest distances"
           " between every pair of vertices"<<"\n";
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                cout<<"INF   ";
            else
                cout<<dist[i][j]<<"   ";
        }
          cout<<endl;
    }
}
 //Print the shortest path , distance and all intermediate vertex
 void print_path(int p[][V], int d[][V])
{
  // cout<<"Hello"<<"\n";
   for(int i=0;i<V;i++)
   {
       for(int j=0;j<V;j++)
       {
```

```
            // cout<<"Hello1"<<"\n";
            if(i!=j)
            {
             cout<<"Shortest path from "<<i+1<<" to "<< j+1<<" => ";
             cout<<"[Total Distance : "<<d[i][j]<<" ( Path : ";
             //cout<<"Hello1"<<"\n";
            int k=j;
            int l=0;
            int a[V];
            a[l++] = j+1;
            while(p[i][k]!=i+1)
            {
                 //cout<<"Hello1"<<"\n";
                a[l++]=p[i][k];
                k=p[i][k]-1;
            }
            a[l]=i+1;
             //cout<<"Hello1"<<"\n";
            for(int r =l;r>0;r--)
            {
                 //cout<<"Hello1"<<"\n";
              cout<<a[r]<<" --> ";
            }
            cout<<a[0]<<" )";
            cout<<endl;

            }
        }
    }
}
 //Floyd Warshall Algorithm
void floydWarshall ()
{
   int dist[V][V], i, j, k;
   int parent2[V][V];
   for (i = 0; i < V; i++)
```

```
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for(int i=0;i<V;i++)
    {
        for(int j=0;j<V;j++)
        {
            parent2[i][j] = parent[i][j];
        }
    }

    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    parent2[i][j] = parent2[k][j];
                }
            }
        }
    }

    printSolution(dist);
    cout<<"\n\n";
    printParents(parent2);
    cout<<"\n\n";
    cout<<"All pair shortest path is given below :"<<"\n";
    print_path(parent2, dist);
}
};
//Main Function
int main()
```

