Algorithm Analysis and Design Project

Algorithms and interesting examples where we can use these algorithms

Name : Atharva Joshi
Roll : 2020111010
Group : B

Index

- Greedy
  - The greedy surfer problem
  - The very basic chore problem
  - The greedy investor problem
  - The Huffman encoding

- Dynamic programming
  - The dynamic surfer problem
  - The teacher and the students
  - The dynamic investor

- Divide and conquer
  - Advanced master theorem for divide and conquer recurrences
  - Algorithm to multiple 2 d-degree polynomials
  - Mergesort
  - Quicksort

- Graphs
  - Depth-First search
  - The military region problem
  - Breadth-First Search
  - The military tower problem
  - Topological sorting
  - The postman problem
  - Disjoint set Union (required for Kruskal’s)
  - The island problem
  - Kruskal’s algorithm
  - Ford-Fulkerson Algorithm for Maximum Flow Problem

- **Sorting**
    - **Selection Sort**
    - **Bubble sort**
    - **Insertion sort**

- **Number theory**
    - **Euclidean algorithm for GCD**
    - **Extended Euclid’s algorithm**
    - **Modular division**

The greedy algorithm

Almost all of us have heard the story of the greedy little boy who tries to take out a lot of candies from a jar if you haven't heard it here is a short version of it
So, a boy visits his aunt and his aunt gives him a jar full of candies and ask him to take out as many candies as he wants in a single turn, the greedy boy tries to take out so many candies that his fist won't come out of the jar, so ultimately he had to settle for a lesser amount of candies but at least he got something, which is better than getting nothing and a hand stuck in a jar.

Now, what if his aunt would have allowed him to take out the candies in 2, 3, or say N turns, obviously the greedy little boy being greedy wants to maximize the number of candies he can take out. The solution to his problem is very simple and logical and the boy takes out as many candies as he can in each turn. It is so simple that we don't even need to prove how it works because any other method would just yield candies less than this method.

Thus as we can see here the little boy, formed **an optimized step**, i.e. taking out as many candies as he can in a single turn, and then **repeating the step** he got the optimized number of candies.

This is how greedy algorithms work -

1) **Can we know the first step of the solution --(Greedy choice property)**
2) **If we take the first step can we rephrase the rest of the problem as a smaller version of the original problem --- (Optimum Substructure Property)**
3) **But we need to show that the first step we take is the optimum solution**

Why is it called greedy algorithm?
At each step we are trying to obtain the optimized output, we are being greedy at every turn.

Here are some examples which can help us understand how the greedy algorithm works.

The Greedy Surfer Problem

(based on activity selection)

A surfer went through the weather reports and now have the exact idea of at what time each wave would come, since the surfer is very greedy, he wants to surf at the maximum number of waves he can, each wave lasts for a bit of time, the surfer is provided with the start time of each wave and the finish time of each wave. Now the surfer uses a greedy approach to calculate the order in which he should surf these waves such that he can surf the maximum number of waves.

We are given n, the number of waves, then $s_1, s_2, s_3 \ldots\ldots s_n$ the start time of each wave and, the time each wave would last $t_1, t_2, t_3 \ldots\ldots t_n$.

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

Solution
We first calculate the finish time of each surf, then we choose the surf with the least finish time, then after surfing that wave, we choose the wave with the least finish time which hasn't started yet and continue doing so.

Here our **Greedy choice** - was to pick the wave with the least finish time
And then we keep repeating this step after surfing a wave.

Why does this work?
Suppose the wave that we choose is different from the wave with the least finish time, then the maximum set of waves we can surf after this can also be surfed if we only surf the wave with the least finish time, thus the maximum answer we would get with this wave can also be achieved with the wave with least finish time, but the opposite is not true thus any other optimal solution can be achieved

if we choose wave with least finish time, but the optimal solution we get with the waves with least finish time cannot be achieved in other cases thus it is optimal.

In this case, the finish time of the waves was, 6, 5, 8, 14, 6, 6, 6
Thus the surfer can surf wave 2 first, then wave 3 and then wave 4, thus the surfer can ride at most 3 waves.

Given below is an algorithm, which when given according to input will tell the surfer about how many waves he can surf at most and will provide him with the set of those waves.

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

struct wave{
    int start_time ;
    int time_taken ;
    int finish_time ;
    int index ;
};

bool my_cmp(struct wave wave1, struct wave wave2){
    return wave1.finish_time < wave2.finish_time ;
}

int main(){
    int n ;
    cin>>n ;
    struct wave waves[n];
    for(int i=0;i<n;i++){
        cin>>waves[i].start_time;
        waves[i].index = i + 1 ;
    }
    for(int i=0;i<n;i++){
        cin>>waves[i].time_taken;
        waves[i].finish_time = waves[i].start_time +
waves[i].time_taken ;
    }
    sort(waves,waves+n,my_cmp);
    //sorts the waves based on finish time
    int current_time = 0 ;
    // stores current time
    int num_waves_surfed = 0  ;
    // stores number of waves surfed
    int set_waves_surfed[n];
    // stores the indices of waves surfer can surf
    for(int i=0;i<n;i++){
        if(waves[i].start_time >= current_time)
        {
            current_time = waves[i].finish_time ;
            set_waves_surfed[num_waves_surfed]=waves[i].index;
            num_waves_surfed++;
        }
    }
    cout<<num_waves_surfed<<endl;
    for(int i=0;i<num_waves_surfed;i++){
        cout<<set_waves_surfed[i]<<' ';
    }
    cout<<endl;
    return 0;
}
```

The Very Basic Chore Problem:

Question

Suppose Bob has certain chores to do in a day but he also has college classes, now bob's mother wants him to perform as many chores as he can in a single day, assuming that all chores take the same amount of time and provided the timing of the classes, Bob wants to find the maximum number of chores he can perform. (we cannot break the chore into 2 parts or more, once started the chore must be completed)

Say the day is divided into 48 parts of half-hour each, we are given the number of classes, the time taken for each chore in minutes) and the timings of each class.

For eg

```
4   35  thus we have 4 classes and, a single chore takes 35 minutes
5    7            timing of the first class is from 2:30 to 3:30, that is 1 hour
9   12            timing of the second class is from 4:30 to 6, that is 1.5 hours
15  21 timing of the third class is from 7:30 to 10:30, that is 3 hours
22  23 timing of the fourth class is from 11 to 11:30, that is a half an hour
```

Solution

Here we can use the greedy method, we first calculate free time intervals that we get and try to perform as many chores as we can in each one of them, thus here

1.) The optimized step is to perform as many chores as we can in the single free time interval
2.) And, then we repeat this step for each free time interval

Why does this work?

Each free time interval is independent of any other time interval as we cant divide a chore into multiple free time intervals, thus optimizing each time interval will yield us the best results.

Now we can see that the intervals of free time that we get here are,

00:00 ---> 2:30, 3:30--->4:30,6:00--->7:30, 10:30--->11, 11:30--->24:00

Thus the free time(in minutes) that we get in intervals is

150, 60, 90, 30, 750

Thus maximum amount of chores that can be done in each interval

$\lfloor 150/35 \rfloor$, $\lfloor 60/35 \rfloor$, $\lfloor 90/35 \rfloor$, $\lfloor 30/35 \rfloor$, $\lfloor 750/35 \rfloor$

That is

4, 1, 2, 0, 21
Thus there sum is 28
Thus here is an algorithm which will help Bob in finding the maximum number of chores he can perform in a day

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

struct _class{
    int start_time ;
    int finish_time ;
};

int main(){
    int num_class;
    int chore_time;
    cin>>num_class>>chore_time;
    struct _class classes[num_class];
    for(int i=0;i<num_class;i++){
        cin>>classes[i].start_time>>classes[i].finish_time ;
    }
    int free_time_start = 0 ;
    int num_chore = 0 ;
    int free_time ;
    for(int i=0 ; i<num_class ;i++){
        free_time = classes[i].start_time - free_time_start ;
        // we will be free till next class starts
        num_chore += free_time*30/chore_time ;
        free_time_start = classes[i].finish_time ;
        // we will be free after current class finishes
    }
    free_time = 48 - free_time_start ;
    // as we are free after the last class
    num_chore += free_time*30/chore_time ;
    cout<<num_chore<<endl;

}
```

