Standard Template Library

1. Vector

In a simple static array, we can not redefine its size. So it is optimal to make static arrays only where the size of the array doesn't change according to the user's needs. Also, the size should be defined before the compilation because stack memory is assigned at the time of compilation of the program. For example,

- **int arr[20];** **Here, the stack memory that the arr requires is fixed**

- **int n;**
  cin >> n;
  int arr[n];
  In this case some random amount of stack memory is assigned to arr but that might be short than what the user might require.

The above problem can be solved using Vectors. **A Vector is a dynamic array that has the ability to modify its shape whenever we perform insertion or deletion in it.** Just like an array, Vector elements are placed in contiguous storage so that they can be accessed using iterators.

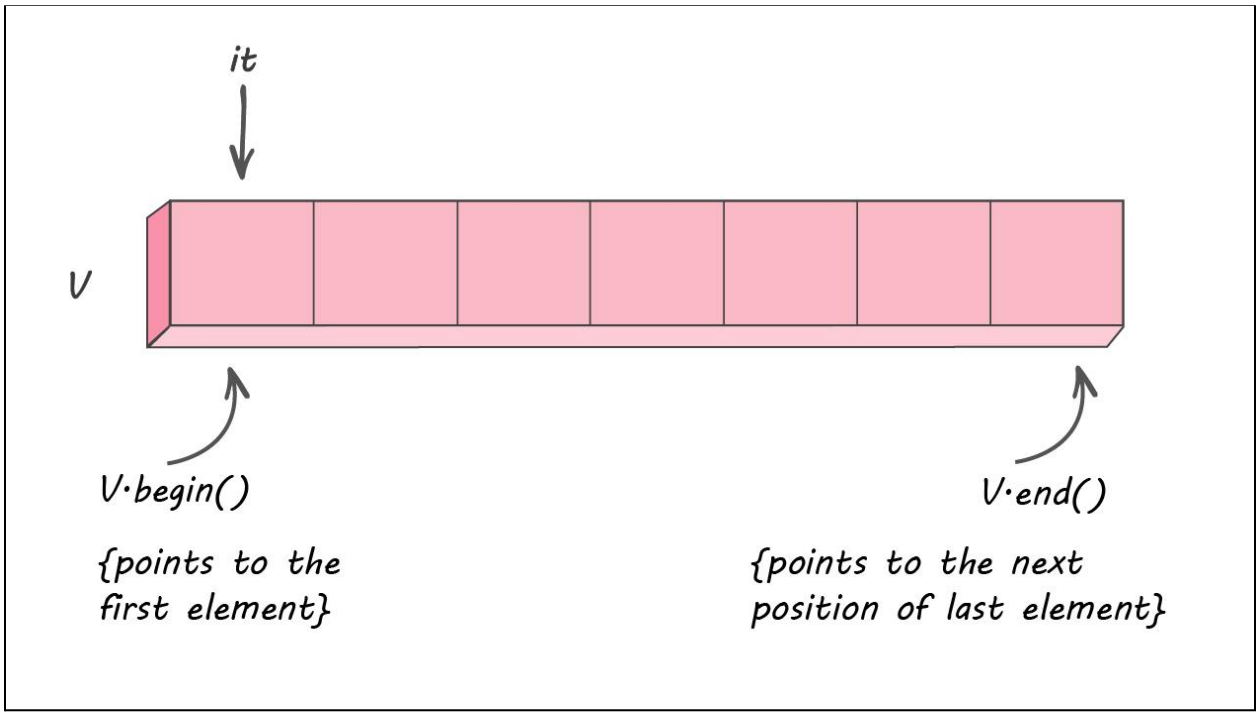
Vector is already present in the standard template library of C++ and can be included in any file using **#include<vector>**

Any new element can be inserted at the end of the vector using the **push_back()** function, which takes differential time because there might be

resizing the vector. While removing the last element takes constant time because no resizing happens.

```
#include<iostream>
#include<vector>

using namespace std;

int main(){

    // while making a vector we need not give the size of the vector
    vector<int> V ;

    // we insert the element at the last index of the vector and
    // the vector automatically reassigns its size
    for(int i=0; i<5; i++){
        V.push_back(i+1);
    }

    return 0;
}
```

There are predefined functions like **begin()**, **end()**, **size()**, etc., that can be used along with a vector.

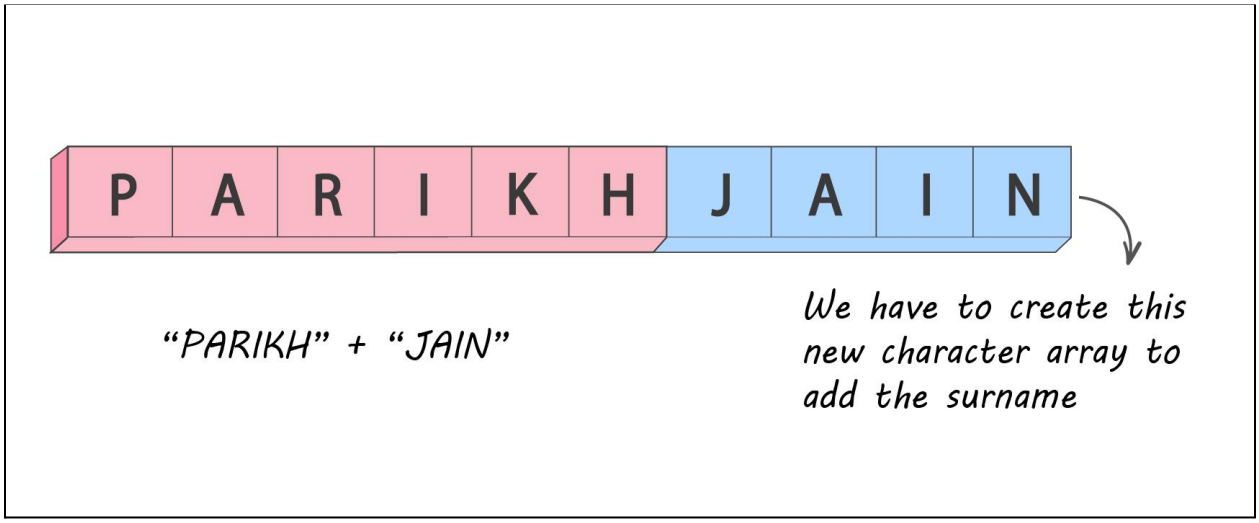


If we need to traverse the entire vector we can do it using creating an iterator:

```
// creating an iterator and traversing the vector using it
vector <int> :: iterator it;
for(it = v.begin(); it != v.end(); it++){
    cout << *it << endl;
}
```

2. Strings

A character array is a static array, and more memory can not be used at run time. Let us suppose we made a character array “PARIKH”. Now we need to add the surname but the length/size of the character array is static, so we have to make a new character array “JAIN” and then use some iteration method to get the full name.



The above problem can be made way easier with a String. A String is a class that defines objects that are represented as a stream of characters.

String class is present in the standard template library of C++, and can be included in any file using **#include<string>**

```
#include<iostream>
#include<string>
```

```
using namespace std;


int main(){

    string s = "PARIKH";
    cout << s << endl; // prints PARIKH
    s = "PARIKH JAIN";
    cout << s << endl; // prints PARIKH JAIN
    return 0;
}
```

Various functionalities of strings are explained in the code below with proper comments:

```
int main(){

    string s = "PARIKH";
    string s1(s, 2, 4);//from index 2 upto 4 characters are copied to s1 from s

    // compare string s and s1
    if(s1.compare(s) == 0){
        cout << s1 << " is equal to " << s << endl;
    } else {
        cout << s1 << " is not equal to " << s << endl;
    }

    // substring from index 1 in s upto 4 characters is stored in s2
    string s2 = s.substr(1, 4);
    cout<< s2 << endl; // prints ARIK
}
```

3. Pair Class

The Pair Class contains a pair of homogeneous or heterogeneous values. It can be understood as a container in which the pair of values are identified by **"first"** and **"second"** and can be of different or same data type.

Pair class is present in the standard template library of C++, and can be included in any file using **#include<utility>**

Syntax for creating a pair class:

pair < first_data_type, second data_type > p(first, second);

Example :

pair < int, char > p(1, 'a');

To access the elements, we use variable name followed by dot operator followed by the keyword first or second:

cout << p.first << " " << p.second << endl;

```
#include<iostream>
#include<utility>

using namespace std;

int main(){

    // various ways to initialize a pair

    pair<int, char> p;
    p = make_pair(2, 'b');

    pair<int, int> p1(3, 4);

    pair<int, char> p2(1, 'a');

    // accessing elements inside a pair using first and second
```

```
    cout << p.first << " " << p.second << endl; // output is 2 b
    cout << p1.first << " " << p1.second << endl; // output is 3 4
    cout << p2.first << " " << p2.second << endl; // output is 1 a

    return 0;
}
```

4. Set

Set is a container which contains unique elements. The element itself is identified by its value. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.
A set can be used in a problem where we have to know whether the element is present or not or suppose we have an array = [1, 2, 3, 4, 5, 6, 5]. Here the number 5 is occurring two times in the array but when we make a set using this array the occurrence of the number 5 is only once just like all the other numbers / elements.

Set is present in the standard template library of C++, and can be included in any file using **#include<set>**

Some functionalities of Set include:

- **begin()** – Returns an iterator to the first element in the set.
- **end()** – Returns an iterator to the theoretical element that follows the last element in the set.
- **size()** – Returns the number of elements in the set.
- **empty()** – Returns whether the set is empty.
- **find()** - Returns an iterator if the element is present.

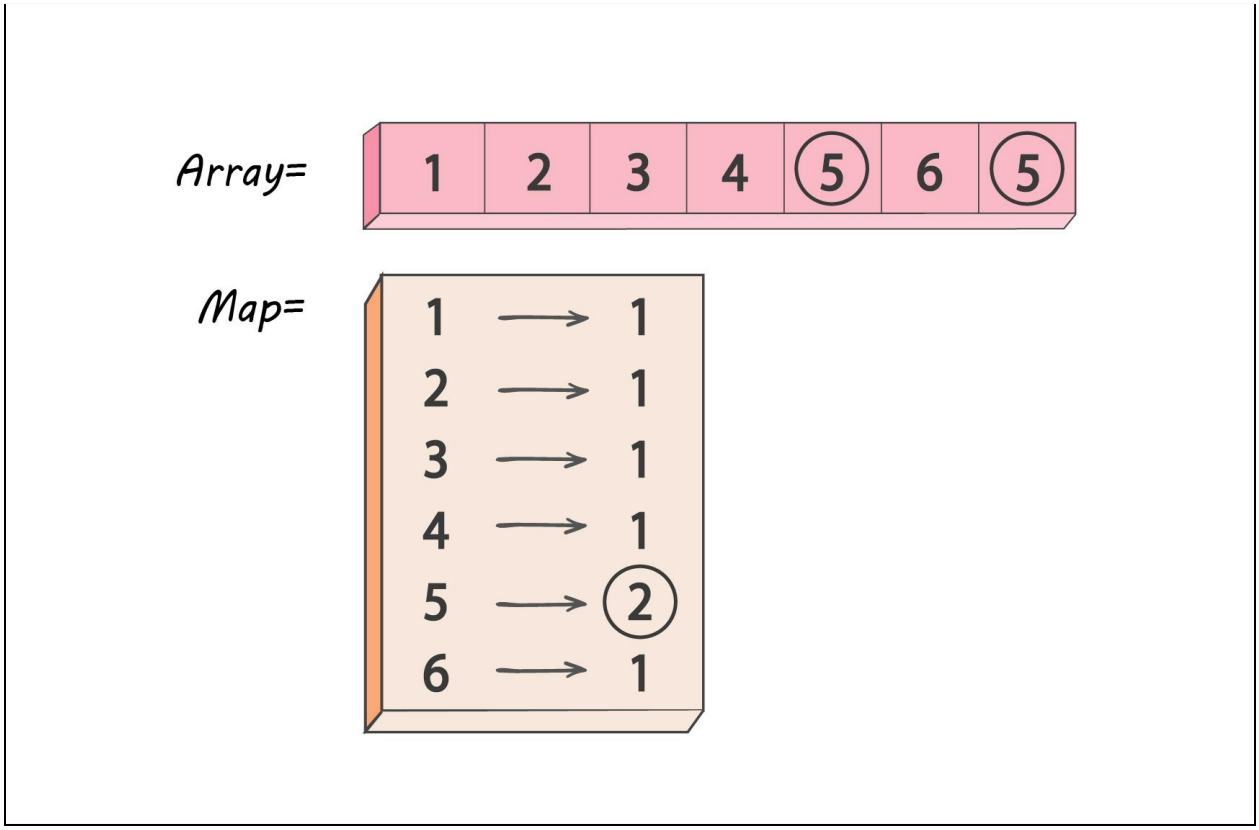
A set is generally implemented using a balanced binary search tree which takes **O(log(N))** time to find an element.

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

```
int main(){

    // initializing a set
    set<int> s;
    int arr[] = {1,2,3,4,5,6,5};

    // inserting elements into a set
    for(int i=0; i<7; i++){
        s.insert(arr[i]);
    }
    // creating an iterator to traverse through the set
    set<int>::iterator it;

    for(it=s.begin(); it!=s.end(); it++){
        cout << *it << endl;
    }
    // finding elements in set
    if(s.find(7) == s.end()){
        cout << "Element not found" << endl;
    } else {
        cout << "Element found" << endl;
    }
    return 0;
}
```

5. Map

Consider the previous example used in sets where we made a set using the **array = [1, 2, 3, 4, 5, 6, 5]**. Now if we want to store the frequency of each element, we can not do that by using a set. Therefore, we use a map that stores their frequencies like a **key-value pair in addition to storing unique elements.**



Map is present in the standard template library of C++, and can be included in any file using **#include<map>**

Syntax to make a map:

map < int, int > M;

Some functionalities of Set include:

- **begin()** – Returns an iterator to the first element in the map
- **end()** – Returns an iterator to the theoretical element that follows last element in the map
- **size()** – Returns the number of elements in the map
- **empty()** – Returns whether the map is empty
- **pair insert(keyValue, mapValue)** – Adds a new element to the map

The **keyValue** and **mapValue** can be accessed using **M -> first** and **M -> second.**

A Map is generally implemented using a balanced binary search tree which takes **O(log(N))** time to do operations.

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

using namespace std;

int main(){


    int arr[] = {1,2,3,4,5,6,5};
    map<int, int> m;

    for(int i=0; i<7; i++){
        m[arr[i]] = m[arr[i]] + 1;
    }

    map<int, int>::iterator it;

    for(it=m.begin();it!=m.end();it++){
        cout<<it->first << " :" << it->second<<endl;
    }
    cout<<endl;
    m.erase(1);
    for(it=m.begin();it!=m.end();it++){
```

```
        cout<<it->first << " :" << it->second<<endl;
    }
}
```

6. unordered_map

A normal Map is implemented using a balanced binary search tree which takes **O(log(N))** time to perform operations like insert , search etc but an **unordered_map** is implemented using a hash table which takes **O(1)** time in average case to perform operations like insert, search, etc. However, in the worst case scenario it takes **O(N)** time.
unordered_map is present in the standard template library of C++, and can be included in any file using **#include<unordered_map>**

```
#include<iostream>
#include<unordered_map>

using namespace std;

int main(){


    int arr[] = {1,2,3,4,5,6,5};
    unordered_map<int,int> m;

    for(int i=0;i<7;i++){
        m[arr[i]] = m[arr[i]]+1;
    }

    unordered_map<int,int>::iterator it;
    for(it=m.begin();it!=m.end();it++){
        cout<<it->first << " :" << it->second<<endl;
    }
    cout<<endl;
    m.erase(1);
    for(it=m.begin();it!=m.end();it++){
        cout<<it->first << " :" << it->second<<endl;
    }

}
```

STL Functions

While working on a problem that requires some basic tasks like sorting, searching, etc we are not required to make a separate functionality to achieve them. Rather, these functionalities are already present in C++ STL.

1. sort()

The sort function in STL is given a start pointer and an end pointer and it sorts any array in **O(log(N))** time complexity.
It can be used in any file using **#include<algorithm>.**

For example, arr = [1, 3, 2, 5, 7, 6]

sort(arr, arr+6);	**Sorts the array in ascending order**
sort(arr, arr+6, greater<int>);	**Sorts the array in descending order**

```
int main(){

    int arr[] = {1, 3, 2, 5, 7, 6};
    sort(arr, arr+6);
    for(int i=0; i<6; i++){
        cout << arr[i] << " ";
    }

    return 0;
}
```

We can also sort the array of objects on the basis of one of the properties of a class or structure to whom the object belongs to. For that we are required

to pass a **compare function** along with other parameters and have to define this compare function.