```
{
   Graph<int> g;
   g.matrix_form2();
  // g.make_parent();
//Intializing the graph given in above picture
   g.addEdge(1,2,3);
   g.matrix_form(1,2,3);
   g.addEdge(1,3,8);
   g.matrix_form(1,3,8);
   g.addEdge(1,5,-4);
   g.matrix_form(1,5,-4);
   g.addEdge(2,4,1);
   g.matrix_form(2,4,1);
   g.addEdge(2,5,7);
   g.matrix_form(2,5,7);
   g.addEdge(3,2,4);
   g.matrix_form(3,2,4);
   g.addEdge(4,3,-5);
   g.matrix_form(4,3,-5);
   g.addEdge(4,1,2);
   g.matrix_form(4,1,2);
   g.addEdge(5,4,6);
   g.matrix_form(5,4,6);
   cout<<"Graph in the form of adjecency list representation : "<<"\n";
   g.printAdj();
   cout<<"Graph in the form of matrix representation : "<<"\n";
   g.print_matrix();
   cout<<"\n\n";
   g.floydWarshall();
```

DO IT YOURSELVES

- ❑ Implement a BFS based algorithm to find the shortest route in **Snakes and Laddders Game**
- ❑ Implement a program to classify each of the graph as **forward edge, back edge** or **cross edge.**
- ❑ Find the shortest path in a weighed graph where each egde is 1 or 2.
- ❑ Read about **Hamiltonian Cycle**. Implement an optimisied **Travelling Salesman Problem** solution using Dynamic Programming. [Refer Tutorial]
- ❑ Read about -
  - **Articulation Points**
  - **Bridges**
  - **Eulerian Paths and Circuits**
- ❑ Read about **Strongly Connected Components** and its algorithms -
  - Kosaraju's algorithm
  - Tarjan's algorithm
- ❑ Solve **Holdiay Accomodation(HOLI)** on Spoj [Refer Tutorial]
- ❑ Implement a **Flood Fill Algorithm**, to color various components of a given pattern. Pattern boundary is represented by '#'
- ❑ Implement an algorithm for '**Splitwise App**', to minimize the cash flow between a group of friends.
- ❑ Solve the Graphs Assignment at **HackerBlocks**

12 Combinatorial Game Theory

Introduction :

Combinatorial games are two-person games with perfect information and no chance moves (no randomization like coin toss is involved that can affect the game). These games have a win-or-lose or tie outcome and determined by a set of positions, including an initial position, and the player whose turn it is to move. Play moves from one position to another, with the players usually alternating moves, until a terminal position is reached. A terminal position is one from which no moves are possible. Then one of the players is declared the winner and the other the loser. Or there is a tie (Depending on the rules of the combinatorial game, the game could end up in a tie. The only thing that can be stated about the combinatorial game is that the game should end at some point and should not be stuck in a loop.

Let's see an Example :

Consider a simple game which two players can play. There are N coins in a pile. In each turn, a player can choose to remove one or two coins.

The players keep alternating turns and whoever removes the last coin from the table wins.

If N=1 then?

If N=2 then?

If N=3 then?

If N=4 then?

If N=5 then?

If N=6 then?

Once you start playing this game, you will realise that it isn't much fun. You will soon be able to devise a strategy which will let you win for certain values of N. Eventually you will figure out that if both players play smartly, the winner of the game is decided entirely by the initial value of N.

Strategy :

Continuing in this fashion, we find that the first player is guaranteed a win unless N is a multiple of 3. The winning strategy is to just remove coins to make N a multiple of 3.

Finders Keepers game :

A and B playing a game in which a certain number of coins are placed on a table. Each player picks at least 'a' and atmost 'b' coins in his turn unless there is less than 'a' coins left in which case the player has to pick all those left.

(a) **Finders-Winners :** In this format, the person who picks the last coin wins. Strategy is to reduce opponent to a state containing $(a + b) \times k$ coins which is a loosing state for opponent.

(b) **Keepers-Losers :** In this format, the person who picks last coin loses. Strategy is to reduce opponent to a state containing $(a + b) \times k + x$ coins (x lies between 1 and a, both inclusive) which is a loosing state for opponent.

PROBLEMS

1. **A and B play game of finder-winners with a = 2 and b = 6. If A starts the game and there are 74 coins on the table initially, how many should A pick?**

Sol: If A picks 2, B will be left with 72 and no matter what number B picks, A can always pick (8 – x) and wrap up.

2. **In a game of keepers-losers B started the game when there were N coins on the table. If B is confident of winning the game and a = 3, b = 5, which of the following cannot be the value of N?**

(a) 94 (b) 92 (c) 76 (d) 66

Sol: (d) In keeper-losers, the motto is to give opponent 8k + 1 coins to win for sure. With 66 coins, no matter what B does, he cannot give 65 coins to A.

3. **A and B play Finders-winners with 56 coins, where A plays first anad a = 1, b = 6. what should B pick immediately after A?**

(a) 2 (b) 3 (c) 4 (d) Can't be determined

Sol: (d) A will lose in any case. But the number of coins B picks depends on what A picks.

4. **In a game of Keepers-losers played with 126 coins where A plays first ans a = 3, b = 6, who is the winner?**

Sol: In order to win, A should leave 9k+1, 9k + 2 or 9k + 3 coins on the table, since he can do by picking 6 coins and hence can win the game.

5. **In a interesting version of game B gets to choose the number of coins on the table and A gets to choose the format of the game it will be as well as pick coins first. If B chooses 144 and a = 1, b = 5 which format should A choose in order to win?**

Sol: A should choose Keepers-losers and pick 5 coins from the table, leaving 139 coins for B

Properties of the above Games :

1. The games are sequential. The players take turns one after the other, and there is no passing.
2. The games are impartial. Given a state of the game, the set of available moves does not depend on whether you are player 1 or player.
3. Chess is a partisan game(moves are not same).
4. Both players have perfect information about the game. There is no secrecy involved.
5. The games are guaranteed to end in a finite number of moves.
6. In the end, the player unable to make a move loses. There are no draws. (This is known as a normal game. If on the other hand the last player to move loses, it is called a misère game)

Impartial games can be solved using Sprague-Grundy theorem which reduces every such game to Game of NIM.

Game of NIM

Given a number of piles in which each pile contains some numbers of stones/coins. In each turn, a player can choose only one pile and remove any number of stones (at least one) from that pile. The player who cannot move is considered to lose the game (i.e., one who take the last stone is the winner).

SOLUTION :

Let n_1, n_2, ...n_k, be the sizes of the piles. It is a losing position for the player whose turn it is if and only if n_1 x or n_2 x or ... x or $n_k = 0$.

Nim-Sum : The cumulative XOR value of the number of coins/stones in each piles/heaps at any point of the game is called Nim-Sum at that point.

If both A and B play optimally (i.e. they don't make any mistakes), then the player starting first is guaranteed to win if the Nim-Sum at the beginning of the game is non-zero. Otherwise, if the Nim-Sum evaluates to zero, then player A will lose definitely.

Why does it work?

From the losing positions we can move only to the winning ones :

- if x or of the sizes of the piles is 0 then it will be changed after our move (at least one 1 will be changed to 0, so in that column will be odd number of 1s).
  From the winning positions it is possible to move to at least one losing:
- if xor of the sizes of the piles is not 0 we can change it to 0 by finding the left most column where the number of 1s is odd, changing one of them to 0 and then by changing 0s or 1s on the right side of it to gain even number of 1s in every column.

From a balanced state whatever we do, we always end up in unbalanced state. And from an unbalanced state we can always end up in atleast one balanced state.

Now, the pile size is called the nimber/Grundy number of the state.

Sprague-Grundy Theorem

A game composed of K solvable subgames with Grundy numbers G_1, G_2 ... G_K is winnable iff the Nim game composed of Nim piles with sizes G1, G2 ... GK is winnable.

So, to apply the theorem on arbitrary solvable games, we need to find the Grundy number associated with each game state. But before calculating Grundy Numbers, we need to learn about another term-Mex.

What is Mex Function ?

'**Minimum excludant**' a.k.a 'Mex' of a set of numbers is the smallest non-negative number not present in the set.

Eg S = {1, 2, 3, 5}

Mex(S) = 0

S1 = {0, 1, 3, 4, 5}

Mex(S1) = 2

Calculating Grundy Numbers

1. **The game starts with a pile of n stones, and the player to move may take any positive number of stones. Calculate the Grundy Numbers for this game. The last player to move wins. Which player wins the game?**