The Greedy Investor Problem

Suppose there is an investor, and somehow he receives such precise inside news that he can calculate the value of each stock at each minute interval, say the market remains open for investing for n minutes, the investor wants to maximize the amount of money he makes in a day,
We are given two integers n and m, n being the time in minutes for which the market remains open, m being the number of stocks.
Next, each m line consists of n number which is the percentage stock of the value of the stock at each minute from 1 to n.

For eg:

```
10 4
0.5    0.4    1      2     0.6    0.7   -0.8    -1     4      5
1     -0.2    0.8   -0.5   1.1    2.4    -3     -2     1.6    2.3
0       1.5    0      1.2   -2     0.2   4      -5     1.2    5
-1       -2    -3     -4     1       7     10     -5     1.2    0
```

Solution
The solution is again as basic as the last question we simply chose the best trade for each minute thus we maximize our profit for each minute, in case all trades are negative then the investor doesn't simply invest any money.
So in this particular example, we choose the following stocks for each minute
2,3,1,1,2,4,4,0,1,1

Thus here the optimal step: is to choose the best stock,
And then we repeat it n times.

Why does this work
It works because if the investor would have had his money invested in some other stock than the most yielding one then he would have made less profit than he could, thus it is optimal to simply choose the best stock for each minute.

Here is an algorithm which when provided with proper input will tell the investor about which stocks to invest in, in each minute

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

int main(){
    int n,m ;
    cin>>n>>m ;
    float stocks[n][m] ;
    float max_profit_percent[n] ;
    int max_profit_percent_index[n] ;
    // stores max profit percent for each minute
    for(int i=0;i<n;i++){
        max_profit_percent[i]=0;
        max_profit_percent_index[i]=0;
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            cin>>stocks[i][j] ;
            if(max_profit_percent[j]<stocks[i][j]){
                max_profit_percent_index[j] = i+1 ;
                max_profit_percent[j] = stocks[i][j] ;
            }
        }
    }

    for(int i=0;i<n;i++){
        cout<<max_profit_percent_index[i]<<' ';
    }
    cout<<endl;

}
```

The Huffmann encoding

It is used to find the most economical way in which a string consisting of various letters can be encoded in binary.

Example

Suppose we have a string consisting of only A, B, C, and D, with the following frequency A - 70 million B - 3 million C - 20 million D - 37 million

Total - 130 million

Variable length encoding

- based on prefix free property (no codeword can be a profix of another codeword)
- Now we encode the most frequent charachter to smallest encoding
- Thus let A be 0
- Let D be 11
- Let B be 100
- And C be 101
- This will need 213 million bits
- We can do better

Rephrasing the problem

Given the frequency f1, f2, ..., fn of n symbols, we want a tree whose leaves each correspond to a symbol and which minimizes the overall length of the encoding

Greedy method (Huffman encoding)

- Since the two least used symbols would need the least use of their encoding, we can use the longest encoding for these, therefore these two symbols would be at the lowest depth

- The frequency of any internal node is the sum of the frequency of its descendant leaves
- The cost of encoding is the sum of the frequency of all leaves and nodes except the root

Procedure

- First we delete the 2 symbols with least frequency , say f0 and f1
- Now we can add am imaginary symbols with frequnce f0 + f1
- Now we can repeat the processs for these n - 1 symbols

Analysis

- Here the activity we chose for "Greedy choice property" was the activity where we choose two least frequent symbols are replace them with a single symbol with a frequency equal to the sum of their frequency
- After the first step the problem which we earlier needed to solve for n symbols, now we only have to solve it for (n-1) symbols, we keep repeating it till we only get 1 symbol thus we get our answer

Dynamic Programming

We all have used google maps or any other navigation application to find the fastest route to our destination, such applications use dynamic programming to calculate the fastest route or the shortest route as per our choice.

Dynamic programming is a recursive solution with some customization. whenever we encounter a recursive solution that demands a similar sort of input at such places we can use dynamic programming, we simply store the results of a smaller subproblem so that we won't need to calculate it again.

For eg, to calculate every new term in the Fibonacci series we would need the last two terms, so instead of calculating the last two terms we just store them so we won't need to calculate them again.

```
fibo[0] = 1
fibo[1] = 1
For i = 2.....n
fibo[i] = fibo[i-1] + fibo[i-2]
```

Dynamic programming have 2 main properties

1) Overlapping subproblem property
2) Optimal substructure property

Overlapping subproblem property

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

Optimal substructure property

A given problem has Optimal Substructure Property if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
For example, the Shortest Path problem has the following optimal substructure property:
If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is a combination of the shortest path from u to x and shortest path from x to v.

The dynamic surfer problem

The same surfer we encountered in the first example, have now realized that riding as many waves as he can, won't yield him the maximum fun, rather there are certain waves which are more fun to surf than others, so this time he decides to allot certain fun points to each wave and now he wants to maximize the total fun points instead of the number of waves he surf

We are provided with n, the number of waves
then $s_1, s_2, s_3 \ldots\ldots s_n$ the start time of each wave and, the time each wave would last $t_1, t_2, t_3 \ldots\ldots t_n$.
then $f_1, f_2, f_3 \ldots\ldots f_n$ the fun points corresponding to each wave.

Now the surfer wants to know the maximum fun points he can collect.

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

Solution
We first sort the waves based on their finish time.
Then we declare an array dp[n], which would store the maximum fun points if we only consider the first i waves,
So dp[0] = fun points of wave 1 , because we only consider the first wave
Now for dp[i] we try to find the wave with closest finish time just less than the start time of i^{th} index wave, say it was the wave at x^{th} index, now dp[i] = f[i] + dp[x] or dp[i] = dp[i-1] whichever is higher, thus `dp[i] = max(dp[x]+f[i], dp[i-1])`

Why do we choose the index with finish time closest to start time of current wave?
There are two conditions that we need to meet for choosing x,

1) Its finish time must be less than the start time of the current wave
2) dp[x] must be maximum, among all waves whose finish time is less than the start time of the current stock

Now dp[x] is maximum, as every succeeding wave in the dp[] array has a value either equal to or greater than its predecessor and since x is the last wave which meets criteria 1 thus has the maximum value of dp[].

Why does this work ??

The maximum funpoints we can achieve till i^{th} index can be either if we surf the wave at i^{th} index or we don't if we don't surf it then dp[i] = dp[i-1] and if we surf it, it then the funpoints at maximum can be the sum of funpoints of the wave at i^{th} index and the maximum funpoints we can achieve before the start time of i^{th} wave, which is dp[x] , where x is the wave with finish time closest to the start time of i^{th} wave.
Thus whichever is higher dp[i-1] or dp[x] + f[i] we choose that as dp[i] .

Overlapping subproblem property in this case
Here we divided the problem of maximising the funpoints of n waves to the subproblem of maximising the funpoints which can be achieved using i number of waves with the least finish time.

Optimal substructure property
Here the optimal solution for dp[i] is obtained using dp[x] or dp[i-1] which are the optimal solutions for according substructures.

Here is an algorithm the surfer can use for knowing the maximum funpoints he can achieve

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

struct wave
{
    int start_time ;
    int finish_time ;
    int time_taken ;
    int funpoints ;
};

bool cmp(struct wave wave1,struct wave wave2)
{
    return wave1.finish_time<=wave2.finish_time;
}

struct wave* waves;


int bs(int pos)
{
    int l=0,h=pos-1,idx=-1;

    while(l<=h)
    {
        int mid=(l+h)/2;

        if(waves[mid].finish_time<=waves[pos].start_time)
        {
            idx=mid;
            l=mid+1;
        }
        else h=mid-1;
    }
    return idx;
}

int main()
{
    int i,j;
    int n;
    cin>>n;
    waves = (struct wave*)malloc(sizeof(wave)*n);
    for(i=0;i<n;i++)
    {
        cin>>waves[i].start_time;
    }
    for(i=0;i<n;i++)
    {
        int t;
        cin>>t;
        waves[i].finish_time = waves[i].start_time + t ;
    }
    for(i=0;i<n;i++)
    {
        cin>>waves[i].funpoints;
    }

    sort(waves,waves+n,cmp);
    int dp[n];
    dp[0]=waves[0].funpoints;
    for(i=1;i<n;i++)
    {
        int ans=waves[i].funpoints;
        int idx=bs(i);
        // uses binary search to find wave with finish time
closest
        // to starting point of current wave
        if(idx!=-1)
        {
            ans+=dp[idx];
        }
        dp[i]=max(dp[i-1],ans);

    }
    cout<<dp[n-1]<<endl;
    return 0;
}
```