```
#include<iostream>
#include<algorithm>

using namespace std;

// creating a structure Interval
// with a start time and end time
struct Interval{
    int st;
    int et;
};
// compare function to pass in the sort function
// to sort on the basis of start time
bool compare(Interval i1,Interval i2){
    return i1.st < i2.st; // > for descending
}

int main(){

    Interval arr[] = {{6, 4}, {3, 4}, {4, 6}, {8, 13}};
    sort(arr, arr+4, compare);

    for(int i=0; i<4; i++){
        cout << arr[i].st << " : " << arr[i].et << endl;
    }

    return 0;
}
```

2. binary_search

Binary search can be applied only on a sorted array and its time complexity is **O(log(N)).**

Syntax : **binary_search(arr, arr+n, element to be searched);**

3. lower_bound

lower_bound() returns an iterator pointing to the first element in the range [first, last). The function returns the index of the next smallest number just greater than or equal to that number we are searching for. If there are multiple values that are equal to the element to be found, lower_bound() returns the index of the first such value.

Syntax : **lower_bound(arr, arr+n, element);**

Examples:

1. **Input: 10 20 30 40 50**
   Output: lower_bound for element 30 at index 2

2. **Input: 10 20 30 40 50**
   Output: lower_bound for element 35 at index 3

4. upper_bound

upper_bound() returns an iterator pointing to the first element in the range [first, last) that is greater than the element that needs to be found, or last if no such element is found. The elements in the range shall already be sorted or at least partitioned with respect to the element.

Syntax : **lower_bound(arr, arr+n, element);**
Examples :

1. **Input : 10 20 30 30 40 50**
   Output : upper_bound for element 30 is at index 4

2. **Input : 10 20 30 40 50**
   Output : upper_bound for element 45 is at index 4

```
int main(){

    int arr[] = {1,3,2,5,7,6};

    sort(arr,arr+6);

    for(int i=0;i<6;i++){
        cout<<arr[i] << " ";
    }
    cout<<endl;
    cout << binary_search(arr,arr+6,2) << endl;

    cout<<lower_bound(arr,arr+6,3) - arr << endl;

    cout<<upper_bound(arr,arr+6,3) - arr << endl;

    return 0;
}
```

STL Examples

1. Remove Duplicates from Array

Input Format :

Line 1: Contains the size of array

Line 2: M integers which are elements of the array separated by space

Output :

Resultant array

Sample Input :

7

2 4 3 3 5 3 4

Sample Output :

2 3 4 5

Explanation :

Approach 1 : This problem can be easily solved using a Set. We will first make a set using the inbuilt STL and a **result** array. Then we will traverse through our input array and start adding the elements in the set. While traversing the input array if we encounter any duplicates, that is if the element has already been added in the set , then we will not add the element to the result array otherwise we will.

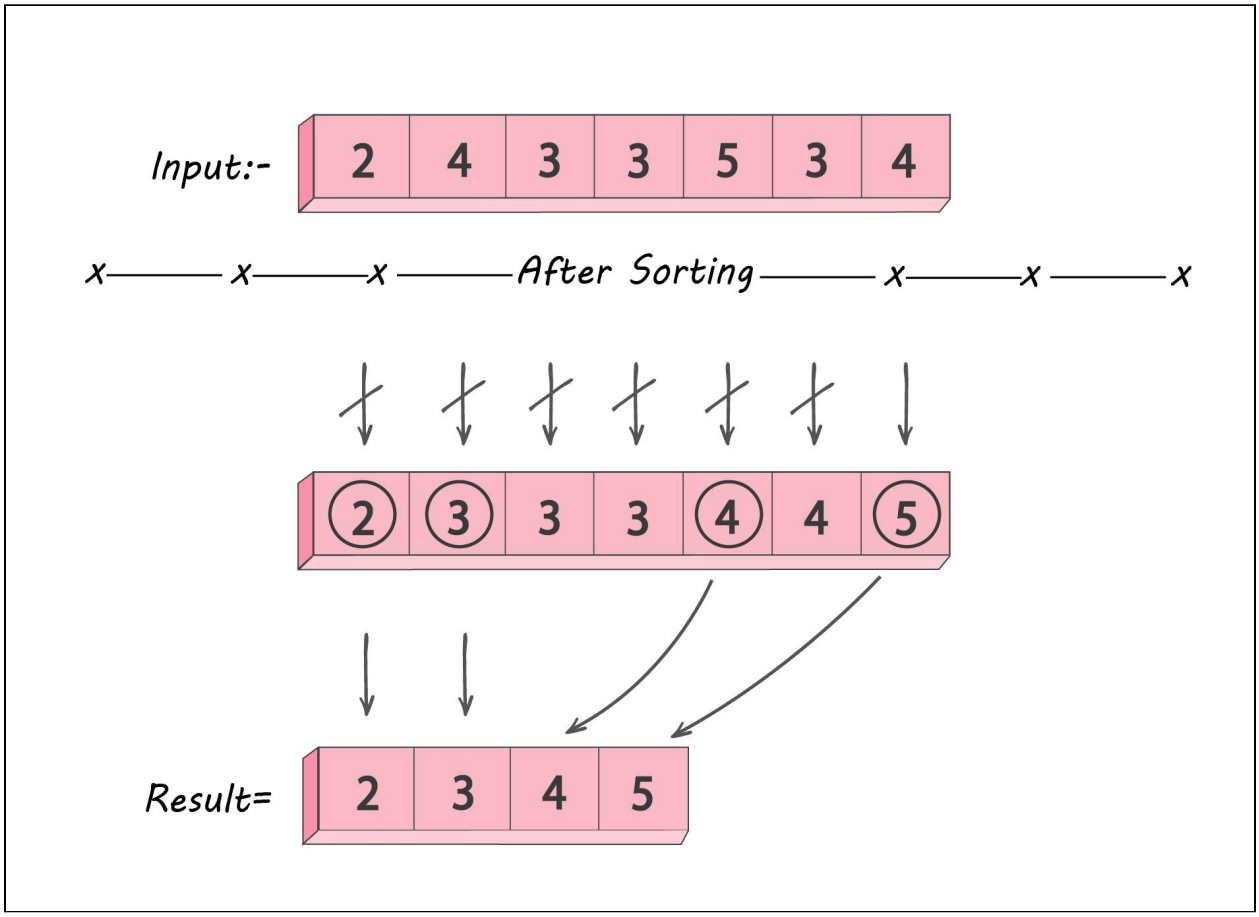
```
vector<int> removeDuplicates(vector<int> input){
```

```
    set<int> s;
    vector<int> result;

    for(int i=0; i<input.size(); i++){

        if(s.find(input[i]) == s.end()){
            s.insert(input[i]);
            result.push_back(input[i]);
        }
    }

    return result;

}
```

Time Complexity: O(N)
Space Complexity: O(N)

Approach 2 : Sort the elements in the input array so that all the duplicate elements will be together. Then traverse through the array and compare elements with the element present at the previous index; if they are equal do not add them in the result array otherwise add them.

For example :




```
vector<int> removeDuplicates(vector<int> input){
    vector<int> result;
    sort(input.begin(), inpt.end());

    result.push_back(input[0]);

    for(int i=1; i<input.size(); i++){

        if(input[i] != input[i-1]){
            result.push_back(input[i]);
        }
    }
    return result;
}
```

Time Complexity : O(n)

2. First Non Repeating Character in a String

If there is no repeating character then print the first character of the string.

Input Format :

Line 1: A String

Output :

First non repeating character

Sample Input :

aDcadhc

Sample Output :

D

Explanation : This problem can be solved using a **map** because with a map we can store the elements along with their frequencies.

```
char nonRepeatingCharacter(string str){

    map<char, int> frequency;
    for(int i=0; i<str.length(); i++){
        char currentChar = str[i];
        frequency[currentChar++];
    }

    for(int i=0; i<str.length(); i++){
        if(frequency[str[i]] == 1){
            return str[i];
        }
    }
    // if there is no unique character in the string
    return str[0];
}
```

Searching & Sorting Applications

1. Aggressive Cows

Problem Statement:

Farmer John has built a new long barn, with N (2 <= N <= 100,000) stalls. The stalls are located along a straight line at positions x1,...,xN (0 <= xi <= 1,000,000,000).

His C (2 <= C <= N) cows don't like this barn layout and become aggressive towards each other once put into a stall. To prevent the cows from hurting each other, FJ wants to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?

Explanation:

Approach 1:

We are given **N stalls (2 <= N** <= **10^6)** and **C cows (2 <= C <= N)** and we have to place the cows in these stalls such that there is largest minimum distance between them. So suppose we are given the input **N = 5** and **C = 3** and the position of the stalls are at **1 2 8 4 9**.

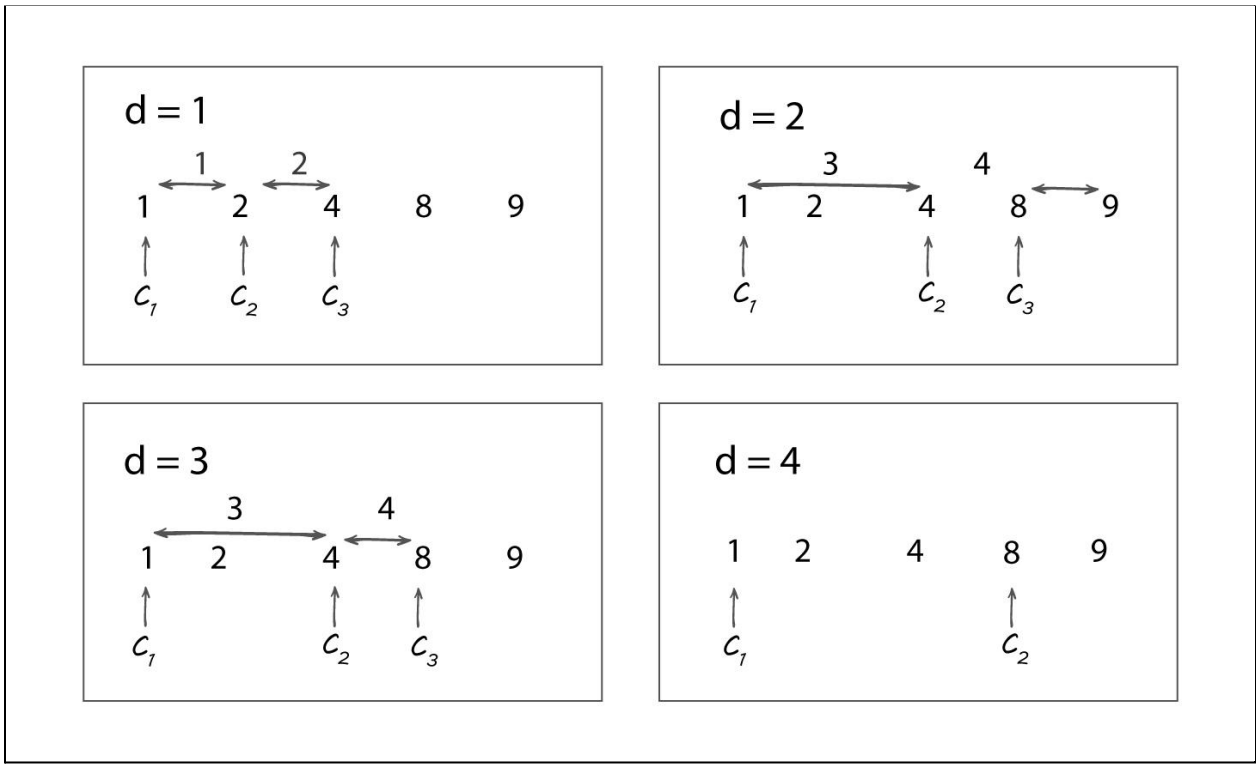
One approach that comes to mind is to arrange these positions of stalls in a **sorted array [1, 2, 4, 8, 9]**. Now, the **min_distance = 0** if we place C1 and C2 at positions 1 & 2; and the **max_distance = 9 - 1 = 8** if the two cows are at the ends of the array. That is, **0 <= distance (d) <= 8**

0 <= d <= position[N-1] - position[0]

Now, for every possible **d,** we check whether we can place C = 3 cows or not and amongst the possible answers we choose the maximum value as our answer.

Problem with this approach:

In the above solution we are traversing and solving for every possible value of **d** so the time complexity of this solution is **O(d*N)** which in the worst case will be $10^9 * 10^6 = 10^{14}$ that is not acceptable. Therefore, we must think of another approach to solve this problem.

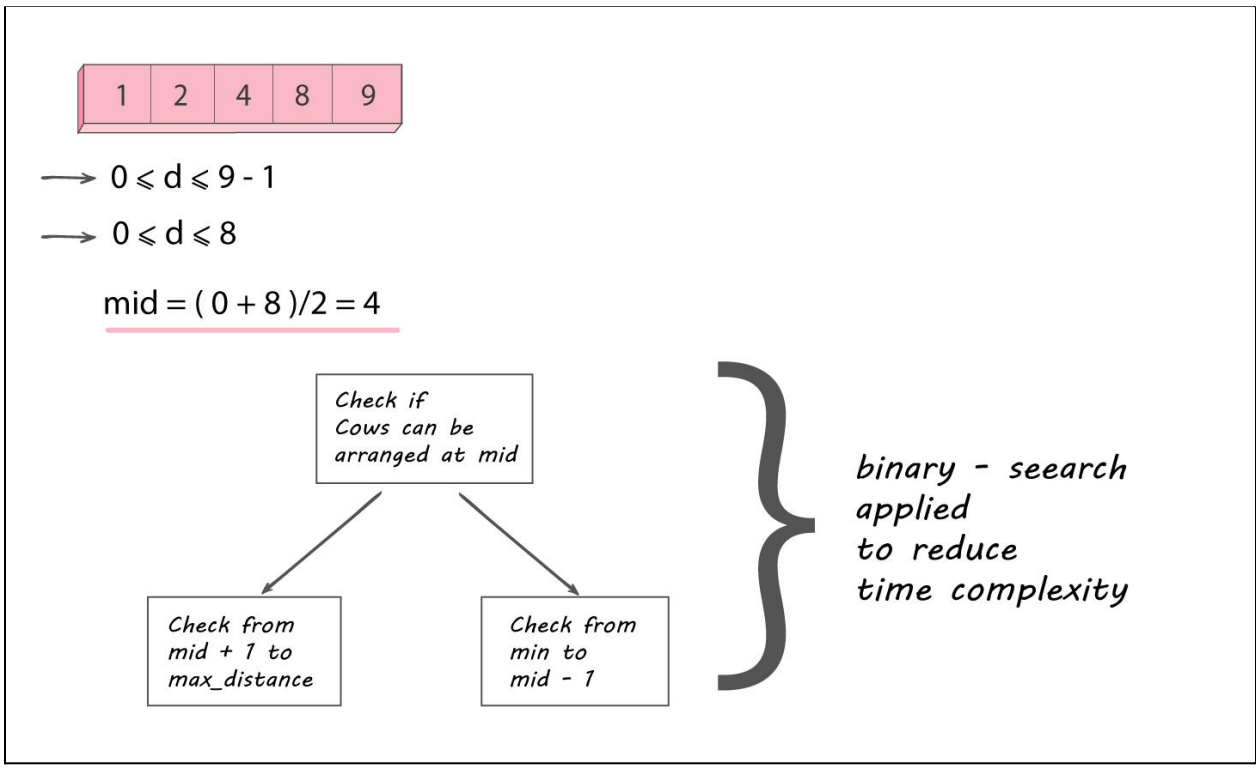


Approach 2:

In approach 1 we were checking for every possible **d** but now we will apply the concept of **binary search** to find out upto which d we can get the answer to reduce the searching process to **O(log(d)).** That is, we find

mid = (min_distance + max_distance)/2;

Now if it is possible to place the cows at mid distance from each other then we will apply binary search from mid + 1 to max_distance otherwise we will apply binary search from min to mid - 1.




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

bool check(int cows,long long positions[],int n,long long distance){

    int count = 1;
    long long last_position = positions[0];

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