```
Grundy(0)=?
Grundy(1)=?
Grundy(n) = Mex (0, 1, 2, ....n-1) = n
int calculateMex(set<int> Set)
{
    int Mex = 0;
    while (Set.find(Mex) != Set.end())
        Mex++;
    return (Mex);
}
// A function to Compute Grundy Number of 'n'
```

```
// Only this function varies according to the game
int calculateGrundy(int n)
{
    if (n == 0)
        return (0);
    set<int> Set; // A Hash Table
    for (int i=1; i<=n; i++)
        Set.insert(calculateGrundy(n-i));
    return (calculateMex(Set));
}
```

2. The game starts with a pile of n stones, and the player to move maytake any positive number of stones upto 3 only. The last player to move wins. Which player wins the game? This game is 1 pile version of Nim.

```
Grundy(0)=?
Grundy(1)=?
Grundy(2)=?
Grundy(3)=?
Grundy(4)=mex(Grundy(1),Grundy(2),Grundy(3))

int calculateMex(set<int> Set)
{
    int Mex = 0;
    while (Set.find(Mex) != Set.end())
        Mex++;
    return (Mex);
}

// A function to Compute Grundy Number of 'n'
// Only this function varies according to the game

int calculateGrundy(int n)
{
    if (n == 0)
```

```
        return (0);
    if (n == 1)
        return (1);
    if (n == 2)
        return (2);
    if (n == 3)
        return (3);

    set<int> Set; // A Hash Table
    for (int i=1; i<=3; i++)
        Set.insert(calculateGrundy(n - i));

    return (calculateMex(Set));
}
```

3. **The game starts with a number- 'n' and the player to move divides the number- 'n' with 2, 3 or 6 a then takes the floor. If the integer becomes 0, it is removed. The last player to move wins. Whi player wins the game?**

```
int calculateGrundy (int n)
{
if (n == 0)
return (0);
set<int> Set; // A Hash Table
Set.insert(calculateGrundy(n/2));
Set.insert(calculateGrundy(n/3));
Set.insert(calculateGrundy(n/6));
return (calculateMex(Set));
}
```

How to apply Sprague Grundy theorem ?

1. Break the composite game into sub-games.
2. Then for each sub-game, calculate the Grundy Number at that position.
3. Then calculate the XOR of all the calculated Grundy Numbers.
4. If the XOR value is non-zero, then the player who is going to make the turn (First Player) will win else he is destined to lose, no matter what.

PROBLEMS

QCJ3

Tom and Hanks play the following game. On a game board having a line of squares labelled from 0,1,2 ... certain number of coins are placed with possibly more than one coin on a single square. In each turn a player can move exactly one coin to any square to the left i.e, if a player wishes to remove a coin from square i, he can then place it in any square which belongs to the set (0,1, ... i-1) . The game ends when all coins are on square 0 and player that makes the last move wins. Given the description of the squares and also assuming that Tom always makes the first move you have tell who wins the game (Assuming Both play Optimally).

(http://www.spoj.com/problems/QCJ3/)

Hint :

Each stone at position P, corresponds to heap of size P in NIM Now, if there are x stones at position n, then n is XORed x times because each stone corresponds to a heap size of n.

Then we use the property of xor operator

```
#include<iostream>
using namespace std;

int main()
{

    int n;
    cin>>n;

   while(n--)
   {

    int s;
    cin>>s;
    long r=0;
    int x;
    for(int i=1;i<=s;i++)
     {
          cin>>x;
        if(x&1)
              r=r^i;
```

```
    }
    cout<<(r==0?"Hanks Wins":"Tom Wins")<<endl;