The teacher and the students

A primary school teacher wants her students to interact with each other, for this purpose she wants to pair them in pairs of 2 with different students each day, the teacher decides that a student being paired up is not necessary but whenever a student is paired up , it must be with a different student, to encourage maximum interaction. She wants to find out, for how many days she can keep up with this condition , before repeating a configuration.

Solution

```
f(n) = ways n students can remain single
       or pair up.

For n-th studentthere are two choices:
1) n-th student remains single, we recur
   for f(n - 1)
2) n-th student pairs up with any of the
   remaining n - 1 students. We get (n - 1) * f(n - 2)

Therefore we can recursively write f(n) as:
f(n) = f(n - 1) + (n - 1) * f(n - 2)
```

Why does this work??

The teacher first takes 1 student as a subset, thus there is only 1 day the students can be arranged in a unique configuration, then adding another student can be done in the following manner,

The new student is not paired

The new student is paired with any of the (n-1) initial students, and the remaining (n-2) students are arranged in their configurations

Thus `f(n) = f(n - 1) + (n - 1) * f(n - 2)`

Overlapping subproblem property in this case

Here we divided the problem of finding the number of days for n students, into subproblems of finding the number of days for (n-1) and (n-2) students and divided this further

Optimal substructure property
Here the optimal solution for dp[i] is obtained using dp[i-1] and dp[i-1] which are the optimal solutions for according substructures.

Here is an algorithm than the teacher can use to know the maximum number of days for which we can always have a unique configuration.

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

int countFriendsPairings(int n)
{
    int dp[n + 1];
    for (int i = 0; i <= n; i++) {
        if (i <= 2)
            dp[i] = i;
        else
            dp[i] = dp[i - 1] + (i - 1) * dp[i - 2];
    }
    return dp[n];
}

int main()
{
    int n ;
    cin>>n;
    cout << countFriendsPairings(n) << endl;
    return 0;
}
```

The Dynamic Investor

The same investor we encountered for greedy examples is now getting some inside news which is not so precise, he gets to know about certain stocks and the percentage change in their value over a certain period of time, The investor cannot reinvest his profits.
Given the start time, finish time, and percentage gains over this time period for some stocks, the investor wants to find the maximum profit (in percentage) he can make. For this purpose, he uses a dynamic programming approach.

→ At A single time he can only invest in a single stock
→ He only considers stocks with positive profit percentage

For eg

```
10
1    12   5    7    6   3    12  2    4    9
3    16   10   8    9   10   13  5    9    12
0.5  1.1  1.5  0.6  2   0.3  4   1.2  0.1  0.3
```

Solution
We use dynamic programming for this purpose, we first sort the stocks based on their finish time, then we choose to make an array dp[n], in which we store the maximum profit we can achieve if we only consider the first i stocks.
So dp[0] = profit of stock with least finish time

For dp[i]
→ either we invest in that stock, or we don't invest in that stock
→ if we don't invest in that stock then dp[i] = dp[i-1] as it is the same as not considering that stock
→ if we invest in that stock, then we find the index of the stock with finish time less than but closest to the starting time of our current stock, say x, then
dp[i] = dp[x] + profit[i]

```
dp[i] = max(dp[x]+f[i], dp[i-1])
```

Why do we choose the index with finish time closest to start time of current stock?
There are two conditions that we need to meet for choosing x,

1) Its finish time must be less than the start time of current stock
2) dp[x] must be maximum, among all stocks whose finish time is less than the start time of the current stock

Now dp[x] is maximum, as every succeeding stock in the dp[] array has a value either equal to or greater than its predecessor, thus dp[x] is the last stock which meets criteria 1 thus has the maximum value of dp[].

Overlapping subproblem property in this case
Here we divided the problem of maximising the profit of n stocks to the subproblem of maximising the profit which can be achieved using i number of stocks with the least finish time.

Optimal substructure property
Here the optimal solution for dp[i] is obtained using dp[x] or dp[i-1] which are the optimal solutions for according substructures.

The investor can use this dynamic programming algorithm for this purpose

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

struct stock
{
    int start_time ;
    int finish_time ;
    float profit ;
};

bool cmp(struct stock stock1,struct stock stock2)
{
    return stock1.finish_time<=stock2.finish_time;
}
```

```
struct stock* stocks;


int bs(int pos)
{
    int l=0,h=pos-1,idx=-1;

    while(l<=h)
    {
        int mid=(l+h)/2;

        if(stocks[mid].finish_time<=stocks[pos].start_time)
        {
            idx=mid;
            l=mid+1;
        }
        else h=mid-1;
    }
    return idx;
}

int main()
{
    int i,j;
    int n;
    cin>>n;
    stocks = (struct stock*)malloc(sizeof(stock)*n);
    for(i=0;i<n;i++)
    {
        cin>>stocks[i].start_time;
    }
    for(i=0;i<n;i++)
    {
        cin>>stocks[i].finish_time ;
    }
    for(i=0;i<n;i++)
    {
        cin>>stocks[i].profit;
    }
    sort(stocks,stocks+n,cmp);
    int dp[n];
    dp[0]=stocks[0].profit;
    for(i=1;i<n;i++)
    {
        int ans=stocks[i].profit;
        int idx=bs(i);
        // uses binary search to find stock with finish time
closest
        // to starting point of current stock
        if(idx!=-1)
        {
            ans+=dp[idx];
        }
        dp[i]=max(dp[i-1],ans);

    }
    cout<<dp[n-1]<<endl;
    return 0;
}
```

Divide And Conquer

If someone asks us to divide a cake in 16 equal parts its unlikely that we will start dividing it by cutting single parts of of 1/16th volume, instead most of us will cut it into 2 halves, then each part into 2 equal parts and so on , because dividing it into 2 equal halves is easier than dividing it a single part of 1/16th volume.