```
        if(positions[i] - last_position >= distance){
            last_position = positions[i];
            count++;
        }

        if(count == cows){
            return true;
        }
    }
    return false;
}

int main(){
    int t;
    cin >> t;
    while(t--){
        int n,c;
        cin >> n >> c;

        long long positions[n];
        for(int i=0;i<n;i++){
            cin >> positions[i];
        }
        sort(positions,positions+n);
        long long start = 0;
        long long end = positions[n-1] - positions[0];

        long long ans = -1;

        while(start<=end){
            long long mid = start + (end-start)/2;

            if(check(c,positions,n,mid)){
                ans = mid;
                start = mid+1;
            }else{
                end = mid-1;
            }

        }
        cout << ans <<endl;
```

```
    }

    return 0;
}
```

2. Inversion Count

Problem Statement:

For a given integer array/list of size N, find the total number of 'Inversions' that may exist.

An inversion is defined for a pair of integers in the array/list when the following two conditions are met.

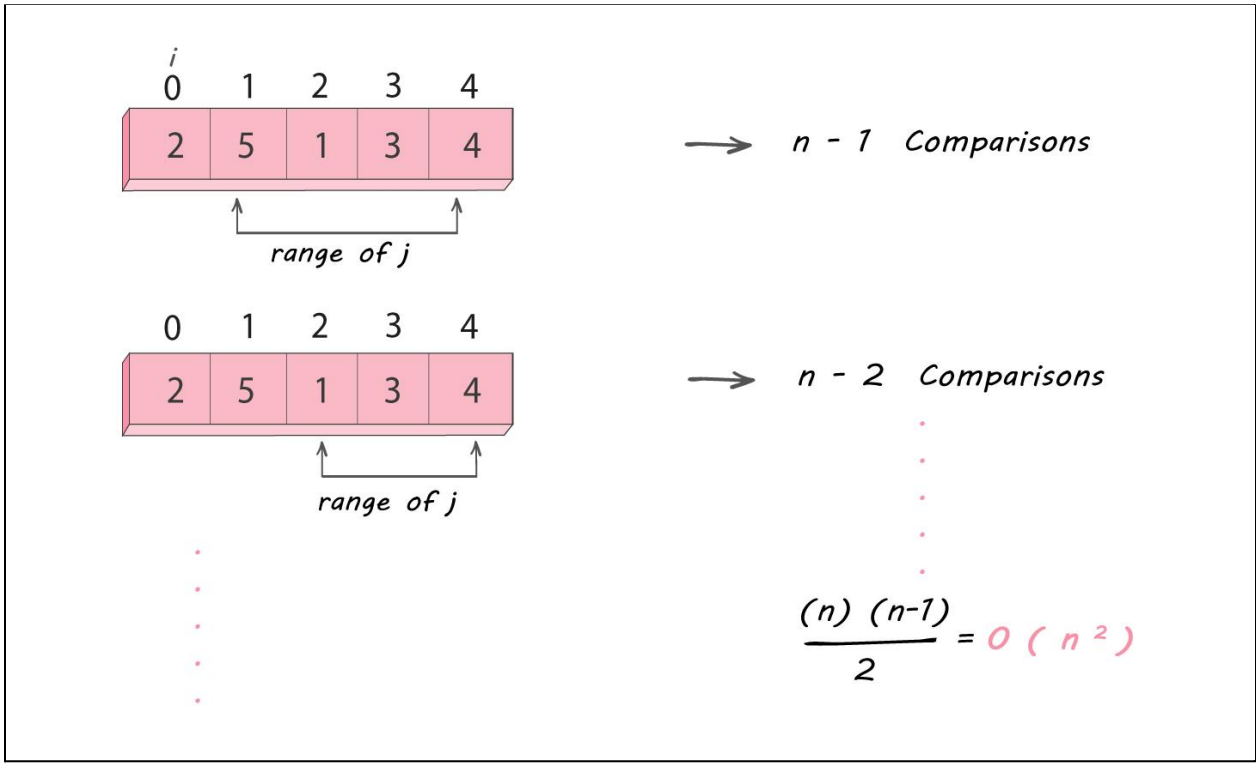
A pair (arr[i], arr[j]) is said to be an inversion when,

1. arr[i] > arr[j]
2. i < j

Where 'arr' is the input array/list, 'i' and 'j' denote the indices ranging from [0, N).

Explanation:

Approach 1: Consider the A = [2, 5, 1, 3, 4]. For this we have to calculate the number of inversions. An inversion is counted when **A[i] > A[j] given that, i < j**. The basic approach that comes to mind is to compare each element with all the elements present in the array from the next index. That is, we use 2 loops and **for each index i we compare A[i] with A[i+1] to A[N-1].**



The time complexity of this approach will be **O(N^2)** and is not acceptable in this problem.

Approach 2: If we divide the array in two parts and then calculate the number of inversions in each part and at the time of merging calculate the newly formed inversions then also we are traversing at almost every element to compare.

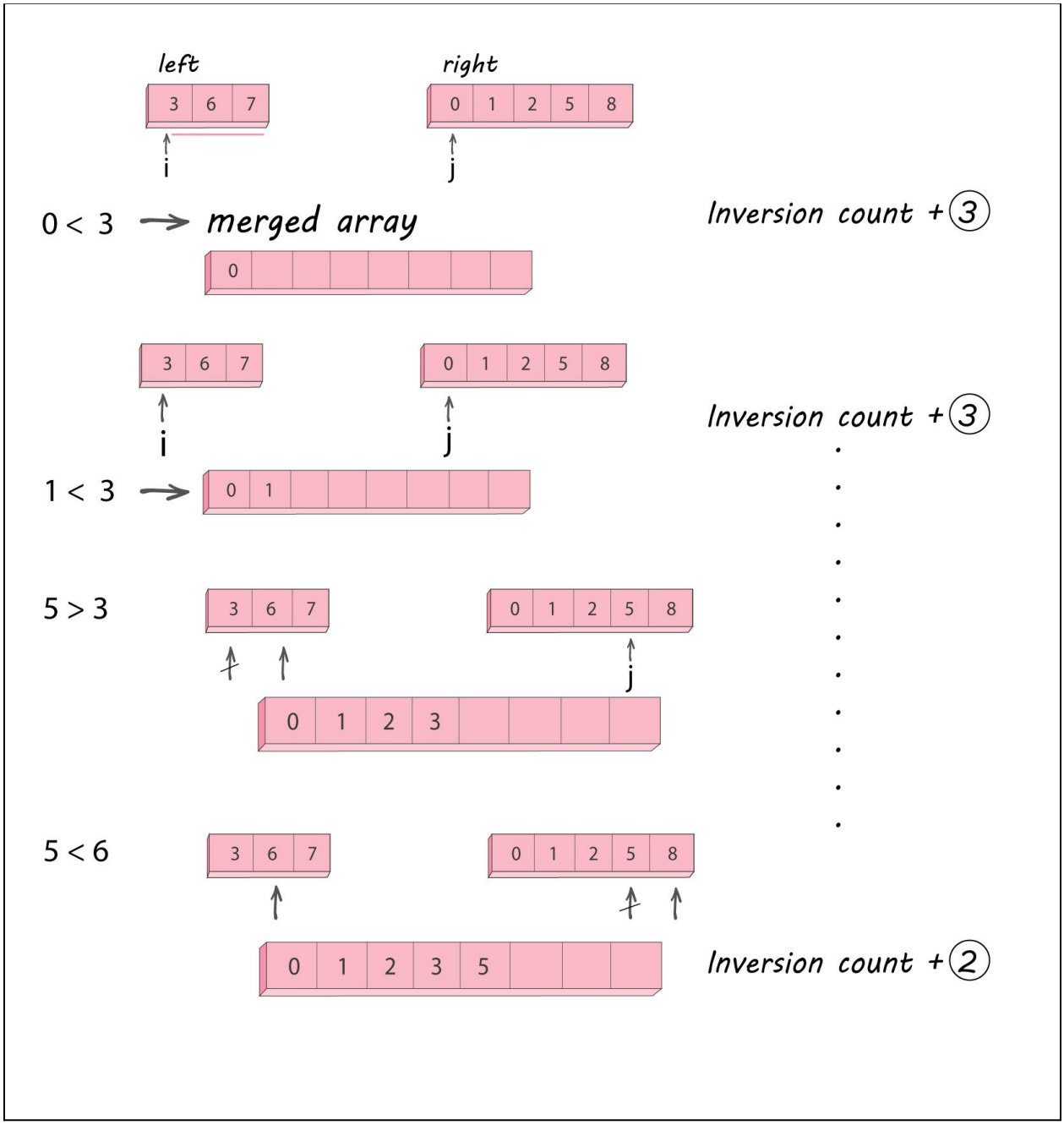
Approach 3: If we add a slight modification to Approach 2 that before merging if we sort the array then it will be easy for us to calculate the newly formed inversions without comparing each and every element.

This approach can be divided into the following steps:

Step 1: Divide the array into two parts.

Step 2: Merge Sort left and right arrays + Calculate the inversion count in respective arrays

Step 3: Total Inversion Count = Left inversion count + Right inversion count + New formed inversions during merging



Now, to calculate the inversion counts while merging let us suppose we have **left array = [3, 6, 7] and right array = [0, 1, 2, 5, 8]**

Pointer i and j are at 3 and 0 respectively

First iteration : 0 < 3 therefore, since both arrays are sorted that means 0 will be smaller to every element in the left array so **increase the inversion count by 3**, add 0 to the merged array and move j pointer to 1.
Second iteration : 1 < 3 therefore increase the inversion count by 3, add 1 to the merged array and move j pointer to 2.
Third iteration : 2 < 3 therefore increase the inversion count by 3, add 2 to the merged array and move j pointer to 5.
Fourth iteration : 5 > 3 so now add 3 to the merged array, move i pointer to 6.
Fifth iteration : 5 < 6 therefore increase the inversion count by 2, add 5 to the merged array and move j pointer to 8.
Continue these iterations while(i<mid && j<=right)

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


long long merge(int A[],int left,int mid,int right){

    int i=left,j=mid,k=0;

    int temp[right-left+1];
    long long count = 0;
    while(i<mid && j<=right){
        if(A[i] <= A[j]){
            temp[k++] = A[i++];
        }else{
            temp[k++] = A[j++];
            count += mid - i;
        }
    }
    while(i<mid){
        temp[k++] = A[i++];
    }
```

```
    while(j<=right){
        temp[k++] = A[j++];
    }

    for(int i=left,k=0;i<=right;i++,k++){
        A[i] = temp[k];
    }
    return count;
}
long long merge_sort(int A[],int left,int right){

    long long count = 0;
    if(right > left){
        int mid = (left + right)/2;

        long long countLeft = merge_sort(A,left,mid);
        long long countRight = merge_sort(A,mid+1,right);
        long long myCount = merge(A,left,mid+1,right);

        return myCount + countLeft + countRight;
    }
    return count;

}
long long solve(int A[], int n)
{
    long long ans = merge_sort(A,0,n-1);
    return ans;
}

int main(){
    int A[] = {5,4,2,3,1};
    cout << solve(A,5);
    return 0;
}
```

Advanced Recursion

Recursion and Strings

Here, we are going to discuss the different problems that can be solved using recursion on strings:

- **Finding the length of a String**

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

int length(char s[]) {
      if (s[0] == '\0') {      // Base case
            return 0;
      }
      int smallStringLength = length(s + 1);  // Recursive call
      return 1 + smallStringLength;           //Small calculation
}


int main() {
      char str[100];
      cin >> str;

      int l = length(str);
      cout << l << endl;
}
```

- **Remove X from a given string**

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

void removeX(char s[]) {
      if (s[0] == '\0') {                 // Base case
            return;
      }

      if (s[0] != 'x') {                  // Small calculation
            removeX(s + 1);               // Recursive call
      } else {
            int i = 1;
            for (; s[i] != '\0'; i++) {
                  s[i - 1] = s[i];
            }
            s[i - 1] = s[i];
            removeX(s);
      }
}

int main() {
      char str[100];
      cin >> str;
      removeX(str);
      cout << str << endl;
}
```

Merge Sort

Consider the example below that shows the working of merge sort over the given array and also how it divides and conquers the array.

Let's now look at the algorithm associated with it...

Algorithm for Merge Sort:

Merge sort keeps on dividing the list into equal halves until it can no longer be divided. By definition, if there is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists, keeping the new list sorted too.

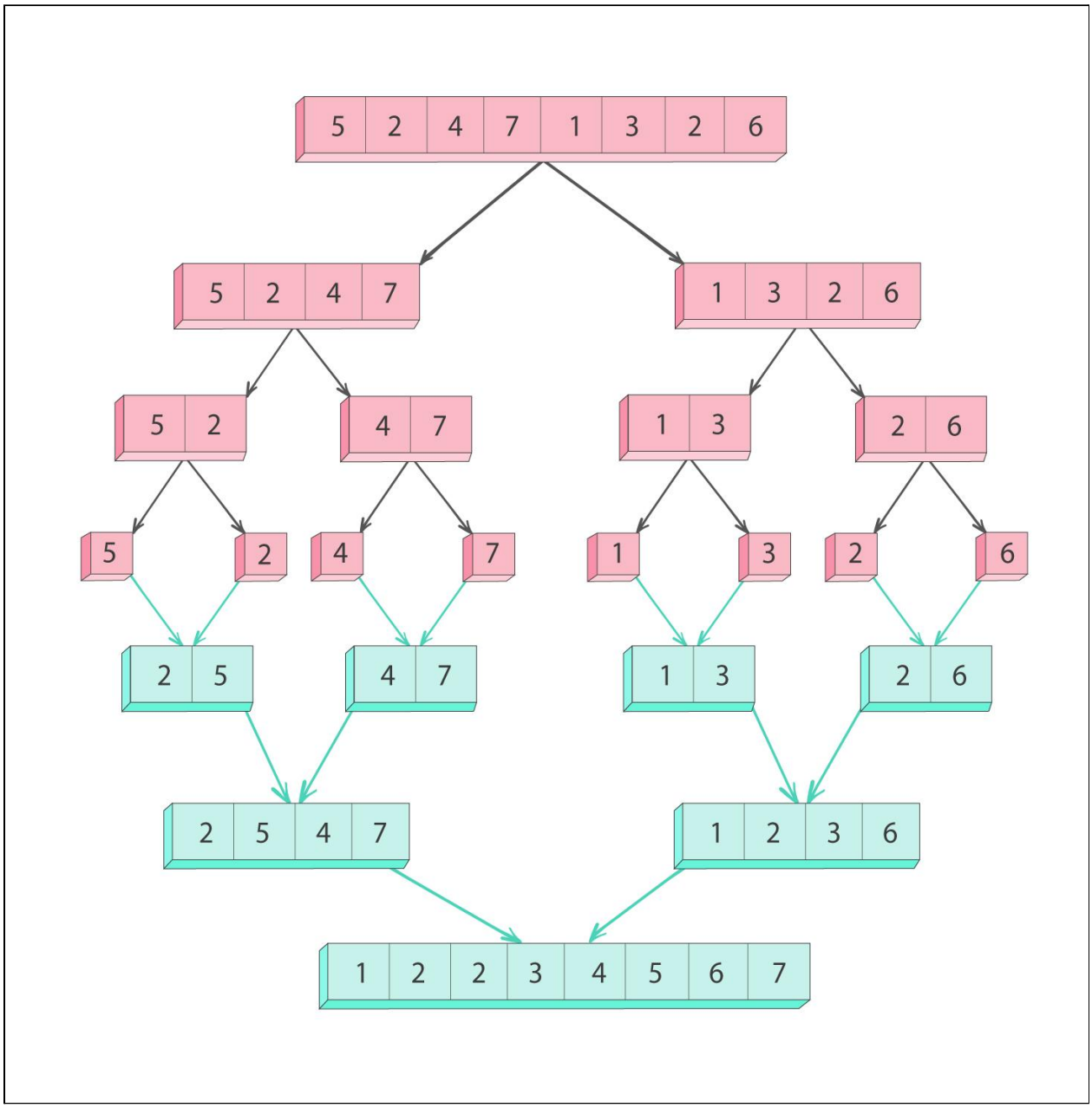
Step 1 – If it is only one element in the list it is already sorted, return.

Step 2 – Divide the list recursively into two halves until it can no longer be divided.

Step 3 – Merge the smaller lists into new lists in sorted order.

Now, you can code it easily by following the above three steps.