  }
  return 0;
  }
```

Try it yourself !

M&M Game
http://www.spoj.com/problems/MMMGAME/
Hint : This is a Miser Game.

Game of Stones
https://www.hackerrank.com/challenges/game-of-stones-1/problem

Piles Again
(Variation of Nim, HackerBlocks)

Game Theory-1
(Grundy Numbers, HackerBlocks)

Game Theory-2
(Sprague Grundy Thm, HackerBlocks)

Game Theory-3
(Sprague Grundy Thm, HackerBlocks)

1. http://www.spoj.com/problems/RESN04
2. http://www.spoj.com/problems/GAME3
3. http://www.spoj.com/problems/GAME31
4. http://www.spoj.com/problems/NGM
5. http://www.spoj.com/problems/NGM2
6. http://www.spoj.com/problems/NUMGAME
7. http://www.spoj.com/problems/NIMGAME

8. http://www.spoj.com/problems/MAIN73
9. http://www.spoj.com/problems/MATGAME
10. http://www.spoj.com/problems/MCOINS
11. http://www.spoj.com/problems/REMGAME
12. http://www.spoj.com/problems/TRIOMINO
13. http://www.spoj.com/problems/BOMBER/
14. http://www.codechef.com/problems/CHEFBRO/
15. http://www.codeforces.com/contest/256/problem/C

SELF STUDY NOTES

13 Geometric Algorithms

How to check if two line segments intersect?

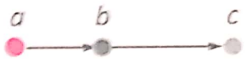
Given two line segments (p1, q1) and (p2, q2), find if the given line segments intersect with each other.

Orientation of an ordered triplet of points in the plane can be

- counterclockwise
- clockwise
- colinear

counterclockwise

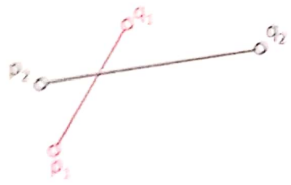
clockwise



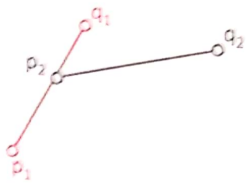
collinear

Two segments (p_1, q_1) and (p_2, q_2) intersect if and only if one of the following two conditions is verified:

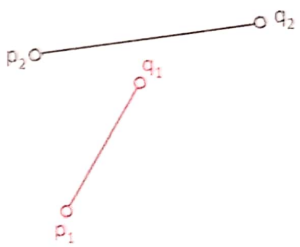
1. (p_1, q_1, p_2) and (p_1, q_1, q_2) have different orientations and (p_2, q_2, p_1) and (p_2, q_2, q_1) have different orientations.



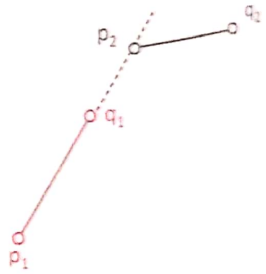
Example 1: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different



Example 2: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different



Example 3: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different



Example 4: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different

2. (p_1, q_1, p_2), (p_1, q_1, q_2), (p_2, q_2, p_1), and (p_2, q_2, q_1) are all collinear and
   - the x-projections of (p_1, q_1) and (p_2, q_2) intersect
   - the y-projections of (p_1, q_1) and (p_2, q_2) intersect

Example 1: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) intersect. The y-projections of (p_1, q_1) and (p_2, q_2) intersect

Example 2: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) do not intersect. The y-projections of (p_1, q_1) and do not (p_2, q_2) intersect

How to compute Orientation?

Use Cross Product :)

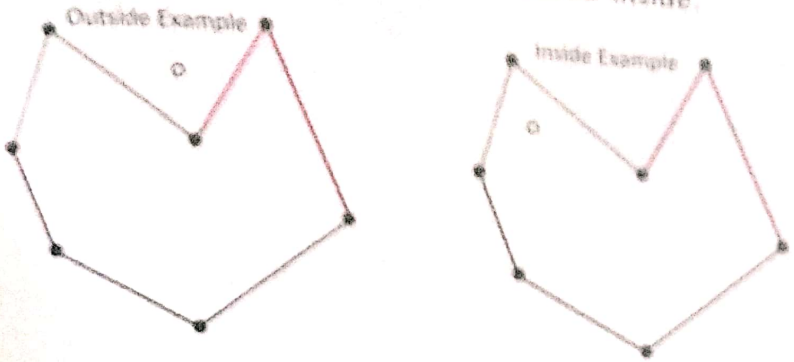
Code:

```
//cross product of OA and OB, this function will tell the orientation of OAB
ll cross(Point o, Point a, Point b) {
    int val = ((a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x));
    if(val == 0)return 0;
    return (val > 0 ? 2 : 1); //1 - clockwise , 2 - anticlockwise
}
struct Point
{
    int x;
    int y;
};
// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;

    return false;
}

bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
```

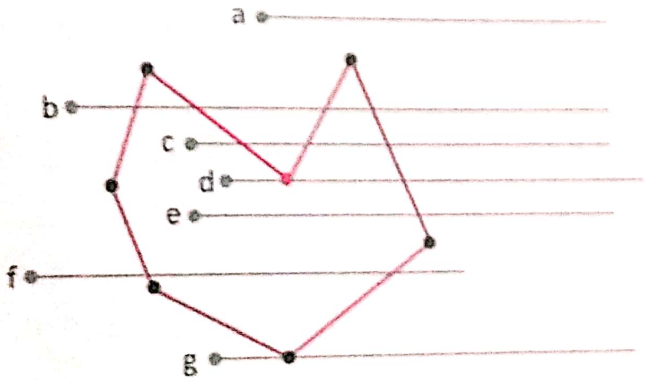
```
    // Find the four orientations needed
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // Case 1
    if (o1 != o2 && o3 != o4)
        return true;
    // Case 2
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and q2 are colinear and q2 lies on segment p1q1

    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2

    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false;
}
```

How to check if a given point lies inside or outside a polygon?

Given a polygon and a point 'p', find if 'p' lies inside the polygon or not. The points lying on the border are considered inside.



1. Draw a horizontal line to the right of each point and extend it to infinity
1. Count the number of times the line intersects with pol ygon edges.
2. A point is inside the polygon if either count of inter sections is odd or point lies on an edge of polygon. If none of the conditions is true, then point lies outside.