There are certain problems in life, where solving it as a whole is difficult as compared to dividing it into subproblems that are easier to solve, at such places we can use divide and conquer algorithms.

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using the following three steps.

1. **Divide**: Break the given problem into subproblems of the same type.
2. **Conquer**: Recursively solve these subproblems
3. **Combine**: Appropriately combine the answers

The most common example of divide and conquer algorithms which we encounter is the mergesort algorithm, where

1) Divide - We first divide the arrays into smaller parts
2) Conquer- We sort these smaller parts
3) Combine - We combine these sorted arrays to get our desired sorted array.

Binary search is another very common algorithm that is based on this principle.

Advanced master theorem for divide and conquer recurrences

Master Theorem is used to determine the running time of algorithms (divide and conquer algorithms) in terms of asymptotic notations.
Consider a problem that can be solved using recursion.

```
function f(input x size n)
if(n < k)
solve x directly and return
else
divide x into $a$ subproblems of size n/b
call f recursively to solve each subproblem
Combine the results of all sub-problems
```

The above algorithm divides the problem into a subproblems, each of size n/b and solve them recursively to compute the problem and the extra work done for the problem is given by f(n), i.e., the time to create the subproblems and combine their results in the above procedure.
So, according to the master theorem, the runtime of the above algorithm can be expressed as:

```
    T(n) = aT(n/b) + f(n)
```

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

Not all recurrence relations can be solved with the use of the master theorem i.e. if

T(n) is not monotone, ex: T(n) = sin n
f(n) is not a polynomial, ex: T(n) = 2T(n/2) + 2n
This theorem is an advance version of master theorem that can be used to

determine running time of divide and conquer algorithms if the recurrence is of the following form :-

$$T(n) = at(n/b) + (\theta n^{k} log^{p} n)$$

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
b > 1, k >= 0 and p is a real number.

Then,

1) $if\ a > b^{k},\ then\ T(n) = \theta(n^{log_{b}a})$

2) $if\ a = b^{k}$

   a) $if\ p > -1,\ then\ T(n) = \theta(n^{log_{b}a}\ log^{p+1} n)$

   b) $if\ p = -1,\ then\ T(n) = \theta(n^{log_{b}a}\ loglogn)$

   c) $if\ p < -1,\ then\ T(n) = \theta(n^{log_{b}a})$

3) $if\ a < b^{k}$

   a) $if\ p >= 0,\ then\ T(n) = \theta(n^{k}\ log^{p} n)$

   b) $if\ p < 0,\ then\ T(n) = \theta(n^{k})$

Given 2 d-degree polynomials, algorithm to multiply them

Naive method

- According to the definition
- time complexity = O(d^2)
- We can do better

First cut solution

- for n points , compute values for polynomial A and B
- from these values compute values of C = A.B , for each n values
- Obtain polynomial C from these values(n>2+1)
- time complexity - O(d^2)
- not efficient

Fast-Fourier Transformation

- Instead of computing the value of the polynomial at n points, we divide the polynomial in two parts , one consisting of even power the other one consisiting of odd
- Then we calculate the value of these two polynomial at n/2 points
- first let the n points were (+)(-)x_0,(+)(-)x_1.....(+)(-)$x_{n/2 - 1}$, thwn the next n/2 points are x_{02},x_{12}...$x_{n/2 - 12}$
- T(n) = 2*T(n/2) + O(n)

Problem with doing this more than 1 time

- In the first step we have pairs of positive and negative values of the same magnitude for which we were calculating the values
- after the second step since we only have squares they are all positive
- Thus we cannot divide these values in half
- So instead we use the imaginary number, we take nth roots of unity as the n numbers

FFT Algorithm

```
function FFT(A,w)
```

```
    if w = 1: return A(1)
    express A(x) in the form Ae(x^2)+xAo(x^2)
    call FFT(Ae,w^2) to evaluate Ae at even powers of w
    call FFT(Ao,w^2) to evaluate Ao at even powers of w
    for j=0 to n-1:
        compute A(w^j) = Ae(w^2j) + w^j*Ao(w^2j)

    return A(w^0),....A(w^n-1)
```

Thus we are done with the evaluation now we need to work on Interpolation

Interpolation

- If the values used to obtain the value of plolynomial were the nth then interpolation can be done easily in the same manner

```
<values> = FFT(<coefficients>,w)
<coefficients> = 1/nFFT(<values>,w^-1)
```

Why is it so

We can see that evaluation is nothing but a linear transformation obtained by the matrix multiplication but matrix M

```
                                         M

 _            _          _                                      _   _   _
|   A(Xo)     |         |  1     Xo     Xo^2  ...    Xo^n-1    | |  a0  |
|   A(Xi)     |         |  1     Xi     Xi^2  ...    Xi^n-1    | |  a1  |
|      .      |    =    |                  .                  | |   .  |
|      .      |         |                  .                  | |   .  |
|      .      |         |                  .                  | |   .  |
|  A(Xn-1)    |         |  1    Xn-1   Xn-1^2 ...   Xn-1^n-1  | | an-1 |
--           --        --                                    --  --   --
```

And interpolation is nothing but multiplication by M^-1

FFT Algorithm

```
function FFT(a,w)
    if w = 1: return a
    (So , Si ..... Sn/2-1) = FFT((a0 , a2 ......an-2),w^2)
    (S`o , S`i ..... S`n/2-1) = FFT((a1 , a3 ......an-1),w^2)
    for j=0 to n-1:
        Rj = Sj + w^jS`j
        Rj+n/2 = Sj - w^jS`j

    return (Ro,Ri ..... Rn-1)
```

MergeSort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = l+ (r-l)/2
    2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

```
void merge(int array[], int const left, int const mid, int const
right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Create temp arrays
    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne = 0, // Initial index of first sub-array
        indexOfSubArrayTwo = 0; // Initial index of second sub-array
    int indexOfMergedArray = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo <
subArrayTwo) {
        if (leftArray[indexOfSubArrayOne] <=
rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray] =
leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
            array[indexOfMergedArray] =
rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
```

Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1 Always pick first element as pivot.

2 Always pick last element as pivot (implemented below)

3 Pick a random element as pivot.

4 Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

```
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element and indicates
the right position of pivot found so far

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Graphs

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices(or nodes) and a set of Edges which connect a pair of nodes.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale, etc.

Depth-First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

We might encounter cycles thus it is necessary to ensure that we dont visit the same node again
Algorithm:

1) Create a recursive function that takes the index of the node and a visited array.
2) Mark the current node as visited and print the node.

3) Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

CPP code for DFS is

```cpp
void Graph::DFS(int v)
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}
```

The military region

There is a military region with V military sites, each site is connect to some other sites using roads, to ensure security the military decides to install check points between all sites, but the existence of cycles defeats this purpose, so the military wants to detect any cycle if it exists.

For input we first input v and e, in the next e lines we input the edges source and destination

For this purpose they decide to use DFS algorithm

Solution
We simply perform DFS algorithm along any random root node, we maintain an array visited[ ], initially its false for all V nodes but if visited we make it true, if we ever revisit a node(which means a cycle) we would return from the function.

Things to note
If there exits a edgefrom u to v, we only add a single edge from u to v, none from v to u, because that would make a cycle

Here is a CPP algorithm for this problem

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

class Graph{
    int V;    // No. of vertices
    list<int> *adj;
    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], bool *rs);
    // used by isCyclic()
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // to add an edge to graph
    bool isCyclic();
    // returns true if there is a cycle in this graph
};

Graph::Graph(int V){
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w){
    adj[v].push_back(w); // Add w to v's list.
}


bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false){
        // Mark the current node as visited and part of
recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i){
            if ( !visited[*i] && isCyclicUtil(*i, visited,
recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }

    }
    recStack[v] = false;
    // remove the vertex from recursion stack
    return false;
```