It has just one disadvantage and it is that it's not an in-place sorting technique, which means it creates a copy of the array and then works on that copy.

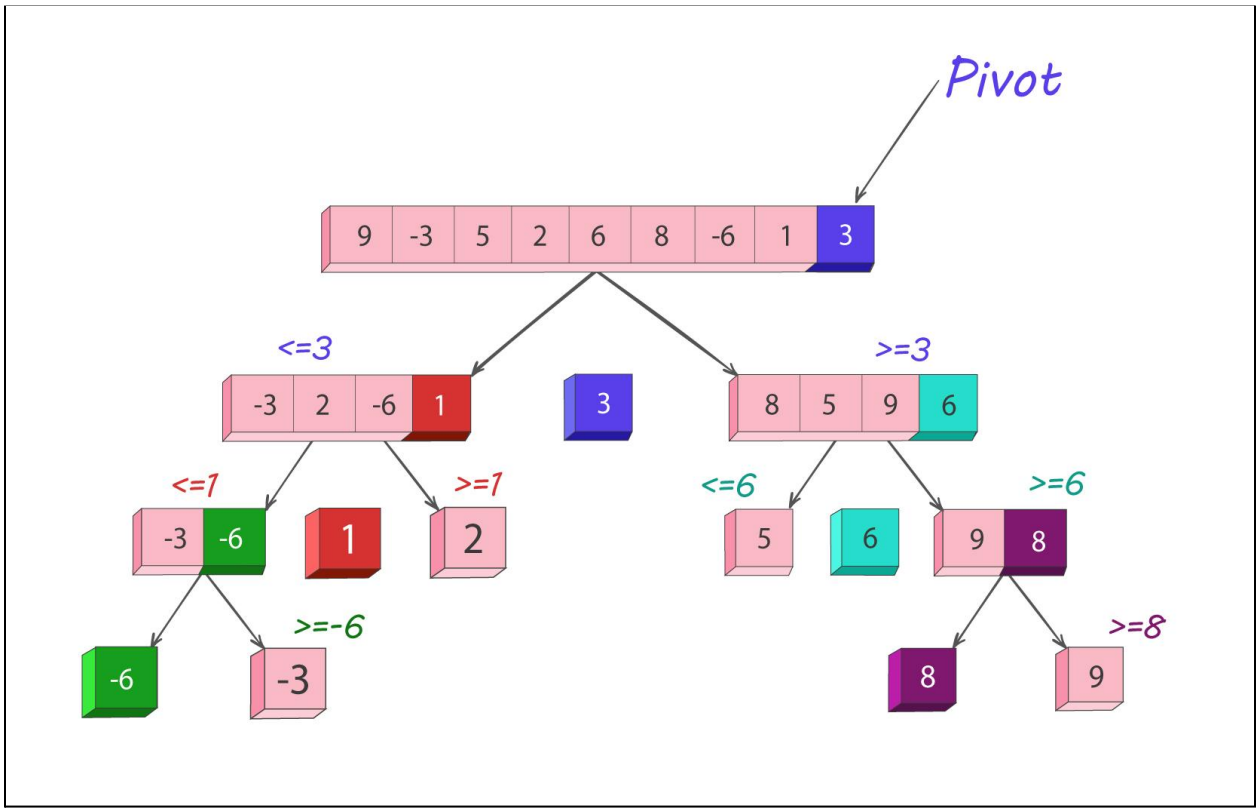


Time Complexity : O(Nlog(N))
Extra Space Complexity : O(N)

Quick Sort

Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it, such that: Left side of pivot contains all the elements that are less than the pivot element and Right side contains all elements greater than the pivot.
It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in an array results in an improved time complexity in most of the cases.
To get a better understanding of the approach consider the following example where a pictorial representation of the quick sort on an array is shown.

Algorithm for Quick Sort:
On the basis of Divide and Conquer approach, quicksort algorithm can be explained as:
Divide
The array is divided into subparts, taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right side.
Conquer
The left and right sub-parts are again partitioned by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.
Combine
This part does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.
Advantage of quicksort over merge sort is that it does not require any extra space to sort the given array, therefore, it is an in-place sorting technique.



Time Complexity : O(Nlog(N))

Extra Space Complexity : O(1)

Strings

Let us think of string as a class, instead of a data type. We all already know that strings are the character arrays that end with a NULL character itself.

Syntax for declaring a string:
string str;

Syntax for declaring a string dynamically:
string *str = new string;

To take input of the string, we can use the **getline()** function. Newline is the delimiter for the getline() function. Follow the syntax below:
getline(cin, str); // where str is the name of the string

You can treat the string like any other character array as well. You can go to any specific index (indexing starts from 0), assign any string value by using equals to (=) operator, etc.

To concatenate two strings you can use the '+' operator. For example, to concatenate two strings s1 and s2 and put it in another string str, syntax is as below:
string str = s1 + s2;

To calculate the length of the string you can use the in-built function (.length()) for that:
int len = str.length();

You can do the same using the following syntax also:
int len = str.size();

To extract a particular segment of the string, we can use the .substr() function.
string s = str.substr(beginning_index, length_ahead_of_starting_index);

Here, if you omit the second argument, then automatically the function takes the complete string, after the specified starting index.
You can also find any particular substring in the given string using .find() function. It will return the starting index of the substring to be found in the given string.
int index = str.find(name_of_the_pattern_to_be_checked);

Note: .find() function finds the first occurring index of the given pattern.

Return Subsequences of a String

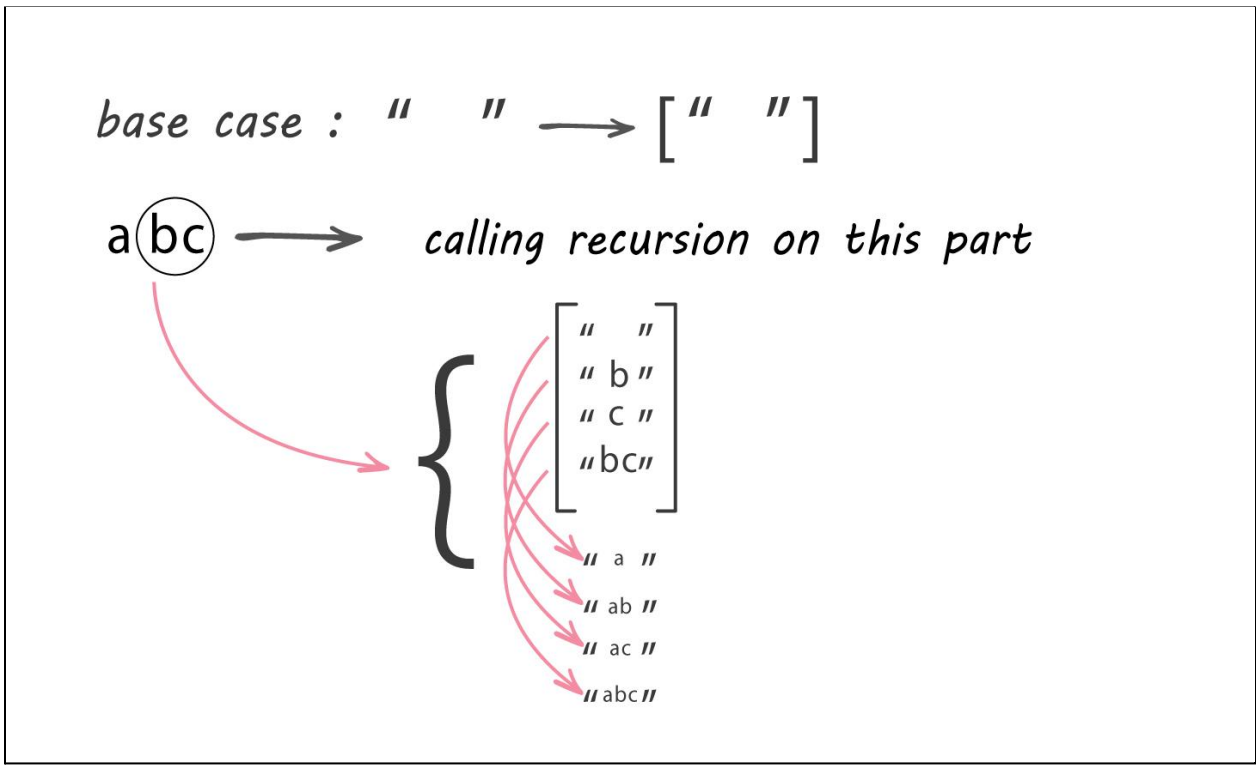
A subsequence can be any combination of the characters present in the string, may these characters be next to each other or far apart doesn't matter.

For example, in string = **"abc"**, we have the following subsequences:

" ", ac, a, b, c, ab, abc, bc

If the length of the string is n, then the total number of substrings that can be generated from it are $\mathbf{2^n}$.

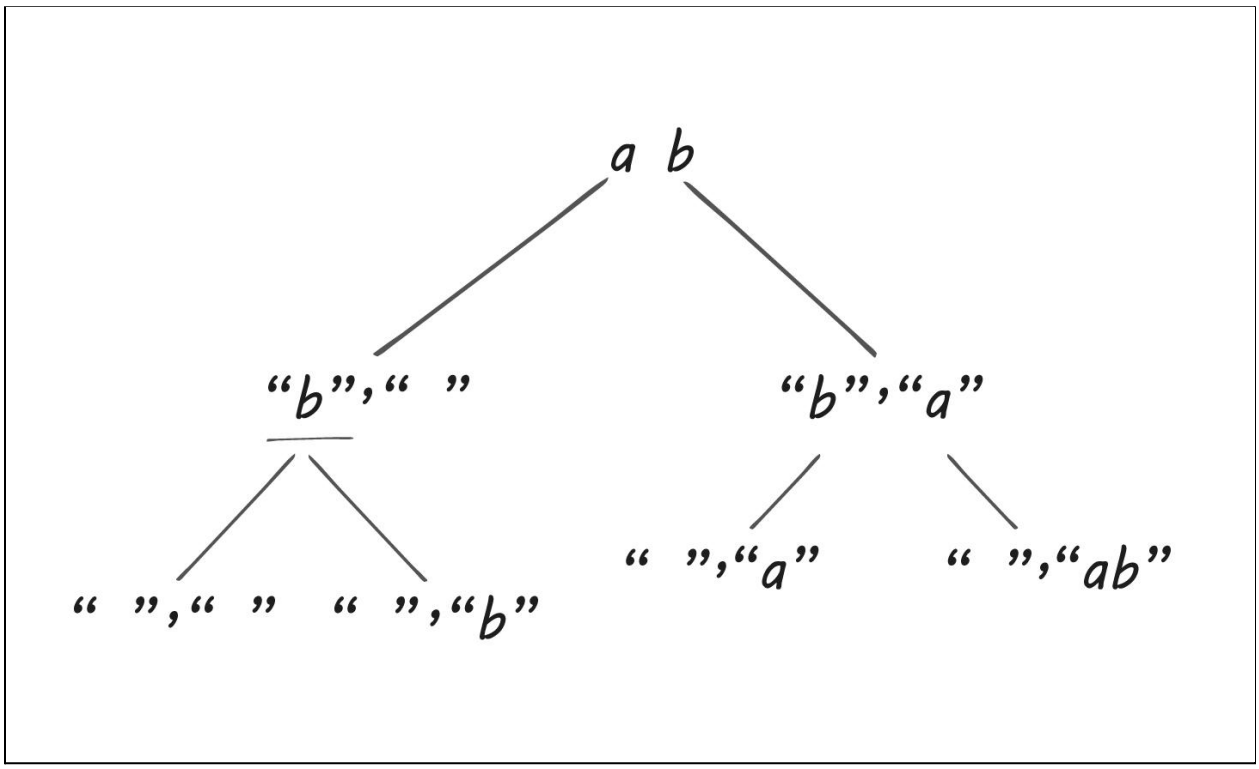
We'll call recursion on **bc** and then in front of all the substrings we add **a** to generate all the substrings inclusive of **a**.



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

int subs(string input, string output[]) {

      // base case

      if (input.empty()) {
            output[0] = "";
            return 1;
      }

       // calling recursion in smaller part

      string smallString = input.substr(1);
      int smallOutputSize = subs(smallString, output);

      // calculation

      for (int i = 0; i < smallOutputSize; i++) {
            output[i + smallOutputSize] = input[0] + output[i];
      }
      return 2 * smallOutputSize;
}

int main() {
      string input;
      cin >> input;
      string* output = new string[1000];
      int count = subs(input, output);
      for (int i = 0; i < count; i++) {
            cout << output[i] << endl;
      }
}
```

Print Subsequences in a different way

String = "**abc**"
Earlier we were calling recursion on **bc** and then calculating corresponding subsequences in which **a** was inclusive.
Now, for every step we go in two different directions and at every step we decide whether to include a character in the output string or not. So basically we hit the base case $\mathbf{2^n}$ times.




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

void print_subs(string input, string output) {
    if (input.length() == 0) {
        cout << output << endl;
        return;
```

```
    }

    print_subs(input.substr(1), output);
    print_subs(input.substr(1), output + input[0]);
}

int main() {
    string input;
    cin >> input;
    string output = "";
    print_subs(input, output);
}
```

Backtracking

Introduction

Backtracking is basically an approach to explore the paths that a problem can follow. It is implemented via recursion.
Thus, we can define backtracking as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

For example, we have 3 bags. Your friend has put a ball inside one of these bags, and has tasked you with finding the bag that contains the ball.

The idea of backtracking suggests that we explore all the possibilities to find the ball. We first look for the ball in bag 1. If the ball is found, all is well and good. If not, we now look for the ball in bag 2. We continue this process till the point where we either find the ball, or we have checked all the bags.

Why Backtracking?

For every problem with clear and well-defined constraints on any solution, Backtracking incrementally builds a candidate to the solution and abandons the candidate (" begins backtrack") as soon as it determines that the candidate can not be completed to a correct solution. Most problems can be solved using other algorithms like Greedy Algorithms, Dynamic Programming in O(Nlog(N), O(N) and O(log(N)) time complexity, and hence outshine the backtracking algorithm (generally, backtracking algorithms have exponential

time and space complexity). However, for a few problems, we only have the backtracking algorithms to solve until now.

Let's discuss a problem to understand the benefit of backtracking approach:

The Knight's Tour

Problem Statement:
Given a N*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess, knights must visit each square exactly once. Print the order of each cell in which they are visited.

Example:
Input :
N = 8
Output:

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Naive Approach: Generate all paths one by one and check if the generated path satisfies the constraints. That is, while there are untried paths, generate

the next path and if this path covers every square on the chess board then print this path.

Backtracking Approach: Backtracking incrementally attacks a problem.

- We start with an empty solution vector and add Knight's move(item) one by one.
- We check whether the current move violates the problem constraint, by using a recursion call.
- If recursion tells us that the solution is not valid, then we remove the previous move(item) and try an alternative move(item).
- If none of the alternative moves(items) work then we go to the previous stage and remove the Knight's move(item) added in the previous stage.
- If we reach the initial stage, that means there is no solution, However if the solution vector becomes complete then we print the solution.

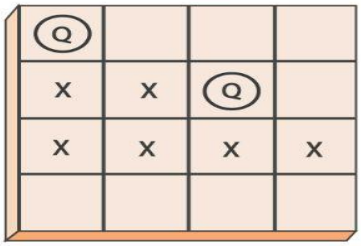
N- Queen Problem

Problem Statement: We are given an **N X N Matrix** and we have to place N Queens in it such that no Queens cross each other .
A Queen can move in all eight directions.