```
bool isInside(Point polygon[], int n, Point p)
{
    // Create a point for line segment from p to infinite
    Point extreme = {INF, p.y};
    // Count intersections of the above line with sides of polygon
    int count = 0, i = 0;
    do
    {
        int next = (i+1)%n;
        // Check if the line segment from 'p' to 'extreme' intersects
    // with the line segment from 'polygon[i]' to 'polygon[next]'
    if (doIntersect(polygon[i], polygon[next], p, extreme))
    {
        // If the point 'p' is colinear with line segment 'i->next',
```

```
            // then check if it lies on segment. If it lies, return true,
            // otherwise false
            if (orientation(polygon[i], p, polygon[next]) == 0)
               return onSegment(polygon[i], p, polygon[next]);

            count++;
        }
        i = next;
        } while (i != 0);
        // Return true if count is odd, false otherwise
        return count&1; // Same as (count%2 == 1)
    }
```

Convex Hull :

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.

Jarvis Algorithm :

The idea of Jarvis's Algorithm is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction. The big question is, given a point p as current point, how to find the next point in output?

The idea is to use orientation() here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, r, q) = counterclockwise".

```
void convexHull(Point points[], int n)
{
    // There must be at least 3 points
    if (n < 3) return;
    // Initialize Result
    vector<Point> hull;
    // Find the leftmost point
    int l = 0;
```

```
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;

    // Start from leftmost point, keep moving counterclockwise
    // until reach the start point again.
    int p = l, q;
    do
    {
        // Add current point to result
        hull.push_back(points[p]);

        // Search for a point 'q' such that orientation(p, x, q)
        //is counterclockwise for all points 'x'. The idea
        // is to keep track of last visited most counterclock-
        // wise point in q. If any point 'i' is more counterclock-
        // wise than q, then update q.
        q = (p+1)%n;
        for (int i = 0; i < n; i++)
        {
            // If i is more counterclockwise than current q, then
            // update q
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;
        }

        // Now q is the most counterclockwise with respect to p
        // Set p as q for next iteration, so that q is added to
        // result 'hull'
        p = q;

    } while (p != l); // While we don't come to first point
}
```

Graham Scan :

Let points[0...n-1] be the input array.

1. Find the bottom-most point by comparing y coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Let the bottommost point be P0. Put P0 at first position in output hull.

2. Consider the remaining n-1 points and sort them by polor angle in counterclockwise order around points[0]. If polor angle of two points is same, then put the nearest point first.
3. After sorting, check if two or more points have same angle. If two more points have same angle, then remove all same angle points except the point farthest from P0. Let the size of new array be m.
4. If m is less than 3, return (Convex Hull not possible)
5. Create an empty stack 'S' and push points[0], points[1] and points[2] to S.
6. Process remaining m-3 points one by one. Do following for every point 'points[i]'

   4.1 Keep removing points from stack while orientation of following 3 points is not counterclockwise (or they don't make a left turn).

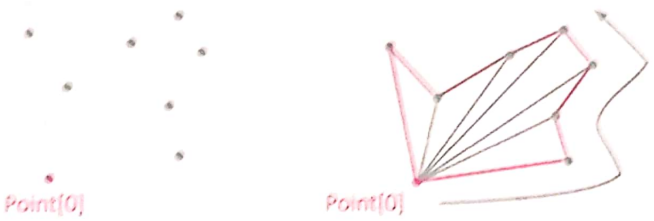
   (a) Point next to top in stack (b) Point at the top of stack

   (c) points[i]

   4.2 Push points[i] to S
7. Print contents of S.

The above algorithm can be divided in two phases :

Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points be sorting them with respect to the bottommost point. Once the points are sorted, they form a simple closed path.



Given Points

Points considered according to increasing angle with respect to point [0] form a simple closed path

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, orientation helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be prev - p, curr - c and next - n. If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it.

```
void convexHull(Point points[], int n)
{
    // Find the bottommost point
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++)
```

```
{
  int y = points[i].y;
  // Pick the bottom-most or chose the left
  // most point in case of tie
  if ((y < ymin) || (ymin == y &&
      points[i].x < points[min].x))
    ymin = points[i].y, min = i;
  }

  // Place the bottom-most point at first position swap(points[0], points[min]);

  // Sort n-1 points with respect to the first point.
  // A point p1 comes before p2 in sorted ouput if p2
  // has larger polar angle (in counterclockwise
  // direction) than p1
  p0 = points[0];
  sort(&points[1], n-1,compare);

  // If two or more points make same angle with p0,
  // Remove all but the one that is farthest from p0
  // Remember that, in above sorting, our criteria was
  // to keep the farthest point at the end when more than
  // one points have same angle.
  int m = 1; // Initialize size of modified array
  for (int i=1; i<n; i++)
  {
      // Keep removing i while angle of i and i+1 is same
      // with respect to p0
      while (i < n-1 && orientation(p0, points[i],points[i+1]) == 0)
      i++;


    points[m] = points[i];
    m++; // Update size of modified array
}
```

```
        // If modified array of points has less than 3 points,
        // convex hull is not possible
        if (m < 3) return;

        // Create an empty stack and push first three points
        // to it.
        stack<Point> S;
        S.push(points[0]);
        S.push(points[1]);
        S.push(points[2]);