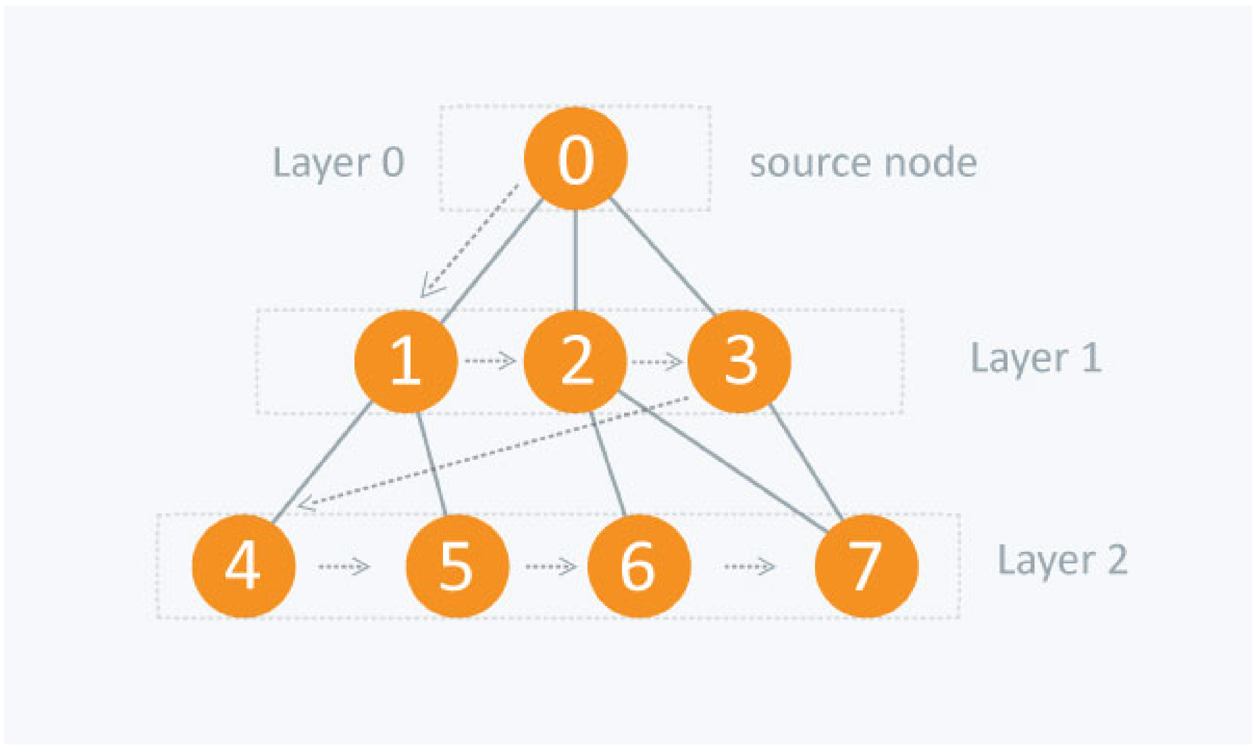
```
}

// Returns true if the graph contains a cycle, else false.
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of
recursion stack
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++){
        visited[i] = false;
        recStack[i] = false;
    }
  // Call the recursive helper function to detect cycle in
different DFS trees
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;
    return false;
}

int main(){
    int V, E;
    cin>>V>>E;
    Graph g(V);
    for(int i=0;i<E;i++){
     int a, b;
     cin>>a>>b;
     g.addedge(a,b);
    }
    if(g.isCyclic())
        cout << "Graph contains cycle";
    else
        cout << "Graph doesn't contain cycle";
    return 0;
}
```

Breadth-First Search

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.
This non-recursive implementation is similar to the non-recursive implementation of depth-first search, but differs from it in two ways:

1) it uses a queue (First In First Out) instead of a stack and
2) it checks whether a vertex has been explored before enqueueing the vertex rather than delaying this check until the vertex is dequeued from the queue.

For, eg, let us take the example of a family tree, with a root, then the first generation, then the second and so on, so while traversing through the family tree in BFS, we consider the root, then all members of first generation, then second and so on.



While executing the BFS algorithm we need to ensure that we don't visit the same node twice or more, thus we maintain a visited[ ] (or explored) array which becomes is false for all nodes but becomes true once we visit them, also, to reduce the time of execution, we use a queue to store nodes that we have visited.

Here is the pseudocode for the BFS algorithm

```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as explored then
                label w as explored
                Q.enqueue(w)
```

The CPP code for BFS algorithm,

```cpp
void Graph::BFS()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    int s = 0 ;
    // starting from root = 0
    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

The military tower problem

→ There are n military bases in a region
→ all bases are connected, but since its a military region to ensure proper checkpoints there are no cycles, thus all bases are connected in a form of tree
→ There are m sites which need to be governed, to ensure that a military site is governed a military tower must be placed within a distance k from the site.
→ Now to maintain the cost of operation, the military only wants to make one tower
→ They want to how many sites are such, that if a tower is built at a site then all marked sites are covered

For input we first input v and n, in the next (v-1) lines we input the edges source and destination, in the next lines we input the n marked sites. In the last line we input the threshold distance K

For eg