Explanation:
Applying Backtracking:

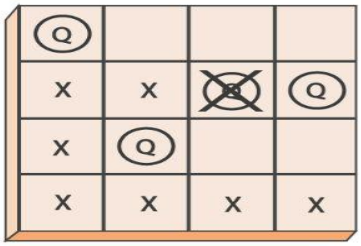
1. **Place yourself at the start point.**
2. **Explore all the paths possible from that starting point.**
3. **Insert your element at only correct positions and once inserted move on to a smaller problem.**
4. **Once you reach the base case, print your solution.**
5. **Once you explore all the paths from one point , backtrack to your previous point.**

N-Queen Problem:-



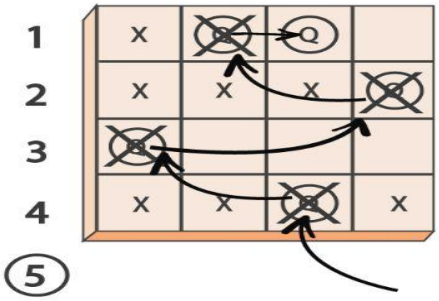
Possible Position

No, Queen can be inserted in 3rd row,
So Backtrack



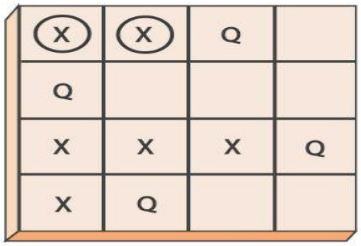
Again Backtrack x 2

x x x x



row > N ∴ i can print
this solution

⇒



This is also a solution
then again backtrack

Code:

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

using namespace std;

int board[11][11];

bool isPossible(int n,int row,int col){

// Same Column
  for(int i=row-1;i>=0;i--){
    if(board[i][col] == 1){
      return false;
    }
  }
//Upper Left Diagonal
  for(int i=row-1,j=col-1;i>=0 && j>=0 ; i--,j--){
    if(board[i][j] ==1){
      return false;
    }
  }

  // Upper Right Diagonal

  for(int i=row-1,j=col+1;i>=0 && j<n ; i--,j++){
    if(board[i][j] == 1){
      return false;
    }
  }

  return true;
}
void nQueenHelper(int n,int row){
  if(row==n){
    // We have reached some solution.
    // Print the board matrix
    // return

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

```
      for(int j=0;j<n;j++){
        cout << board[i][j] << " ";
      }
    }
    cout<<endl;
    return;

  }

  // Place at all possible positions and move to smaller problem
  for(int j=0;j<n;j++){

    if(isPossible(n,row,j)){
      board[row][j] = 1;
      nQueenHelper(n,row+1);
      board[row][j] = 0;
    }
  }
  return;

}
void placeNQueens(int n){

  memset(board,0,11*11*sizeof(int));

  nQueenHelper(n,0);

}
int main(){

  placeNQueens(4);
  return 0;
}
#include <bits/stdc++.h>

using namespace std;

int board[11][11];

bool isPossible(int n,int row,int col){
```

```
// Same Column
  for(int i=row-1;i>=0;i--){
    if(board[i][col] == 1){
      return false;
    }
  }
//Upper Left Diagonal
  for(int i=row-1,j=col-1;i>=0 && j>=0 ; i--,j--){
    if(board[i][j] ==1){
      return false;
    }
  }

  // Upper Right Diagonal

  for(int i=row-1,j=col+1;i>=0 && j<n ; i--,j++){
    if(board[i][j] == 1){
      return false;
    }
  }

  return true;
}
void nQueenHelper(int n,int row){
  if(row==n){
    // We have reached some solution.
    // Print the board matrix
    // return

    for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
        cout << board[i][j] << " ";
      }
    }
    cout<<endl;
    return;

  }
```

```
  // Place at all possible positions and move to smaller problem
  for(int j=0;j<n;j++){

    if(isPossible(n,row,j)){
      board[row][j] = 1;
      nQueenHelper(n,row+1);
      board[row][j] = 0;
    }
  }
  return;

}
void placeNQueens(int n){

  memset(board,0,11*11*sizeof(int));

  nQueenHelper(n,0);

}
int main(){

  placeNQueens(4);
  return 0;
}
```

Rat In A Maze Problem

Problem Statement: We are given an **N X N Matrix** and we have a starting point and a destination point. There is a rat who has to reach the destination point from the starting point through a possible path and we have to print this path.

Matrix =

1	**1**	**0**
1	**1**	**1**
1	**1**	**1**

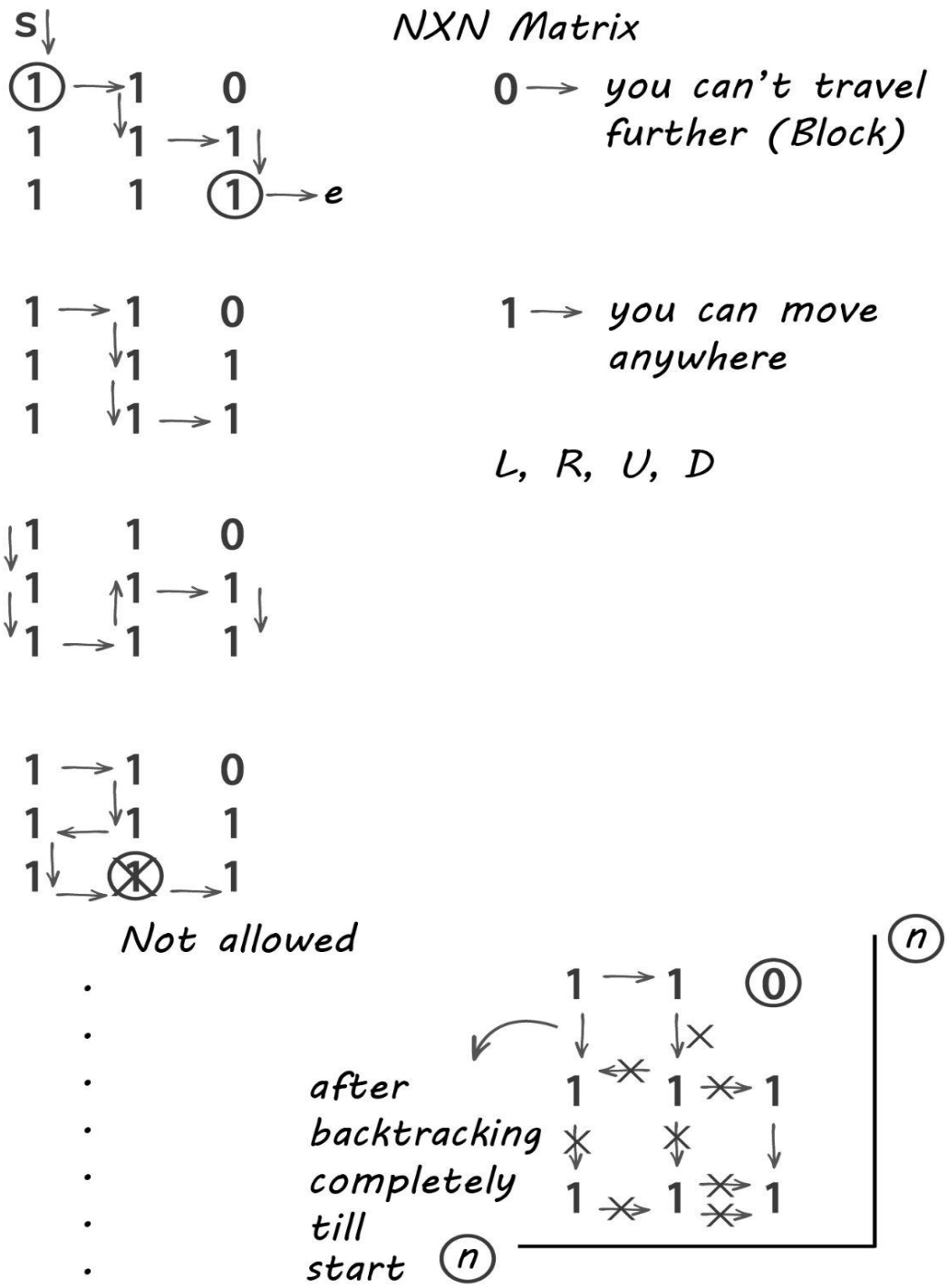
0 means that the rat can't travel further.
1 means we can move anywhere from that point, that is, left, right, up and down.

Explanation:

1. **Create a solution matrix, initially filled with 0's.**
2. **Create a recursive function, which takes the initial matrix, output matrix and position of rat (i, j).**
3. **if the position is out of the matrix or the position is not valid then return.**
4. **Mark the position output[i][j] as 1 and check if the current position is destination or not. If the destination is reached, print the output matrix and return.**
5. **Call recursion at position (i+1, j) and (i, j+1).**
6. **Unmark position (i, j), i.e output[i][j] = 0.**

CODING NINJAS

Rat in a Maze :-



Code:

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


void printSolution(int** solution,int n){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cout << solution[i][j] << " ";
        }
    }
    cout<<endl;
}
void mazeHelp(int maze[][20],int n,int** solution,int x,int y){


    if(x == n-1 && y == n-1){
        solution[x][y] =1;
        printSolution(solution,n);
        solution[x][y] = 0;
        return;
    }
    if(x>=n || x<0 || y>=n || y<0 || maze[x][y] ==0 || solution[x][y] ==1){
        return;
    }
    solution[x][y] = 1;
    mazeHelp(maze,n,solution,x-1,y);
    mazeHelp(maze,n,solution,x+1,y);
    mazeHelp(maze,n,solution,x,y-1);
    mazeHelp(maze,n,solution,x,y+1);
    solution[x][y] = 0;
}
void ratInAMaze(int maze[][20], int n){
  int** solution = new int*[n];
  for(int i=0;i<n;i++){
    solution[i] = new int[n];
  }
  mazeHelp(maze,n,solution,0,0); }
```

Modulo Arithmetic

Introduction

Modulo (%) operation: This operation returns us the remainder of the division. Example: 17 % 5 = 2. The modulo of a positive number **n** ranges from **0 to n - 1**. In many questions we are required to take **%(10^9 + 7)**. This is because the maximum positive number that we can store in long long is **(2^{64} - 1)**. But the solution might demand to multiply two numbers and their product may be greater than **(2^{64} - 1)**. **For example : 2^{40} * 2^{32} =2^{72}**.

%(10^9 + 7) : This specific number is chosen because:

1. It is a very large number.
2. It is a prime number.

Properties

Modulo arithmetic has the following properties:

1. **(a + b) % c = ((a % c) + (b % c)) % c**
2. **(a * b) % c = ((a % c) * (b % c)) % c**
3. **(a - b) % c = ((a % c) - (b % c)) % c**
4. **(a / b) % c = ((a % c) / (b % c)) % c**, given that a Multiplicative Modulo Inverse should exist.

The major use of modulo arithmetic is to prevent range overflow at every step, that is , in case the operations on any of the operands in the solution cause the operand to go out of range of its data type, then taking modulo at that very step makes sure that our program/code works correctly.

Now, **if (x % m) = -ve** , then we change the formula to **(x%m + m) % m**.

Number of Balanced BTs

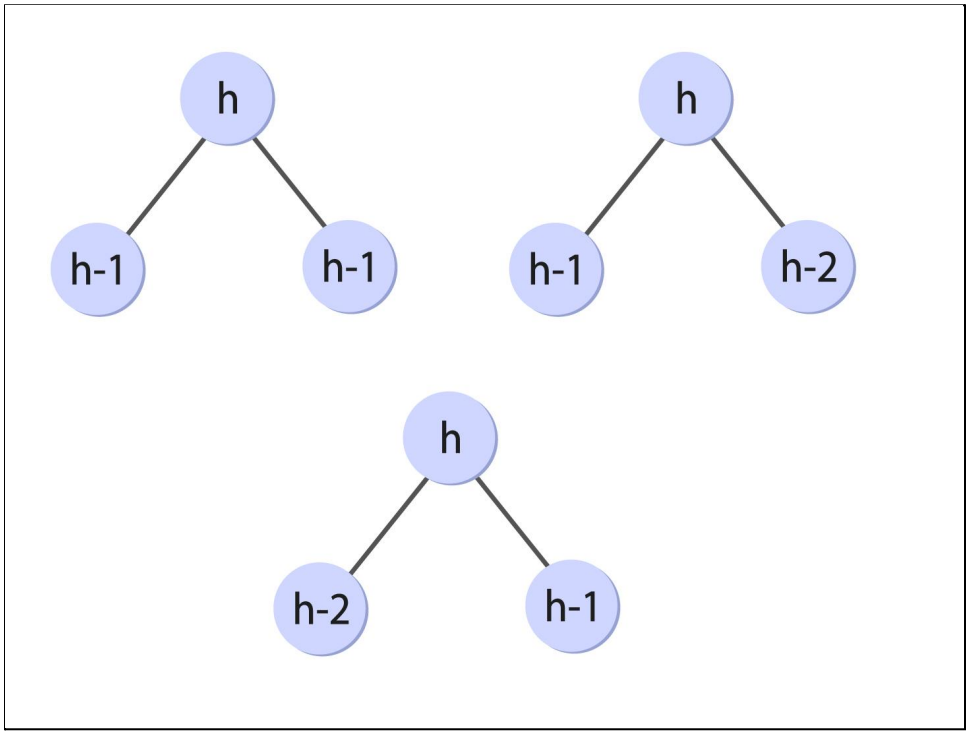
Problem Statement: Given height = **h**. Find the number of balanced binary trees possible.

Explanation:

Binary Tree: A generic tree with at most two child nodes for each parent node is known as a binary tree.
A binary tree is made of nodes that constitute a **left** pointer, a **right** pointer, and a **data** element. The **root** pointer is the topmost node in the tree. The left and right pointers recursively point to smaller subtrees on either side. A null pointer represents a binary tree with no elements, i.e., an empty tree. A formal definition is: a binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

For a **Balanced Binary Tree**, **| lh - rh | <= 1** , that is the difference between left height and right height must be less than or equal to 1 **for every node**.

Possibilities: Suppose there are **x** number of trees for **h - 1** height, **y** for **h -2** number of height. Therefore we have three possibilities:



Total possibility : x*x + x*y + x*y

Code:

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

using namespace std;


int balancedBTs(int h){

    if(h==0 || h==1){
        return 1;
    }

    int m = pow(10,9) + 7;
    int x = balancedBTs(h-1);
    int y = balancedBTs(h-2);
```

```
    long res1 = (long)x*x;
    long res2 = (long)x*y*2;

    int ans1 = (int)(res1%m);
    int ans2 = (int)(res2%m);

    int ans = (ans1+ans2)%m;

    return ans;
}

int main(){

    int h=8;
    cout << balancedBTs(h) << endl;
    return 0;
}
```

Ad hoc Problems

Introduction

Ad hoc means for one specific cause or approaching a solution in an unplanned way. In programming, it means working in bits and pieces. For example, there are 2 programmers 1 and 2. They have to complete 4 modules, say W, X, Y ,Z. If there is a plan then they can decide on which module to work on, but in an ad hoc approach they can work in an unplanned manner.

Live Problem 1: Equalize

Link : https://codeforces.com/contest/1037/problem/C

Problem Statement:

You are given two binary strings **a** and **b** of the same length. You can perform the following two operations on the string a:

Swap any two bits at indices i and j respectively ($1 \le i,j \le n$), the cost of this operation is $|i-j|$, that is, the absolute difference between i and j.

Select any arbitrary index i ($1 \le i \le n$) and flip (change 0 to 1 or 1 to 0) the bit at this index. The cost of this operation is 1.

Find the minimum cost to make the string a equal to b. It is not allowed to modify string b.

Explanation:

Agenda: Find the minimum cost of converting string a to string b.

- Cost of flipping a bit: **O(1)**
- Cost of swapping bits from at i and j: **O(|i-j|)**

A = 1 0 0 and B = 0 0 1, if we swap the first and last bit of A that makes A equal to B with a cost of 2 - 0 = **2**. If we had flipped the first and last bits, then our cost would have been 2.

But considering **S = 1 0 1 0 and T = 1 1 0 1**, if **S[i] == T[i]**, then we do nothing at those bits. Now, flipping costs us only 1 and swapping to longer positions will cost more. For example, if for any string we have to change the bits at position 0 and 3, by flipping the two, our cost will only be **2** but if we use a swap operation, then our cost will be **3 - 0 = 3**.

For positions next to each other, the swap operation is a better choice because it will cost us **(i + 1) - i = 1**, but if we flip the individual bits then our cost will be **1 + 1 = 2**. So we swap only with the next element, that is if **S[i + 1] != T[i + 1] && S[i + 1] != S[i]**, then we swap.