        // Process remaining n-3 points
        for (int i = 3; i < m; i++)
        {
            // Keep removing top while the angle formed by
            // points next-to-top, top, and points[i] makes
            // a non-left turn
            while (orientation(nextToTop(S), S.top(), points[i]) != 2)
                S.pop();
            S.push(points[i]);
        }
        // Now stack has the output points, print contents of stack
    }
```

Andrew's monotone chain convex hull algorithm :

It constructs the convex hull of a set of 2-dimensional points.

It does so by first sorting the points lexicographically (first by xcoordinate, and in case of a tie, by y-coordinate), and then constructing upper and lower hulls of the points.

Input: a list P of points in the plane.

Sort the points of P by x-coordinate (in case of a tie, sort by y-coordinate).

Initialize U and L as empty lists.

The lists will hold the vertices of upper and lower hulls respectively.

for i = 1, 2, ..., n:

while L contains at least two points and the sequence of last two points of L and the point P[i] does not make a counter-clockwise turn:

remove the last point from L

append P[i] to L

for i = n, n-1, ..., 1:

while U contains at least two points and the sequence of last two points of U and the point P[i] does not make a counter-clockwise turn:

remove the last point from U

append P[i] to U

Remove the last point of each list (it's the same as the first point of the other list).

Concatenate L and U to obtain the convex hull of P.

Points in the result will be listed in counter-clockwise order.

How will you check whether a given point lies inside a triangle or not??

Hint : Don't give the answer discussed above for polygon.

Problem - KTHCON

A 1-concave polygon is a simple polygon (its sides don't intersect or touch) which has at least 1 concave interior angle.

There are N points on a plane. Let S be the maximum area of a 1-concave polygon with vertices in those points. Compute 2S. Note that if there is no such (1-concave) polygon, you should print -1.

Constraints:

$1 \leq T \leq 100$

$3 \leq N \leq 105$

No two points coincide (have identical both x and y coordinates).

No three points are collinear.

The sum of N over all test cases won't exceed 5·105.

$|x|, |y| \leq 109$

Input :

2

5

2 2

-2 -2

2 -2

```
-2 2
0 1
3
0 0
1 0
0 1
```

Output

```
28
-1
```

Code :

```
ll cross(pair < ll, ll > o, pair < ll, ll >a, pair < ll,
ll > b) {
//cross product of OA and OB
return ((a.first - o.first)*(b.second - o.second) - (a.second - o.second)*(b.first
- o.first));
}

void convex_hull(vector < pair < ll, ll > > &pt) {
    //function to calculate the hull
    if (pt.size() < 3)return;
    sort(pt.begin(), pt.end());
    vector < pair < ll, ll > > up, dn;
    int k = 0;
    for (int i = 0; i < pt.size(); ++i) {
        while (up.size() > 1 && cross(up[up.size() - 2],
        up[up.size() - 1], pt[i]) >= 0)up.pop_back();
        while (dn.size() > 1 && cross(dn[dn.size() - 2],
        dn[dn.size() - 1], pt[i]) <= 0)dn.pop_back();
        up.push_back(pt[i]);
        dn.push_back(pt[i]);
    }
    pt = dn;
    for (int i = up.size() - 2; i >= 1; i--) {
        pt.push_back(up[i]);
    }
}
```

```
ll triangle_area(pair < ll, ll > a, pair < ll, ll > b, pa
ir < ll, ll > c) {
    return abs(cross(a, b, c));
}
int main() {

cin >> t;
while (t--) {
    vector < pair < ll, ll > > outer;
    vector < pair < ll, ll > > inner;
    map < pair < ll, ll >, int > mp;
    cin >> n1;
    for (int i = 0; i < n1; ++i) {
        cin >> a >> b;
        outer.push_back(make_pair(a, b));
        mp[make_pair(a, b)] = 1;
    }
    convex_hull(outer);
    for (int i = 0; i < outer.size(); ++i)mp[outer[i]] = 0;
    for (map < pair < ll, ll >, int > ::iterator it = mp.begin(); it !=
    mp.end(); ++it) {
        if ((it->second) == 1) {
            inner.push_back(it->first);
        }
    }
    convex_hull(inner);
    if (inner.size() == 0) {
        cout << "-1\n";
        continue;
    }
    //now we got the inner and outer HUll
    //calc the outer HUll area
    ll area = 0;
    for (int i = 1; i < outer.size() - 1; ++i) {area += triangle_area(outer[0]
    outer[i], outer[i + 1]);
```

```
    }
    //cout << area << "\n";
    //got the area of convex hull now we have to remove one point from inner hull
    ll minTrianglearea = INT64_MAX;
    int fptr = 0, sptr = 0;
    for (int i = 0; i < outer.size(); ++i) {
        ll currarea = triangle_area(outer[i], outer[(i + 1) % outer.size()],
        inner[sptr]);

        while (true) {
            ll newarea = triangle_area(outer[i], outer[(i + 1) % outer.size()], inner[(sptr
            + 1) % inner.size()]);
            if (newarea < currarea) {
                currarea = newarea;
                sptr = (sptr + 1) % inner.size();
            }
            else {
                break;
            }
        }
        minTrianglearea = min(minTrianglearea, currarea);
    }
    cout << abs(area - minTrianglearea) << "\n";