```
10 3                    // n sites out of which 3 are marked sites
0 1                     // road between site 0 and 1
0 3
0 8
3 2
3 6
3 7
3 5
5 9
5 4
1 2 4                   // marked sites
3                       // Threshold distance
```

Solution
We use BFS for this purpose. The main thing to observe in this problem is that if we find two marked sites that are at the largest distance from each other considering all pairs of marked sites then if a site is at a distance less than K from both of these two nodes then it will be at a distance less than K from all the marked nodes because these two nodes represent the extreme limit of all

marked nodes, if a node lies in this limit then it will be at a distance less than K from all marked nodes otherwise not.

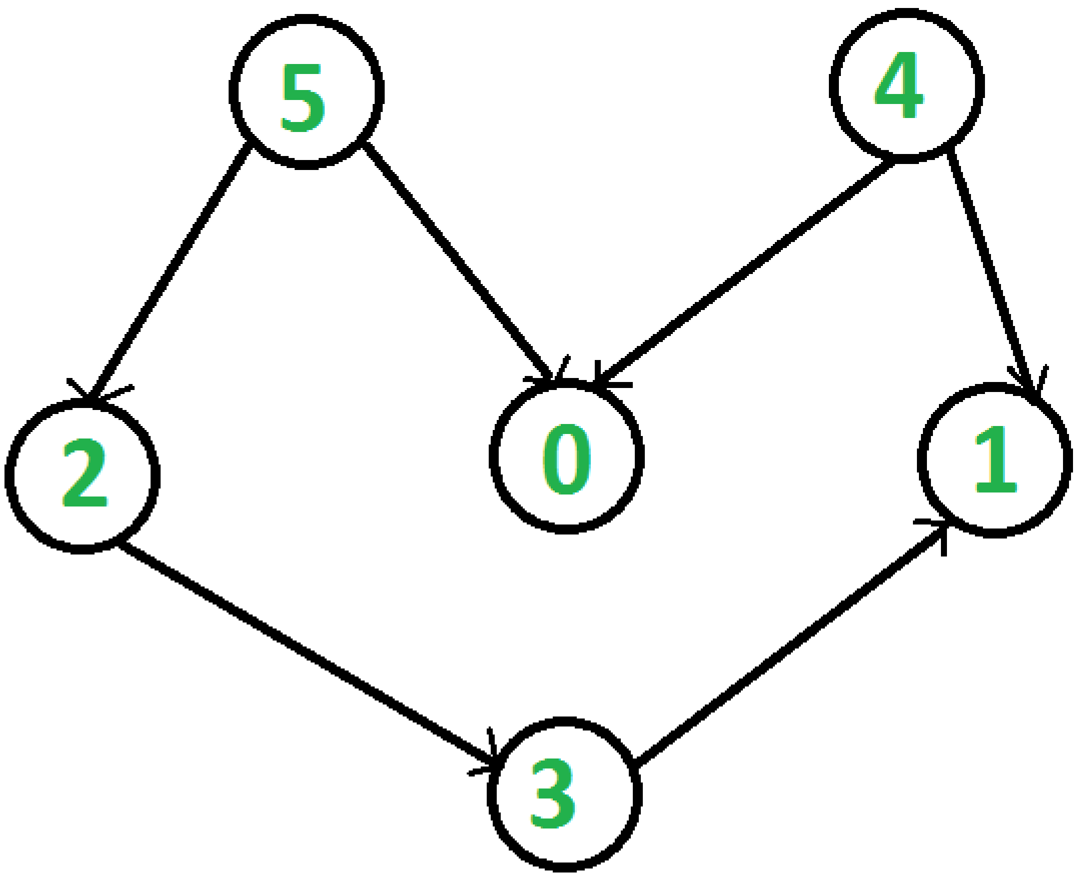
So what exactly are we doing
→ We take a random node and BFS about it to find the farthest marked node from this random vertex, say u
→ from u, we BFS to find the farthest marked node, say v
→ We calculated distance of each node from u and v, if this distance is less than K for both u and v, then we can build a tower at this site.

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

int bfsWithDistance(vector<int> g[], bool mark[], int u,
                                        vector<int>& dis)
{
    int lastMarked;
    queue<int> q;

    //  push node u in queue and initialize its distance as 0
    q.push(u);
    dis[u] = 0;

    //  loop untill all nodes are processed
    while (!q.empty())
    {
        u = q.front();      q.pop();
        //  if node is marked, update lastMarked variable
        if (mark[u])
            lastMarked = u;

        // loop over all neighbors of u and update their
        // distance before pushing in queue
        for (int i = 0; i < g[u].size(); i++)
        {
            int v = g[u][i];

            //  if not given value already
            if (dis[v] == -1)
            {
                dis[v] = dis[u] + 1;
// distance of node is equal to distance
                q.push(v);            // of connecting node + 1

            }
        }
    }
    //  return last updated marked value, as that is the node
farthest from our root
    return lastMarked;
}

// method returns count of nodes which are in K-distance
// range from marked nodes
int nodesKDistanceFromMarked(int edges[][2], int V,
                             int marked[], int N, int K)
{

    vector<int> g[V];


    int u, v;
    for (int i = 0; i < (V - 1); i++)
    {
        u = edges[i][0];
        v = edges[i][1];

        g[u].push_back(v);
      // we make the graph here using vectors
        g[v].push_back(u);
    }
    //  fill boolean array mark from marked array
    bool mark[V] = {false};
    for (int i = 0; i < N; i++)
        mark[marked[i]] = true;

    //  vectors to store distances
    vector<int> tmp(V, -1), dl(V, -1), dr(V, -1);

    //  first bfs(from any random node) to get one
    // distant marked node
    u = bfsWithDistance(g, mark, 0, tmp);      // u is the node
fathest from 0

    /*  second bfs to get other distant marked node
        and also dl is filled with distances from first
        chosen marked node */
    v = bfsWithDistance(g, mark, u, dl);       // v is the node
farthest from u

    //  third bfs to fill dr by distances from second
    // chosen marked node
    bfsWithDistance(g, mark, v, dr);

    int res = 0;
    //  loop over all nodes
    for (int i = 0; i < V; i++)
    {
        // increase res by 1, if current node has distance
        // less than K from both extreme nodes
        if (dl[i] <= K && dr[i] <= K)  // distance from u and v
(which are the most
            res++;              // distant maked nodes should
be less than K
    }
    return res;
}


int main()
{

    int V, N;
    cin>>V>>N;
    int edges[V-1][2];
    int marked[N];
    for(int i=0;i<V-1;i++){
     cin>>edges[i][0]>>edges[i][1];
    }
    for(int i=0;i<N;i++){
     cin>>marked[i];
    }
    int K ;
    cin>>k;
    cout << nodesKDistanceFromMarked(edges, V, marked, N, K);
    return 0;
}
```

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting.

In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

Here is CPP algorithm for topological sorting

```
void topologicalSortUtil(int v, bool visited[],stack<int>&
Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack
    // which stores result
    Stack.push(v);
}


void topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to store Topological
    // Sort starting from all
    // vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false) {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}
```

The Postman Problem
There are N people, some of which wants to deliver some mails to some other people.

Now the postman wants to minimize his work, thus he only wants to visit every person once, thus he decides that he will only visit a person if he gets the mails he wants to deliver to that person and on the same visit, he will take the mails the peron wants to send. The postman wants to find in what order should he visit these people for which he uses a topological sorting algorithm.

Input → we get V and E as input, where V is the number of people and E is the number of letters, the next E lines contain 2 numbers the sender and receiver.

Solution
We can think of this problem as a topological sorting problem, where there is a directed edge from person u to v, if u wants to send a letter to v, thus the postman will visit u before v, thus collecting all the letters he wants to give to v, thus minimizing his visits.

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


class Graph {
    int V;
    list<int>* adj;
    void topologicalSortUtil(int v, bool visited[],
                             stack<int>& Stack);

public:
    Graph(int V);
    void addEdge(int v, int w);
    void topologicalSort();
};
```

```
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    // Add w to v's list.
    adj[v].push_back(w);
}


void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int>& Stack)
{
    visited[v] = true;

    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    Stack.push(v);
}

void Graph::topologicalSort()
{
    stack<int> Stack;


    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    while (Stack.empty() == false) {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

// Driver Code
int main()
{
    int V,E;
    cin>>V>>E;
    Graph g(V);
    for(int i=0;i<E;i++){
     int a,b;
     cin>>a>>b;
     g,addEdge(a,b);
    }

    cout << "Following is a Topological Sort of the given "
            "graph \n";