Code:

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


int main(){

    int n;
    cin >> n;
    string s, t;
    cin >>s >> t ;
    int cost =0;
    for(int i=0; i<n; ){
        if(s[i] != t[i]){
            //  either swap or flip and increment the cost
            if(i+1 < n && s[i] != s[i+1] && s[i+1] != t[i+1]){
                i+=2;
            } else {
                i+=1;
            }
            cost++;

        } else {
            i++;
```

```
        }
    }
    cout << cost;
    return 0;
}
```

Live Problem 2: Rectangular Area

Problem Statement:

You are given N rectangles, which are centered in the center of the Cartesian coordinate system and their sides are parallel to the coordinate axes. Each rectangle is uniquely identified with its width (along the x-axis) and height (along the y-axis). Navdeep has coloured each rectangle in a certain colour and now wants to know the area of the coloured part of the paper. Please refer to the sample test case 1 and image used in it for better understanding.

Explanation:

Sample Input 1 :

3

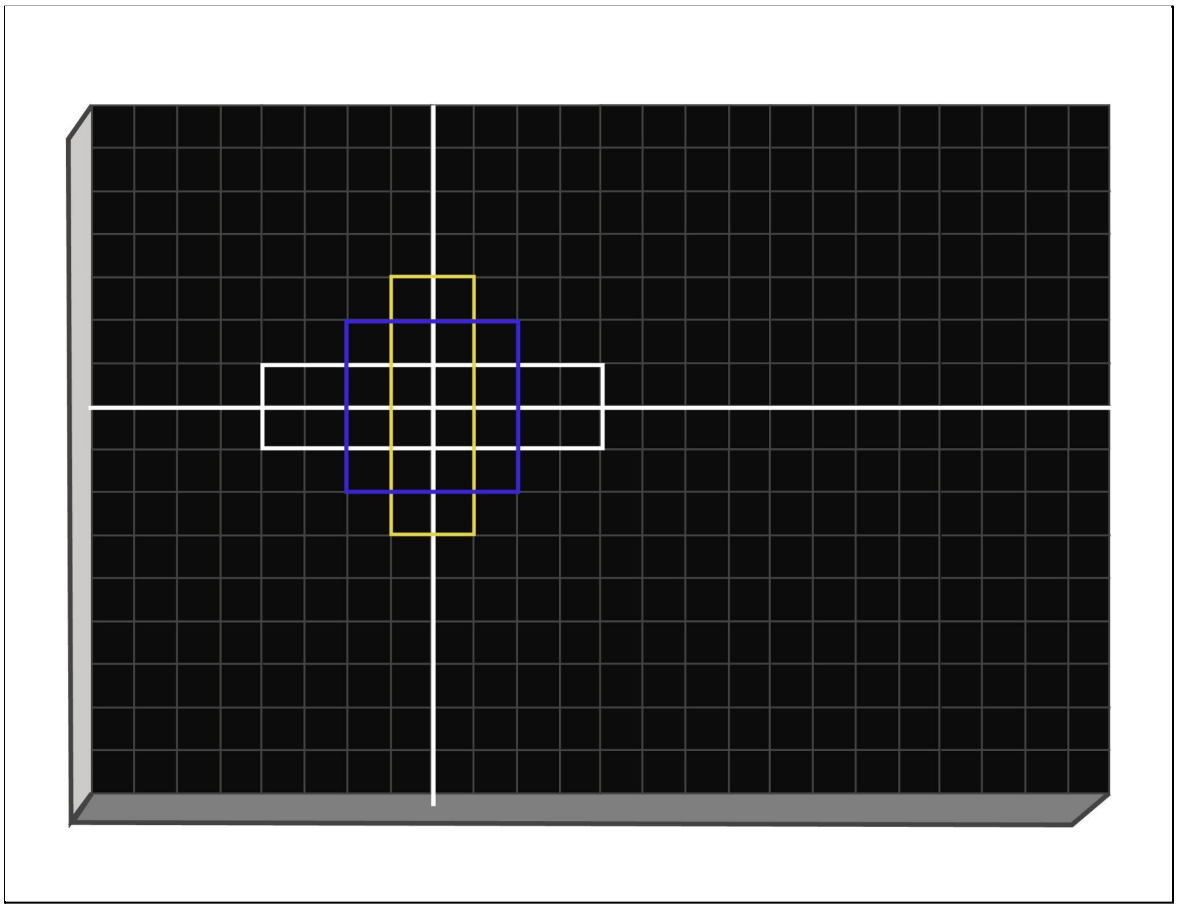
8 2 (represented by white coloured rectangle)

4 4 (represented by blue coloured rectangle)

2 6 (represented by yellow coloured rectangle)

Sample Output 1 :

28



In other words, he wants to know the number of unit squares that belong to at least one rectangle. This can also be seen in the image. There are 28 unit squares, which come in the bounds of at least one rectangle.

We need not calculate the area of the entire figure, rather we start from the rightmost part of the figure and start calculating the area according to the height. Once we reach the origin we can multiply our answer by 2 to calculate the area of the entire figure.

Steps:

1. Create a vector that stores the heights.
2. Start from the rightmost point and iterate towards the left one unit at a time till you reach the origin.

3. If height of the left point/ unit is less than the previous right one, then height[i] = height[i+1]

Code:

```
/*
    Time complexity: O(N + M)
    Space complexity: O(M)
    where N is the number of rectangles
    and M is the range of X coordinates of the rectangle
*/
#include <bits/stdc++.h>
using namespace std;
long long int rectArea(pair<int, int> arr[], int n) {
    int height[100002];
    for (int i = 0; i < 100002; i++) {
        height[i] = 0;
    }
    for (int i = 0; i < n; i++) {
        height[arr[i].first / 2] = max(height[arr[i].first / 2],arr[i].second);
    }
    long long int ans = 0;
    for (int i = 100000; i > 0; i--) {
        height[i] = max(height[i], height[i + 1]);
        ans += height[i];
    }
    return ans * 2;
}
int main() {
    int n;
    cin >> n;
    int a, b;
    pair<int, int> arr[n];
    for (int i = 0; i < n; i++) {
        cin >> a >> b;
        arr[i] = make_pair(a, b);
    }
    cout << rectArea(arr, n) << endl;
}
```

Dynamic Programming - 1

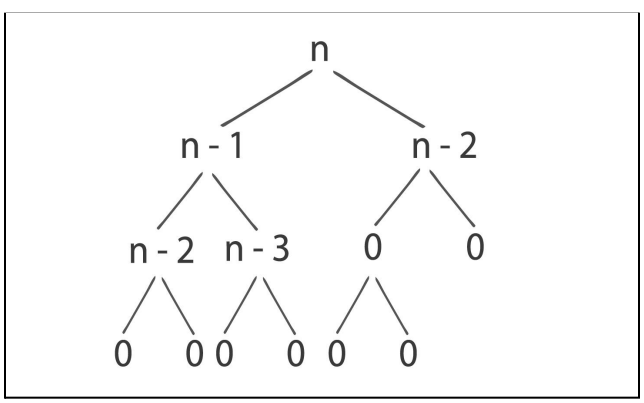
Introduction

Suppose we need to find the nth Fibonacci number using recursion that we have already found out in our previous sections. Let's directly look at its code:

```
int fibo(int n) {
    if(n <= 1) {
        return n;
    }
    int a = fibo(n - 1);
    int b = fibo(n - 2);
    return a + b;
}
```

Here, for every n, we need to make a recursive call to f(n-1) and f(n-2).

For f(n-1), we will again make the recursive call to f(n-2) and f(n-3). Similarly, for f(n-2), recursive calls are made on f(n-3) and f(n-4) until we reach the base case. The recursive call diagram will look something like shown below:

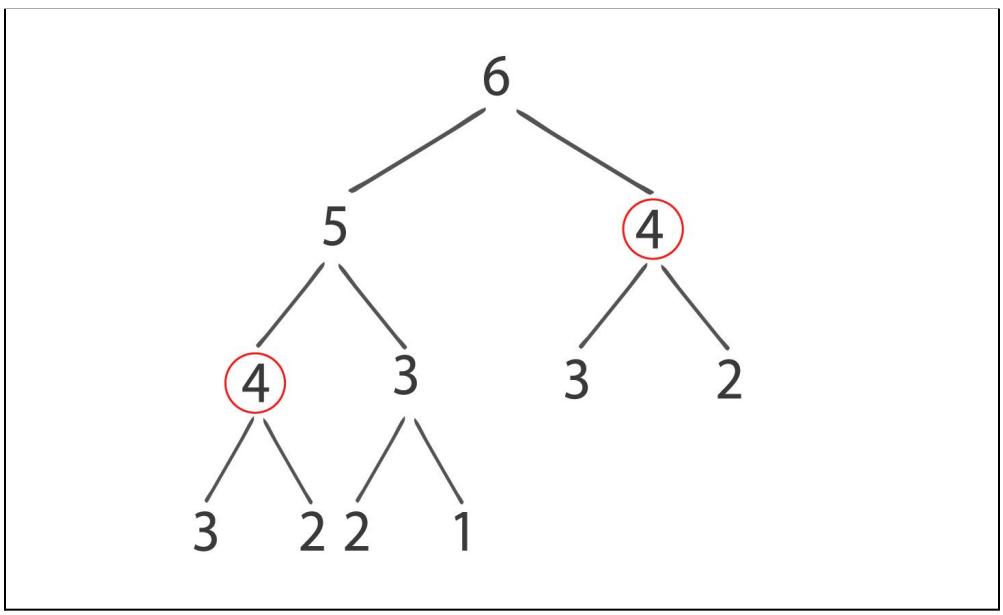


At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one). At every level, we are doing $2^n K$ work (where n = 0, 1, 2, ...). Since reaching 1 from n will take n calls, therefore, at the last level, we are doing $2^{n-1}k$ work. Total work can be calculated as:

$$(2^0 + 2^1 + 2^2 + \ldots + 2^{n-1}) * k \simeq 2^n k$$

Hence, it means time complexity will be $O(2^n)$.

We need to improve this complexity as it is terrible. Let's look at the example below for finding the 6th Fibonacci number.



IMPORTANT OBSERVATION: We can observe that there are repeating recursive calls made over the entire program. As in the above figure, for calculating f(5), we need the value of f(4) (first recursive call over f(4)), and for calculating f(6), we again need the value of f(4)(second similar recursive call over f(4)). Both of these recursive calls are shown above in the outlining circle. Similarly, there are many others for

which we are repeating the recursive calls. Generally, in recursion, there are repeated recursive calls, which increases the time complexity of the program.

To overcome this problem, we will store the output of previously encountered values(preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again. This way, we can improve the running time of our code. This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called **memoization**.

To achieve this in our example we will simply take an answer array initialized to -1. Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not. If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value. After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.

Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most (n+1) only.

Let's look at the memoization code for Fibonacci numbers below:

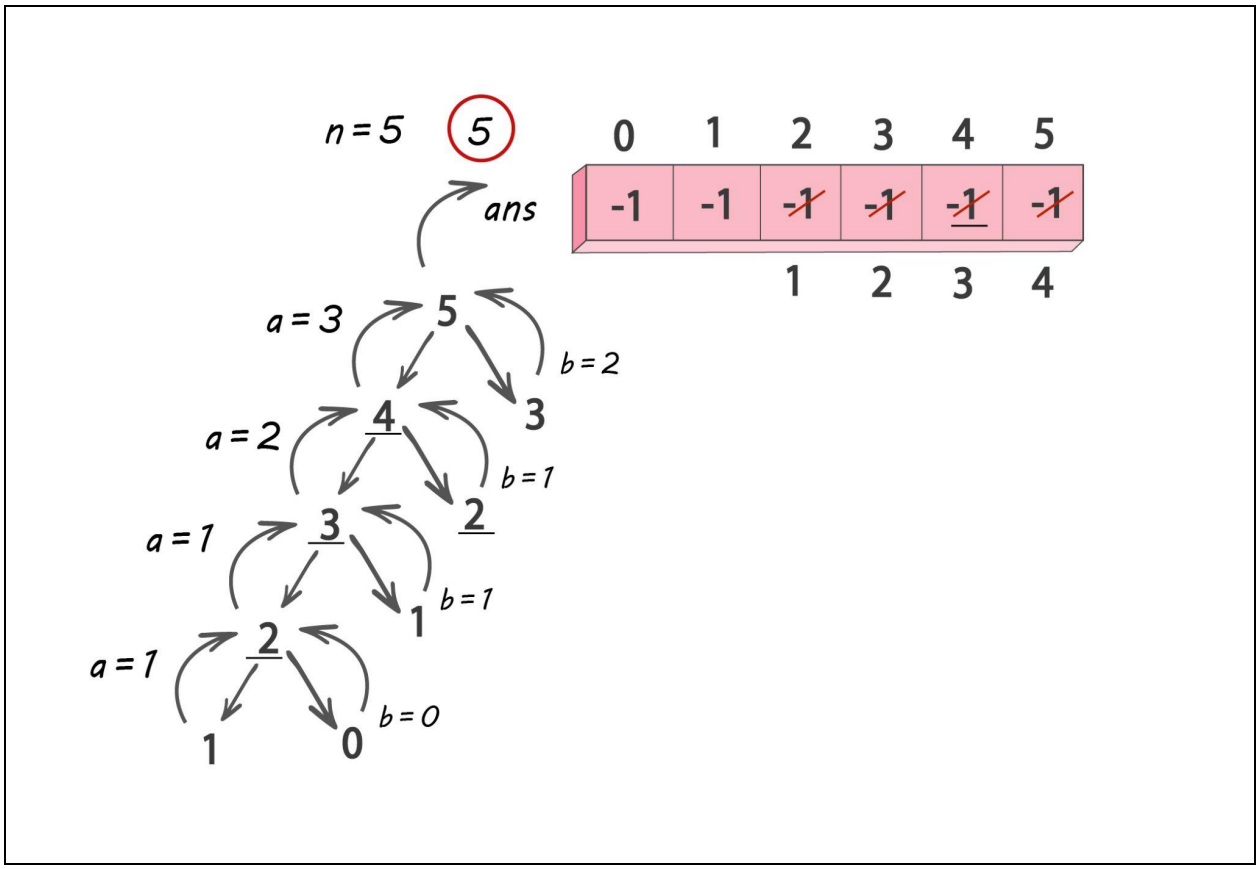
```
int fibo_helper(int n, int *ans) {
    if(n <= 1) {            // Base case
        return n;
    }

    // Check if output already exists
    if(ans[n] != -1) {
```

```
        return ans[n];
    }

    // Calculate output
    int a = fibo_helper(n-1, ans);
    int b = fibo_helper(n-2, ans);

    // Save the output for future use
    ans[n] = a + b;

    // Return the final output
    return ans[n];
}


int fibo_2(int n) {
    int *ans = new int[n+1];

    for(int i = 0; i <= n; i++) {       // -1 marks that answer for
corresponding
        ans[i] = -1;                    // number does not exist
    }
    return fibo_helper(n, ans);         // Now simply following the
memoization
}
```

Let's dry run for n = 5, to get a better understanding:

Again, if we observe, we can see that for any number we are not able to make a recursive call on the right side of it which means that we can make at most 5+1 = 6 (n+1) unique recursive calls which reduce the time complexity to O(n) which is highly optimized as compared to simple recursion.

Summary: Memoization is a **top-down approach** where we



save the previous answers so that they can be used to calculate the future answers and improve the time complexity to a greater extent.

Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored. Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes. In cases of Fibonacci numbers, these indexes are 0 and 1 as f(0) = 0 and f(1) = 1. So we can directly allot these two values to our answer array and then use these to calculate f(2), which is f(1) + f(0), and so on for every other index. This can be simply done iteratively by running a loop from i = (2 to n). Finally, we will get our answer at the 5th index of the answer array as we already know that the i-th index contains the answer to the i-th value.

Simply, we are first trying to figure out the dependency of the current value on the previous values and then using them to calculate our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, following a **bottom-up approach** to reach the desired index. This approach of converting recursion into iteration is known as **Dynamic programming(DP).**

Let us now look at the code for calculating the nth Fibonacci number:

```
int fibo_3(int n) {
    int *ans = new int[n+1];

    ans[0] = 0;    // Storing the independent values in the solution array.
    ans[1] = 1;

    for(int i = 2; i <= n; i++) { //Following bottom-up approach to reach n
        ans[i] = ans[i-1] + ans[i-2];
    }
    int output = arr[n];
    delete [] arr;
    return output;                  // final answer
}
```

Note: Generally, memoization is a recursive approach, and DP is an iterative approach.

For all further problems, we will do the following:

1. Figure out the most straightforward approach for solving a problem using recursion.
2. Now, try to optimize the recursive approach by storing the previous outputs using memoization.

3. Finally, replace recursion by iteration using dynamic programming. (The best possible solution that seems to appear for every problem when in case of recursion the space complexity is more in comparison to iteration)

Alphacode

Problem Statement: Alice and Bob need to send secret messages to each other and are discussing ways to encode their messages:

Alice: *"Let's just use a very simple code: We'll assign 'A' the code word 1, 'B' will be 2, and so on down to 'Z' being assigned 26."*

Bob: *"That's a stupid code, Alice. Suppose I send you the word 'BEAN' encoded as 25114. You could decode that in many different ways!"*

Alice: *"Sure you could, but what words would you get? Other than 'BEAN', you'd get 'BEAAD', 'YAAD', 'YAN', 'YKD' and 'BEKD'. I think you would be able to figure out the correct decoding. And why would you send me the word 'BEAN' anyway?"*

Bob: *"OK, maybe that's a bad example, but I bet you that if you got a string of length 5000 there would be tons of different decodings and with that many you would find at least two different ones that would make sense."*

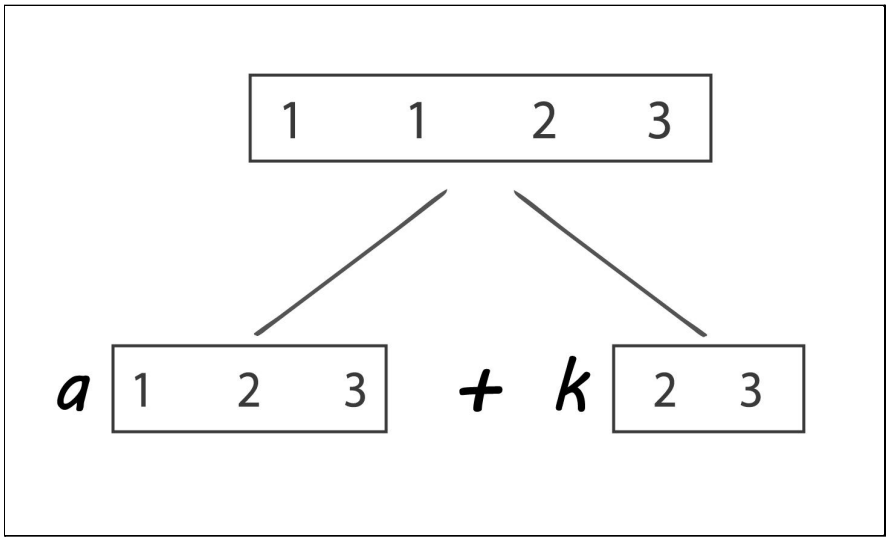
Alice: *"How many different decodings?"*

Bob: *"Jillions!"*

For some reason, Alice is still unconvinced by Bob's argument, so she requires a program that will determine how many decodings there can be for a given string using her code.

Explanation:

Example: Let's take a message **1 1 2 3.** Now this message can be reduced or decoded in two different ways by recursion:



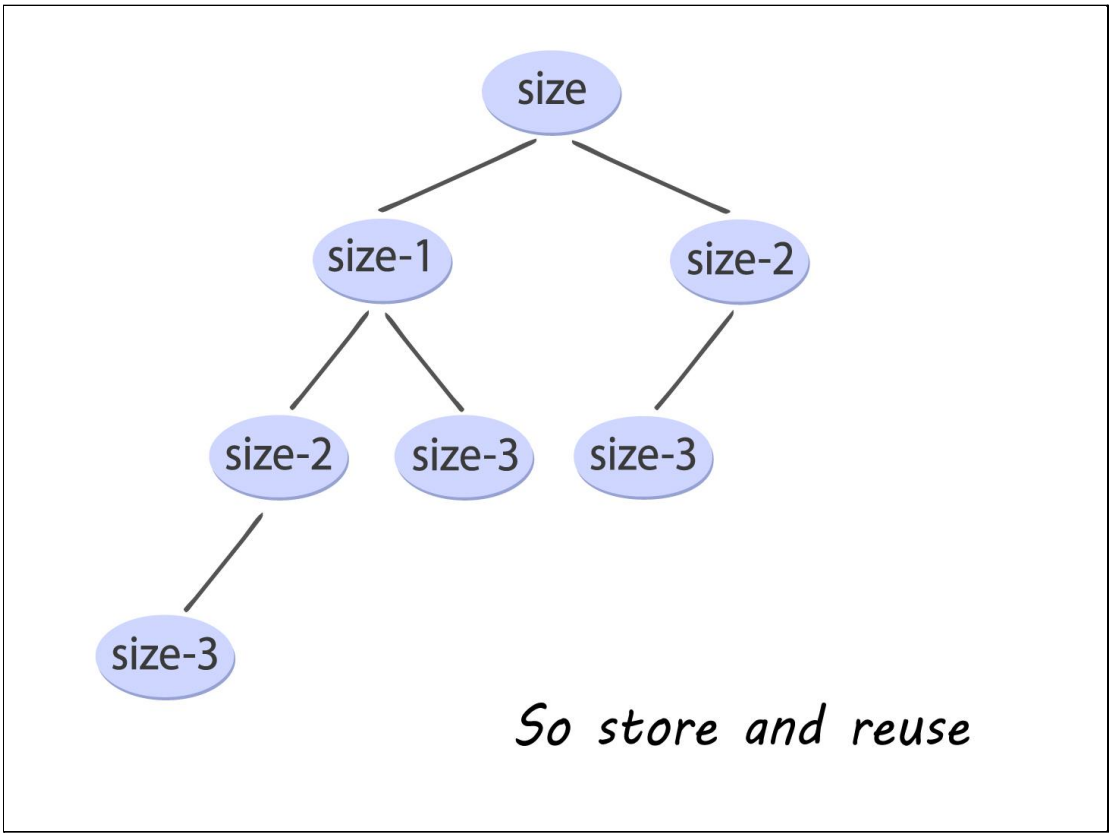
We will always make the first recursive call , but for the second one we check whether the two consecutive digits form a number that is less than or equal to 26.

Now, let's take a look at the code for better understanding:

```
int num_codes(int* n, int size) {
     if (size == 1) {
         return 1;
     }
     if (size == 0) {
         return 1;
     }
     int output = num_codes(n, size - 1);
     if (output[size - 2] * 10 + output[size - 1] <= 26) {
         output += num_codes(n, size -2);
     }
     return output;
}
```

Note: In the above code, we have called recursion from the end rather than start.

Now, we have to check if we can optimize the code that we have written. It can be done using memoization. But, for memoization to apply, we need to check if there are any overlapping sub-problems so that we can store the previous values to obtain the new ones.



Here, if we blindly make two recursive calls at each step, then the time complexity will approximately be $O(2^n)$.

From the above, it is clearly visible that there are repeating sub-problems. Hence, this problem can be optimized using memoization.

```
int num_codes2(int *n, int size, int *arr){

    if(size == 1){
        return 1;
    }
    if(size == 0){
        return 1;
    }
    if(arr[size] > 0){
        return arr[size];
    }
    int output = num_codes(n, size - 1, arr);
    if (output[size - 2] * 10 + output[size - 1] <= 26) {
        output += num_codes(n, size -2, arr);
    }
    arr[size] = output;
    return output;
}
```

Iterative Solution:

We make an array and the ith index of that array will store the number of possible codes if we just consider first i digits.

Code:

```
int num_codes_i(int * input, int size) {
    int* output = new int[size + 1];
    output[0] = 1;
    output[1] = 1;

    for (int i = 2; i <= size; i++) {
        output[i] = output[i - 1];
        if (input[i-2] *10 + input[i - 1] <= 26) {
            output[i] += output[i - 2];
        }
    }
    int ans = output[size];
```

```
    delete [] output;
    return ans;
}
```

Longest Increasing Subsequence

Problem Statement: You are given an array of positive integers as input. Write a code to return the length of the largest such subsequence in which the values are arranged first in strictly ascending order and then in strictly descending order.

Such a subsequence is known as bitonic subsequence. A purely increasing or purely decreasing subsequence will also be considered as a bitonic sequence with the other part empty.

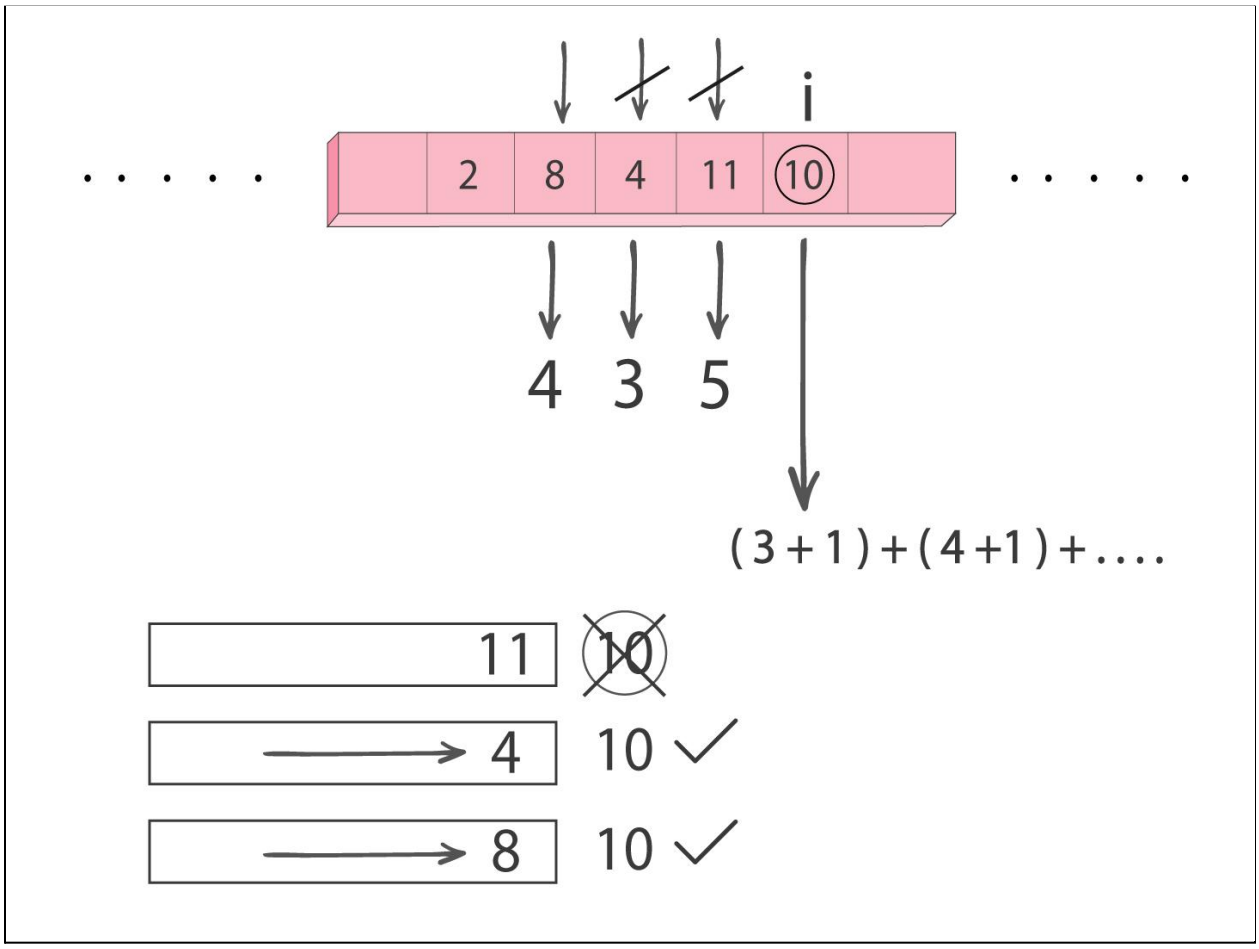
Note that the elements in the bitonic subsequence need not be consecutive in the given array but the order should remain the same.

Explanation:

The brute force approach to this problem is to find all the subsequences and then give the maximum length. The time complexity of this will be exponential as there can be 2^n subsequences.

Dynamic Programming Approach is to divide this problem into a smaller problem by finding the longest subsequence finishing at each index.

Let's try to understand this by an example:



Time Complexity of this solution is **O(n²)**.

Code:

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

int lis(int* input, int n) {
    int* output = new int[n];
    output[0] = 1;
    for (int i = 1; i < n; i++) {
        output[i] = 1;
        for (int j = i - 1; j >= 0; j--) {
            if (input[j] > input[i]) {
                continue;
```

```
            }
            int possibleAns = output[j] + 1;
            if (possibleAns > output[i]) {
                output[i] = possibleAns;
            }
        }
    }
    int best = 0;
    for (int i = 0; i < n; i++) {
        if (best < output[i]) {
            best = output[i];
        }
    }
    delete [] output;
    return best;
}

int main() {
    int n;
    cin >> n;
    int * input = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> input[i];
    }
    int ans = lis(input, n);
    cout << ans << endl;
    delete [] input;
}
```

Staircase Problem

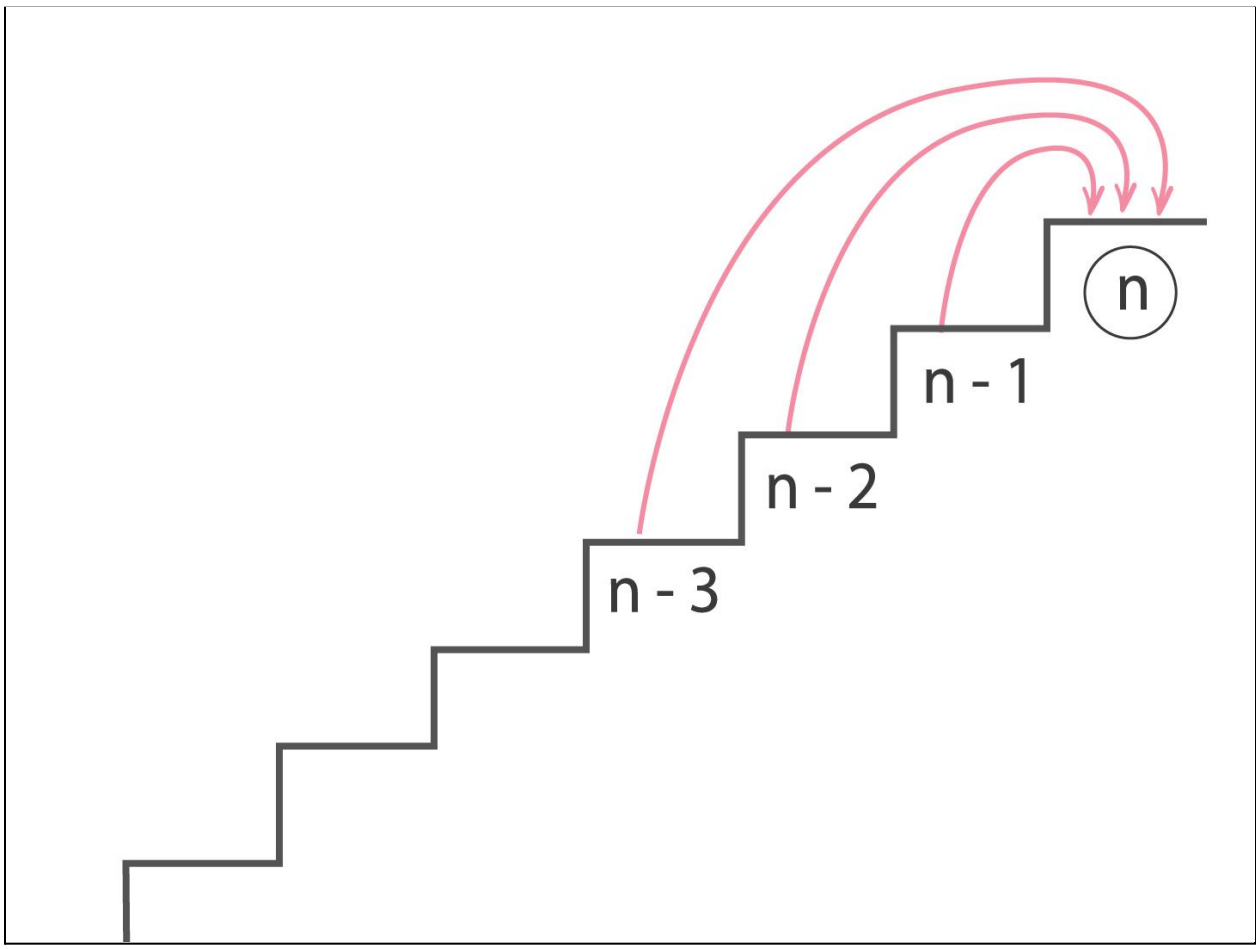
Problem Statement: A child is running up a staircase with n steps and can hop either 1 step, 2 steps or 3 steps at a time. Implement a method to count how many possible ways the child can run up to the stairs. You need to return the number of possible ways.