}

return 0;
}
```

14 Fast Fourier Transformation

Step-1 FFT

$$A(x) = a_0x^0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}$$

Divide it into two polynomials, one with even coefficients(0) and the other with the odd coefficients(1):

$$A_0(x) = a_0x^0 + a_2x^1 + \ldots + a_{n-2}x^{n/2-1}$$

$$A_1(x) = a_1x^0 + a_3x^1 + \ldots + a_{n-1}x^{n/2-1}$$

Finally we can write original polynomial as combination of Ao and A1:

$$A(x) = A_0(x^2) + xA_1(x^2)$$

Now writing th original polynomial A(x) in point form for n-th root of unity:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1$$

$$y_{k+n/2} = A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2}A_1(w_n^{2k+n}) = A_0(w_n^{2k}w_n^n)A_1(w_n^{2k}w_n^n)$$

$$= A(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1$$

Finally our **point form** will be:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1,$$

$$y_{k+n/2} = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1$$

FFT()

```
vector<complex<double> > fft(vector<complex<double> > a){
    int n = (int)a.size();
    if(n <= 1)
        return a;
    //Dividing a0 and a1 as even and odd polynomial of degree n/2
    vector<complex<double> > a0(n/2), a1(n/2);
    for(int i = 0;i<n/2;i++){
        a0[i] = a[2*i];
        a1[i] = a[2*i + 1];
    }
```

```
    //Divide step
    //Recursively calling FFT on polynoimial of degree n/2
    a0 = fft(a0);
    a1 = fft(a1);
    double ang = 2*PI/n;
    //defining w1 and wn
    complex<double> w(1) , wn(cos(ang),sin(ang));
    for(int i = 0;i<n/2;i++){
        //for powers of k<= n/2
        a[i] = a0[i] + w*a1[i];
        //powers of k > n/2
        a[i + n/2] = a0[i] - w*a1[i];
        //Updating value of wk
        w *= wn;
    }
    return a;
}
```

Time Complexity: O(nlogn)

Step 2: Convolution

Multiply()

```
void multiply(vector<int> a, vector<int> b){
    vector<complex<double> > fa(a.begin(),a.end()), fb(b.begin(),b.end());
    int n = 1;
    //resizing value of n as power of 2
    while(n < max(a.size(),b.size()))
        n <<= 1;
    n <<= 1;
    //cout<<n<<endl;
    fa.resize(n);
    fb.resize(n);
    //Calling FFT on polynomial A and B
    //fa and fb denotes the point form
    fa = fft(fa);
    fb = fft(fb);
```

```
    //Convolution step
    for(int i = 0;i<n;i++){
        fa[i] = fa[i] * fb[i];
    }
    //Converitng fa from coefficient back to point form
    fa = inv_fft(fa);
    return;
}
```

Time Complexity: O(n) (exluding the time for calculating FFT())

Step 3: Inverse FFT

So, suppose we are given a vector $(y_0, y_1, y_2, \ldots y_{n-1})$.Now we need to restore it back to coefficient form.

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \vdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Then the vector $\{a_0, a_1, a_2, \ldots a_{n-1}\}$can be found by multiplying the vector $\{y_0, y_1, y_2, \ldots y_{n-1}\}$ by an inverse matrix:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \vdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Thus we get the formula:

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Comparing it with the formula for y_k:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj},$$

Inverse_FFT()

```
vector<complex<double> > inv_fft(vector<complex<double> > y){
    int n = y.size();
    if(n <= 1)
       return y;
    vector<complex<double> > y0(n/2), y1(n/2);
    for(int i = 0;i<n/2;i++){
       y0[i] = y[2*i];
       y1[i] = y[2*i + 1];
    }
    y0 = inv_fft(y0);
    y1 = inv_fft(y1);
    double ang = 2 * PI/n * -1;
    complex<double> w(1), wn(cos(ang), sin(ang));
    for(int i = 0;i<n/2;i++){
       y[i] = y0[i] + w*y1[i];
       y[i + n/2] = y0[i] - w*y1[i];
       y[i] /= 2;
       y[i + n/2] /= 2;
       w *= wn;
    }
    //each element of the result is divided by 2
    //assuming that the division by 2 will take place at each level of recursion
    //then eventually just turns out that all the elements are divided into n.

    return y;
}
```

Time Complexity: O(nlogn)

Ques: Very Fast Multiplicaiton

http://www.spoj.com/problems/VFMUL/

```
#include<iostream>
#include<vector>
#include<complex>
#include<algorithm>

using namespace std;
```

```
#define PI 3.14159265358979323846
vector<complex<double> > fft(vector<complex<double> > a){
    //for(int i = 0;i<a.size();i++)cout<<a[i]<<" ";cout<<endl;
    int n = (int)a.size();
    if(n <= 1)
        return a;
    vector<complex<double> > a0(n/2), a1(n/2);
    for(int i = 0;i<n/2;i++){
        a0[i] = a[2*i];
        a1[i] = a[2*i + 1];
        //cout<<n<<" "<<a0[i]<<" "<<a1[i]<<endl;
    }
    a0 = fft(a0);
    a1 = fft(a1);
    double ang = 2*PI/n;
    complex<double> w(1) , wn(cos(ang),sin(ang));
    for(int i = 0;i<n/2;i++){
        a[i] = a0[i] + w*a1[i];
        a[i + n/2] = a0[i] - w*a1[i];
        w *= wn;
        //cout<<a[i]<<" "<<a[i+n/2]<<endl;
    }
    return a;
}

vector<complex<double> > inv_fft(vector<complex<double>>y){
    int n = y.size();
    if(n <= 1)
        return y;
    vector<complex<double> > y0(n/2), y1(n/2);
    for(int i = 0;i<n/2;i++){
        y0[i] = y[2*i];
        y1[i] = y[2*i + 1];
    }
    y0 = inv_fft(y0);
```