    // Function Call
    g.topologicalSort();

    return 0;
}
```

Prim’s minimum spanning tree

What is Minimum Spanning Tree?
Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Prim’s algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.
A group of edges that connects two set of vertices in a graph is called cut in graph theory. So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim’s Algorithm Work?
The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Algorithm
1) Create a set mstSet that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as

INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn't include all vertices
….a) Pick a vertex u which is not there in mstSet and has minimum key value.
….b) Include u to mstSet.
….c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v
The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

How to implement the above algorithm?
We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

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


#define V 5
 // for this example we only consider a graph with 5 vertices

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
```

```
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<"
\n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
vertices of m
            // mstSet[v] is false for vertices not yet included
in MST
            // Update the key only if graph[u][v] is smaller
than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v]
< key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// Driver code
int main()
```

```
{

    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

The military road problem

There are V number of military site in a military region which are interconnected, now the military wants to construct roads between them, such that each site is reachable, but since these sites are not visited often the military wants to lay as less road as possible to reduce cost, even though time of travel may increase. For this purpose the military wants to find the minimum number of connected sites between which a road must be layed, for this purpose they use prim's algorithm

input
V, number of vertices, then a matrix of V*V dimension, where a(i)(j) means the distance of road connect node i to node j, if a(i)(j)=0, that means no road exists.

Solution
This can be considered similar to a undirected weighed graph for which we need to find MST such that all nodes are connected using edges such that sum of weight is as minimum as possible

CPP algorithm for this problem

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




// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<"
\n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
vertices of m
            // mstSet[v] is false for vertices not yet included
in MST
            // Update the key only if graph[u][v] is smaller
than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v]
< key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}
```

```
// Driver code
int main()
{

    int V ;
    cin>>V ;
    int graph[V][V] ;

    for(int i=0;i<V;i++){
     for(int j=0;j<V;j++){
          cin>>graph[i][j];
     }
    }

    primMST(graph);

    return 0;
}
```

Dijsktra’s shortest path algorithm

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.
Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.
Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn’t include all vertices

….a) Pick a vertex u which is not there in sptSet and has a minimum distance value.

….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

```
#include <iostream>
using namespace std;
#include <limits.h>

// For this example we only consider a graph with 9 vertices
#define V 9

// A utility function to find the vertex with minimum distance
value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{

    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array
void printSolution(int dist[])
{
    cout <<"Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout  << i << " \t\t"<<dist[i]<< endl;
}
```

```
// Function that implements Dijkstra's single source shortest
path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the
shortest
    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is
included in shortest
    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as
false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of
vertices not
        // yet processed. u is always equal to src in the first
iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the
picked vertex.
        for (int v = 0; v < V; v++)
```

```
            // Update dist[v] only if is not in sptSet, there is
an edge from
            // u to v, and total weight of path from src to  v
through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}

// driver program to test above function
int main()
{

    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}
```

Union-Find , Union of disjoint sets

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset. Here first we have to check if the two subsets belong to same set. If no, then we cannot perform union.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an algorithm to detect cycle. This is another method based on Union-Find. This method assumes that the graph doesn't contain any self-loops.

We can keep track of the subsets in a 1D array, let's call it parent[].

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

// a structure to represent an edge in graph
class Edge
{
public:
    int src, dest;
};

// a structure to represent a graph
class Graph
{
public:
    // V-> Number of vertices, E-> Number of edges
    int V, E;
    // graph is represented as an array of edges
    Edge* edge;
};

// Creates a graph with V vertices and E edges
Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph();
    graph->V = V;
    graph->E = E;

    graph->edge = new Edge[graph->E * sizeof(Edge)];

    return graph;
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    parent[x] = y;
}

// The main function to check whether a given graph contains
// cycle or not
int isCycle(Graph* graph)
{
    // Allocate memory for creating V subsets
    int* parent = new int[graph->V * sizeof(int)];

    // Initialize all subsets as single element sets
    memset(parent, -1, sizeof(int) * graph->V);

    // Iterate through all edges of graph, find subset of
    // both vertices of every edge, if both subsets are
    // same, then there is cycle in graph.
    for (int i = 0; i < graph->E; ++i) {
        int x = find(parent, graph->edge[i].src);
        int y = find(parent, graph->edge[i].dest);

        if (x == y)
            return 1;

        Union(parent, x, y);
    }
    return 0;
}

// Driver code
int main()
{
    /* Let us create the following graph
        0
        | \
        |  \
        1---2 */
    int V = 3, E = 3;
    Graph* graph = createGraph(V, E);

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

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

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        cout << "graph contains cycle";
    else
        cout << "graph doesn't contain cycle";

    return 0;
}
```

Find the number of Islands | Set 2 (Using Disjoint Set)

Given a boolean 2D matrix, find the number of islands.

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}

A cell in the 2D matrix can be connected to 8 neighbours.

This is a variation of the standard problem: “Counting the number of connected components in an undirected graph”.

We can solve the question using disjoint set data structure. The idea is to consider all 1 values as individual sets. Traverse the matrix and do a union of all adjacent 1 vertices. Below are detailed steps.

Approach:
1) Initialize result (count of islands) as 0
2) Traverse each index of the 2D matrix.
3) If the value at that index is 1, check all its 8 neighbours. If a neighbour is also equal to 1, take the union of the index and its neighbour.
4) Now define an array of size row*column to store frequencies of all sets.
5) Now traverse the matrix again.
6) If the value at index is 1, find its set.
7) If the frequency of the set in the above array is 0, increment the result be 1.

Kruskal’s minimum spanning tree

We will be using the previously learnt DSU algorithm for this algorithm

Here are the steps to obtain a graph’s MST using Kruskal’s algorithm

```
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the
spanning tree formed so far. If cycle is not formed, include
this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning
tree.
```

Here is the cpp algorithm for Kruskal’s MST

```cpp
#include<bits/stdc++.h>
using namespace std;
// DSU data structure
//  path compression + rank by union

class DSU
{
    int *parent;
    int *rank;

public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++)
        {
            parent[i] = -1;
            rank[i] = 1;
        }
    }

    // Find function
    int find(int i)
    {
        if (parent[i] == -1)
            return i;

        return parent[i] = find(parent[i]);
    }
    // union function
    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2)
        {
            if (rank[s1] < rank[s2])
            {
                parent[s1] = s2;
                rank[s2] += rank[s1];
            }
            else
            {
                parent[s2] = s1;
                rank[s1] += rank[s2];
            }
        }
    }
};

class Graph
```

```cpp
{
    vector<vector<int>> edgelist;
    int V;

public:
    Graph(int V)
    {
        this->V = V;
    }

    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({w, x, y});
    }

    int kruskals_mst()
    {
        // 1. Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU
        DSU s(V);
        int ans = 0;
        for (auto edge : edgelist)
        {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];

            // take that edge in MST if it does form a cycle
            if (s.find(x) != s.find(y))
            {
                s.unite(x, y);
                ans += w;
            }
        }
        return ans;
    }
};

int main()
{

    int n, m;
    cin >> n >> m;

    Graph g(n);
    for (int i = 0; i < m; i++)
    {
     int x, y, w;
     cin >> x >> y >> w;
     g.addEdge(x, y, w);
    }

    cout << g.kruskals_mst();
    return 0;
}
```

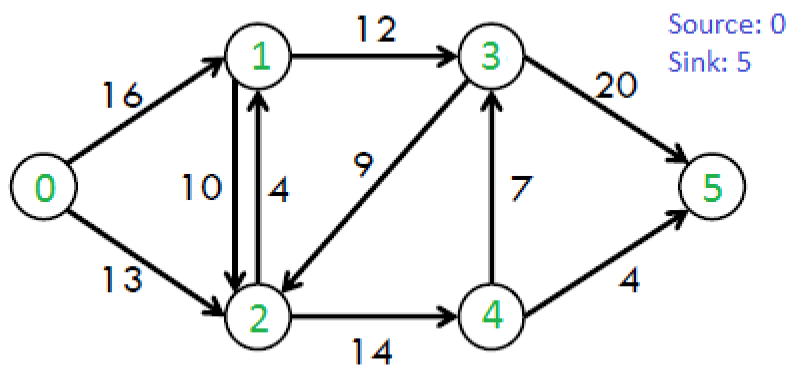
Ford-Fulkerson Algorithm for Maximum Flow Problem

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* ‘s’ and *sink* ‘t’ in the graph, find the maximum possible flow from s to t with following constraints:

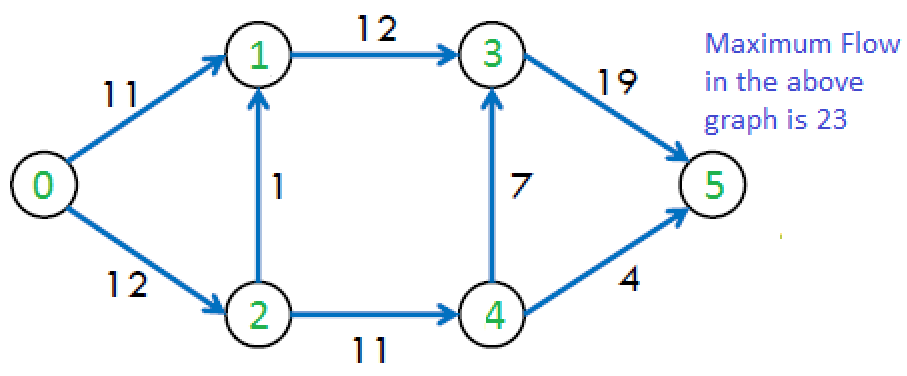
a) Flow on an edge doesn’t exceed the given capacity of the edge.

b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.



```
Ford-Fulkerson Algorithm
The following is simple idea of Ford-Fulkerson algorithm:
1) Start with initial flow as 0.
2) While there is a augmenting path from source to sink.
           Add this path-flow to flow.
3) Return flow.
```

Time Complexity: Time complexity of the above algorithm is O(max_flow * E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max_flow * E).

How to implement the above simple algorithm?

Let us first define the concept of Residual Graph which is needed for understanding the implementation.

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called **residual capacity** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.

Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.

The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges We need to add path flow along reverse edges because may later need to send flow in reverse direction

Below is the implementation of Ford-Fulkerson algorithm. To keep things simple, graph is represented as a 2D matrix.

```cpp
#include <iostream>
#include <limits.h>
#include <queue>
#include <string.h>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink
  't' in residual graph. Also fills parent[] to store the
  path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not
    // visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source
    // vertex as visited
    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 0; v < V; v++) {
            if (visited[v] == false && rGraph[u][v] > 0) {
                // If we find a connection to the sink node,
                // then there is no point in BFS anymore We
                // just have to set its parent and can return
                // true
                if (v == t) {
                    parent[v] = u;
                    return true;
                }
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    // We didn't reach sink in BFS starting from source, so
    // return false
    return false;
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph
    // with given capacities in the original graph as
    // residual capacities in residual graph
    int rGraph[V]
              [V]; // Residual graph where rGraph[i][j]
                   // indicates residual capacity of edge
                   // from i to j (if there is an edge. If
                   // rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
```

```
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to
                   // store path

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source to
    // sink
    while (bfs(rGraph, s, t, parent)) {
        // Find minimum residual capacity of the edges along
        // the path filled by BFS. Or we can say find the
        // maximum flow through the path found.
        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and
        // reverse edges along the path
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}

// Driver program to test above functions
```

```cpp
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V]
        = { { 0, 16, 13, 0, 0, 0 }, { 0, 0, 10, 12, 0, 0 },
            { 0, 4, 0, 0, 14, 0 },  { 0, 0, 9, 0, 0, 20 },
            { 0, 0, 0, 7, 0, 4 },   { 0, 0, 0, 0, 0, 0 } };

    cout << "The maximum possible flow is "
         << fordFulkerson(graph, 0, 5);

    return 0;
}
```

Sorting

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.
In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.
Following example explains the above steps:

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

CPP code for selection sort

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Bubble sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
Example:

First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**),
Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is

completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1** **2** 4 5 8) –> (**1** **2** 4 5 8)
(1 **2** **4** 5 8) –> (1 **2** **4** 5 8)
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8)
(1 2 4 **5** **8**) –> (1 2 4 **5** **8**)

CPP implpementation for bubble sort

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver code
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout<<"Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

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

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

```
// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

/* Driver code */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Number theory

Euclidean Algorithm for GCD

GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors. The algorithm is based on the below facts.

- If we subtract a smaller number from a larger (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find remainder 0.

Below is a recursive function to evaluate gcd using Euclid's algorithm.

```
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}
```

Extended Euclid’s algorithm

Extended Euclidean algorithm also finds integer coefficients x and y such that:

```
ax + by = gcd(a, b)
```

The extended Euclidean algorithm updates results of gcd(a, b) using the results calculated by recursive call gcd(b%a, a). Let values of x and y calculated by the recursive call be x1 and y1. x and y are updated using the below expressions.

$$x = y_1 - \lfloor b/a \rfloor * x_1$$
$$y = x_1$$

```
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b%a, a, &x1, &y1);

    // Update x and y using results of
    // recursive call
    *x = y1 - (b/a) * x1;
    *y = x1;

    return gcd;
}
```

How does Extended Algorithm Work?

As seen above, x and y are results for inputs a and b,

a.x + b.y = gcd ----(1)

And x1 and y1 are results for inputs b%a and a
   (b%a).x1 + a.y1 = gcd

When we put b%a = (b - (⌊b/a⌋).a) in above,
we get following. Note that ⌊b/a⌋ is floor(b/a)

   (b - (⌊b/a⌋).a).x1 + a.y1 = gcd

Above equation can also be written as below
   b.x1 + a.(y1 - (⌊b/a⌋).x1) = gcd   ---(2)

After comparing coefficients of 'a' and 'b' in (1) and
(2), we get following
   x = y1 - ⌊b/a⌋ * x1
   y = x1

How is Extended Algorithm Useful?

The extended Euclidean algorithm is particularly useful when a and b are coprime (or gcd is 1). Since x is the modular multiplicative inverse of “a modulo b”, and y is the modular multiplicative inverse of “b modulo a”. In particular, the computation of the modular multiplicative inverse is an essential step in RSA public-key encryption method.

Modular Division

Given three positive numbers a, b and m. Compute a/b under modulo m. The task is basically to find a number c such that (b * c) % m = a % m.

Can we always do modular division?
The answer is “NO”. First of all, like ordinary arithmetic, division by 0 is not defined. For example, 4/0 is not allowed. In modular arithmetic, not only 4/0 is not allowed, but 4/12 under modulo 6 is also not allowed. The reason is, 12 is congruent to 0 when modulus is 6.

When is modular division defined?
Modular division is defined when modular inverse of the divisor exists. The inverse of an integer ‘x’ is another integer ‘y’ such that (x*y) % m = 1 where m is the modulus.
When does inverse exist? As discussed here, inverse a number ‘a’ exists under modulo ‘m’ if ‘a’ and ‘m’ are co-prime, i.e., GCD of them is 1.

How to find modular division?

```
The task is to compute a/b under modulo m.
1) First check if inverse of b under modulo m exists or not.
    a) If inverse doesn't exists (GCD of b and m is not 1),
          print "Division not defined"
    b) Else return  "(inverse * a) % m"
```