```
    y1 = inv_fft(y1);
    double ang = 2 * PI/n * -1;
    complex<double> w(1), wn(cos(ang), sin(ang));
    for(int i = 0;i<n/2;i++){
        y[i] = y0[i] + w*y1[i];
        y[i + n/2] = y0[i] - w*y1[i];
        y[i] /= 2;
        y[i + n/2] /= 2;
        w *= wn;
    }
    return y;
}
void multiply(vector<int> a, vector<int> b){
    vector<complex<double> > fa(a.begin(),a.end()), fb(b.begin(),b.end());
    int n = 1;
    while(n < max(a.size(),b.size()))
        n <<= 1;
    n <<= 1;
    //cout<<n<<endl;
    fa.resize(n);
    fb.resize(n);
    fa = fft(fa);
    fb = fft(fb);
    for(int i = 0;i<n;i++){
        fa[i] = fa[i] * fb[i];
        //cout<<fa[i]<<endl;
    }
    fa = inv_fft(fa);
    vector<int> res(n);
    for(int i = 0;i<n;i++){
        res[i] = int(fa[i].real() + 0.5);
        //cout<<res[i]<<endl;
    }
    int carry = 0;
    for(int i = 0;i<n;i++){
```

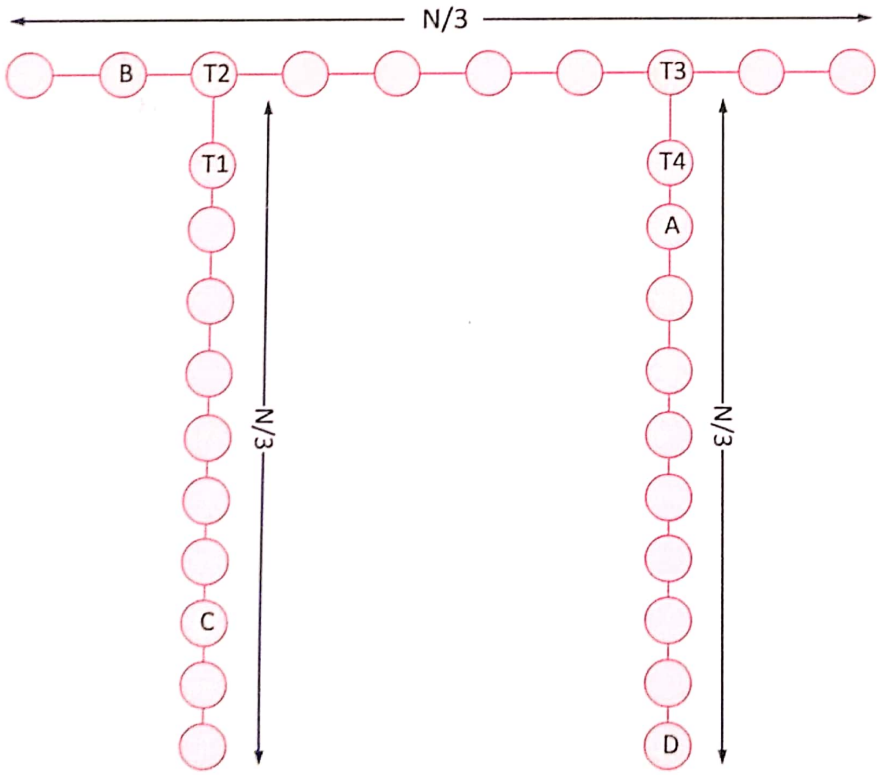
```
        res[i] = res[i] + carry;
        carry = res[i] / 10;
        res[i] %= 10;
    }
    bool flag = 0;
    for(int i = n-1;i>=0;i--){if(res[i] || flag){printf("%d",res[i]);flag = 1;}}
        if(!flag)printf("0");
        printf("\n");
        return;
}
int main(){
    int n;
    scanf("%d",&n);
    while(n--){
        string x,y;
        vector<int> a,b;
        cin>>x>>y;
        for(int i = 0;i<x.length();i++){
            a.push_back(int(x[i] - '0'));
        }
        for(int i = 0;i<y.length();i++){
            b.push_back(int(y[i] - '0'));
        }
        reverse(a.begin(),a.end());
        reverse(b.begin(),b.end());
        multiply(a,b);
    }
    return 0;
}
```

15 Heavy Light Decomposition

Heavy Light Decomposition

- **Balanced Binary Tree** is good, need to visit at most **2log N** nodes to reach from any node to any other node in the tree.
- If a balanced binary tree with N nodes is given, then many queries can be done with O(log N) complexity. **Distance of a path, Maximum/Minimum in a path, Maximum contiguous sum** etc etc.
- **Chains** are also good. We can do many operations on array of elements with O(log N) complexity using **segment tree / BIT**.
- **Unbalanced Tree** is bad

Unbalanced Trees :



Travelling from one node to other requires **O(n)** time.