Time complexity of your code should be O(n).

Since the answer can be pretty large print the answer % mod(10^9 +7)

Explanation:

There are **n** steps and reaching the last step could have been achieved from **(n-1)**th, **(n-2)**th or **(n-3)**th step because the child can only take 3 or less steps at a time.



So, basically it is like a fibonacci series as **S(n) = S(n-1) + S(n-2) + S(n-3)**, and implementing the solution using recursion will not be an ideal approach. So we will use the Dynamic Programming approach just like we did in the Fibonacci series.

Base Cases:

1. There is only one way to take no step.
2. There is only one way to take one step.
3. There are two ways to take 2 steps, i.e., {1, 1} and {2}.

Coin Change Problem

Problem Statement: You are given an infinite supply of coins of each denomination D = {D0, D1, D2, D3, ... Dn-1}. You need to figure out the total number of ways W, in which you can make a change for Value V using coins of denominations D.

Note : Return 0, if change isn't possible.

W can be pretty large so output the answer % mod(10^9 + 7)

Explanation:

This seems to be a very simple problem in which, like the previous one, **F(n) = F(n-D_0) + F(n-D1) + F(n-D2) and so on..** But there may be similar cases in this problem, for example, if we take **n = 4**, and we have **1, 2, 5 Rs coins** then **{1, 1, 2} and {1, 2, 1}** are exactly the same because in coins the order doesn't matter.

n, {D_0, D_1, D_2, D_3, ... D_4}

The problem can be reduced in two ways:

1. D_0 is included: **n-D_0, {D_0, D_1, D_2, D_3, ... D_4}**
2. D_0 is not included: **n, {D_1, D_2, D_3, ... D_4}**

Base Cases:

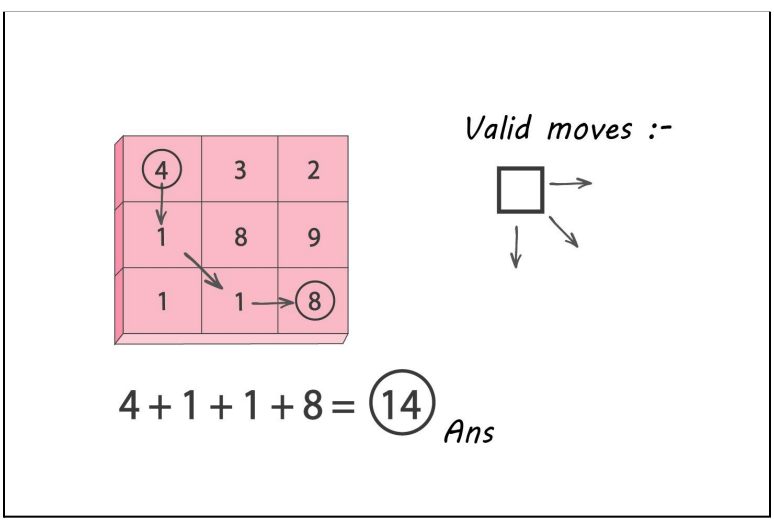
1. If n is equal to 0, return 1
2. If length of array is equal to 0, return 0

Minimum Cost

Problem Statement: We are given a 2D array with [i][j]th cell holding the cost to travel through that cell. Our goal is to find the minimum cost to reach the end point from the start.

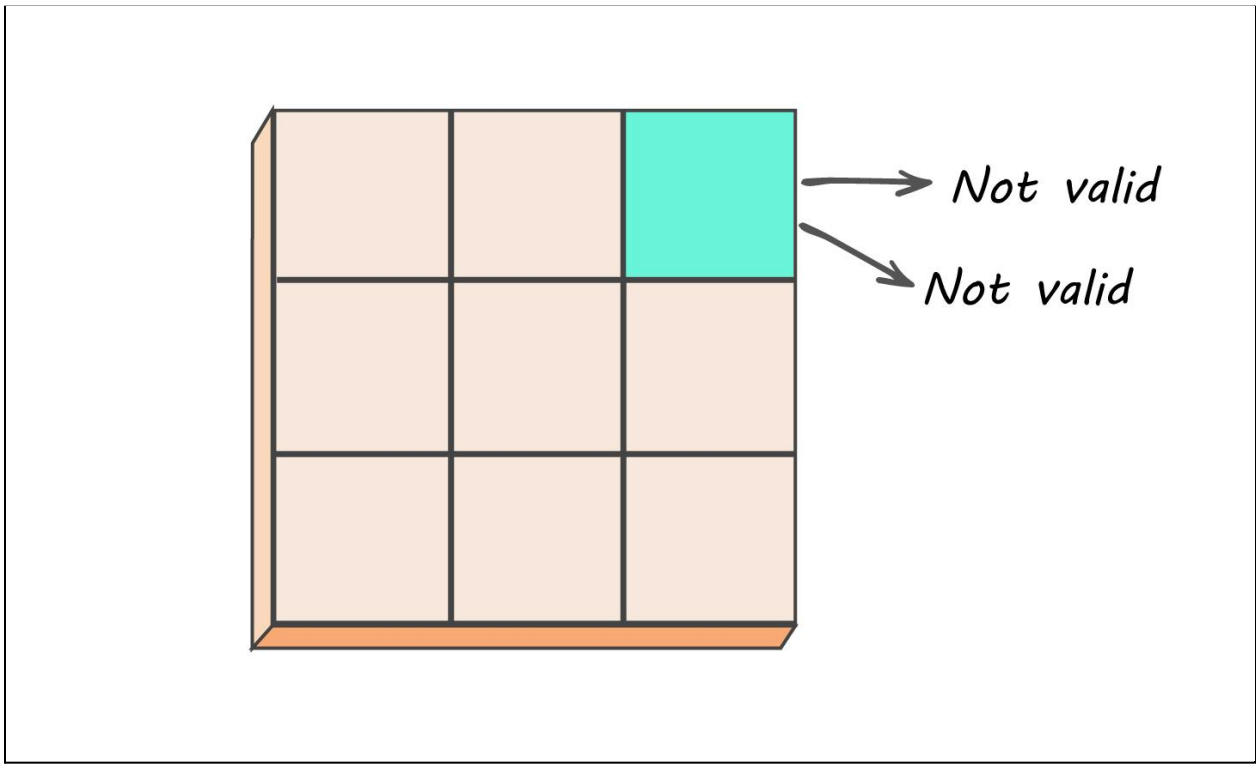
Explanation:

Naive Approach: The valid directions in which we can move are right, down and down-right.



Now, when we are at the edge of the matrix, one or two of our paths will also become invalid. So to choose, which path to take at that time, we have two options:

1. Either to check every time whether we are at an edge.
2. For an invalid move at the edge, we take the cost to travel to be infinite, so it automatically doesn't get chosen.



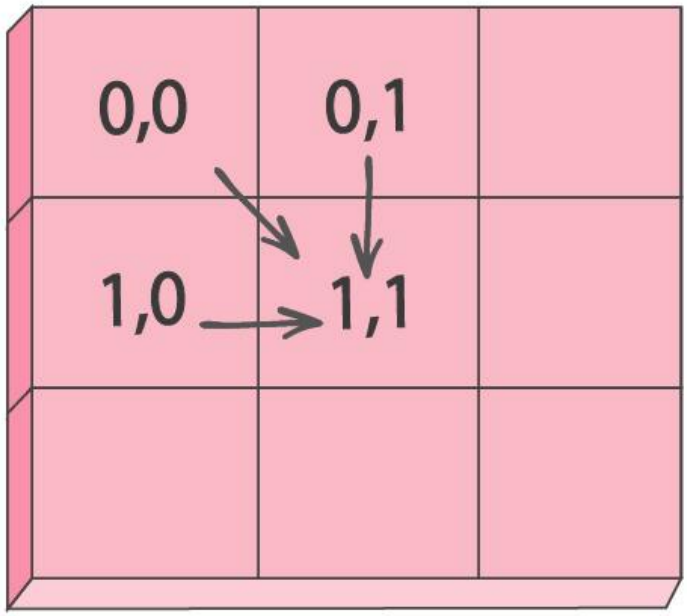
Code:

```
int min_cost(int** input, int si, int sj, int ei, int ej) {
    if (si == ei && sj == ej) {
        return input[ei][ej];
    }
    if (si > ei || sj > ej) {
        return INT_MAX;
    }
```

```
    int option1 = min_cost(input, si + 1, sj, ei, ej);
    int option2 = min_cost(input, si + 1, sj + 1, ei, ej);
    int option3 = min_cost(input, si, sj + 1, ei, ej);
    return input[si][sj] + min(option1, min(option2, option3));
}

int main() {
    int ** input = new int*[3];
    input[0] = new int[3];
    input[1] = new int[3];
    input[2] = new int[3];
    input[0][0] = 4;
    input[0][1] = 3;
    input[0][2] = 2;
    input[1][0] = 1;
    input[1][1] = 8;
    input[1][2] = 3;
    input[2][0] = 1;
    input[2][1] = 1;
    input[2][2] = 8;

    cout << min_cost(input, 0,0,2,2) << endl;
    cout << min_cost2(input,3,3) << endl;
    delete [] input[0];
    delete [] input[1];
    delete [] input[2];
    delete [] input;
}
```

Time Complexity: From every cell we are making three recursive calls, so if the height of the tree is m, then this solution has time complexity **O(3^m)** which is exponential.

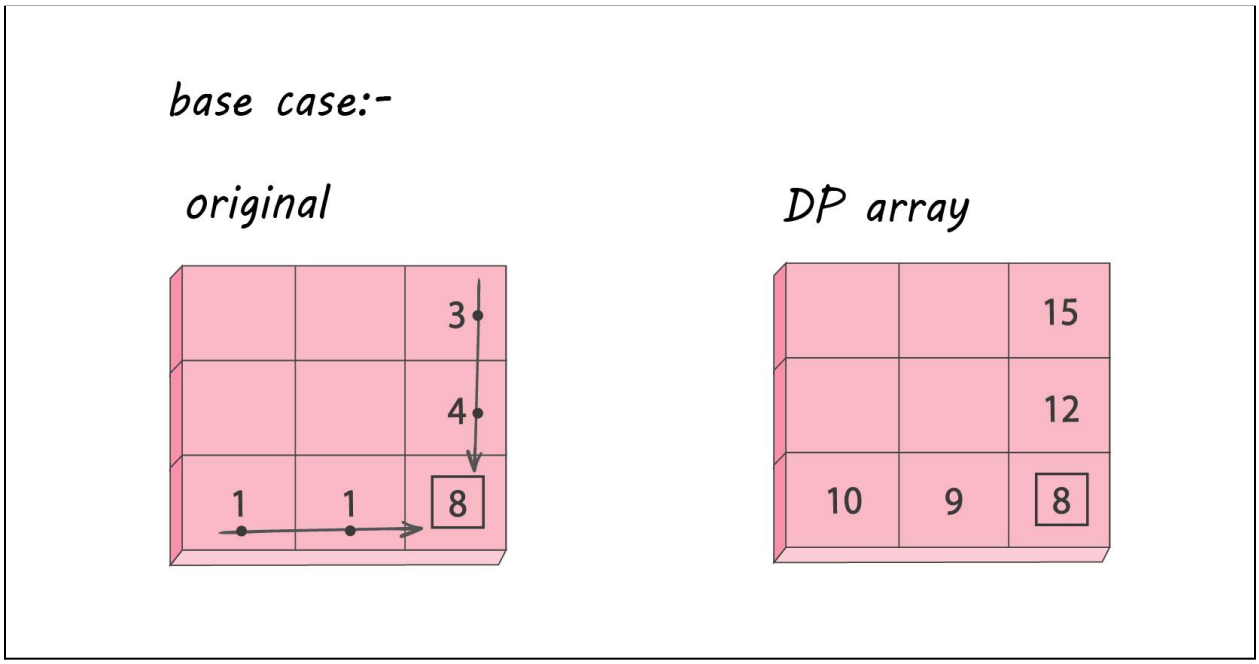
But this can be reduced using memoization, since lots of repetitions are happening in this solution.



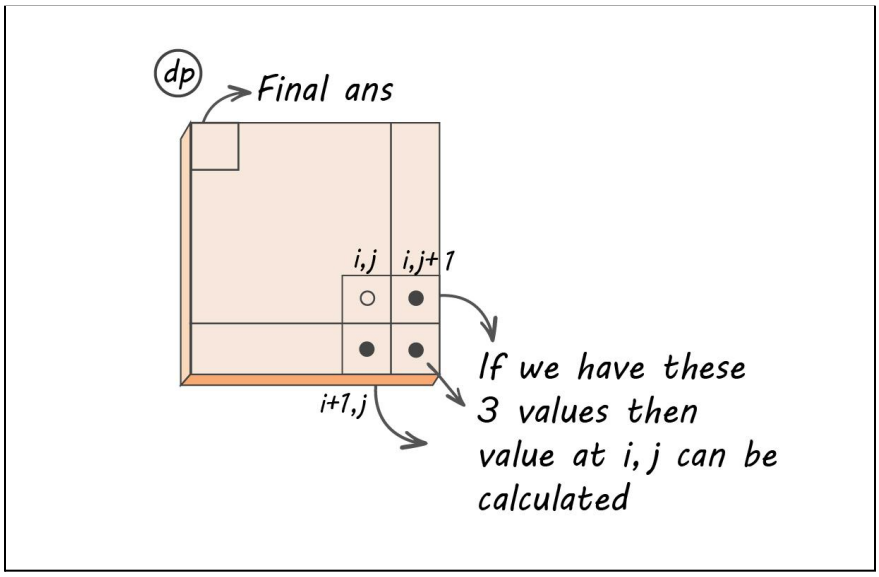
To implement this, we simply pass another 2D array to check whether some [i][j] position has already been checked or not. Try to implement this yourself.

Iterative Solution:

We make a dp array that at each index, stores the cost to go to the last index from the current index.



Now, to fill the value at [i][j], we need three values at **[i][j+1], [i+1][j], [i+1][j+1].**



Code:

```
int min_cost2(int** input, int m, int n) {
    int ** dp = new int*[m];
    for (int i = 0; i < m; i++) {
        dp[i] = new int[n];
    }
    dp[m - 1][n-1] = input[m-1][n-1];
    for (int i = m - 2; i >= 0; i--) {
        dp[i][n -1] = dp[i + 1][n-1] + input[i][n-1];
    }

    for (int j = n - 2; j >=0; j--) {
        dp[m -1][j] = dp[m - 1][j + 1] + input[m-1][j];
    }

    for (int i = m - 2; i >=0; i--) {
        for (int j = n - 2; j >=0 ; j--) {
            dp[i][j] = input[i][j] + min(dp[i+1][j], min(dp[i+1][j+1],
dp[i][j + 1]));
        }
    }
    return dp[0][0];
}
```

Dynamic Programming - 2

Longest Common Subsequence

Problem Statement: Given two strings S1 and S2 with lengths M and N respectively, find the length of the longest common subsequence.

A subsequence of a string S whose length is K, is a string containing characters in the same relative order as they are present in S, but not necessarily contiguous. Subsequences contain all the strings of length varying from 0 to K. For example, subsequences of string "abc" are -- ""(empty string), a, b, c, ab, bc, ac, abc.

Explanation:

Example1: We are given with two strings **s1 = "adebc" and s2 = "dcadb".**

"a", "d", "b", "c", "ad", "ab", "db", "dc" and "adb" are present as a subsequence in both the strings in which "adb" has the maximum length. There are no other common subsequences of length greater than 3 and hence the answer.

Example2: We are given with two strings **s1 = "abcdef" and s2 = "a**1**b**2d3**c**4**def"** then the longest common subsequence among these two is **abcdef**.

Naive Approach: If we look carefully at the first character of the strings then we can have two cases:

- **The first characters of the two strings are equal.**
    - This means we can call our LCS function on the remaining string and whatever answer we get, we can add 1 to it for the final answer, i.e., **1 + LCS (s1 + 1, s2 + 1)**.

- **The first characters of the two strings are not equal.**
    - Consider an example in which s1 starts with **a** and s2 starts with **b**. This creates three more possibilities:
        - LCS starts with a.
        - LCS starts with b.
        - LCS starts with neither a nor b.

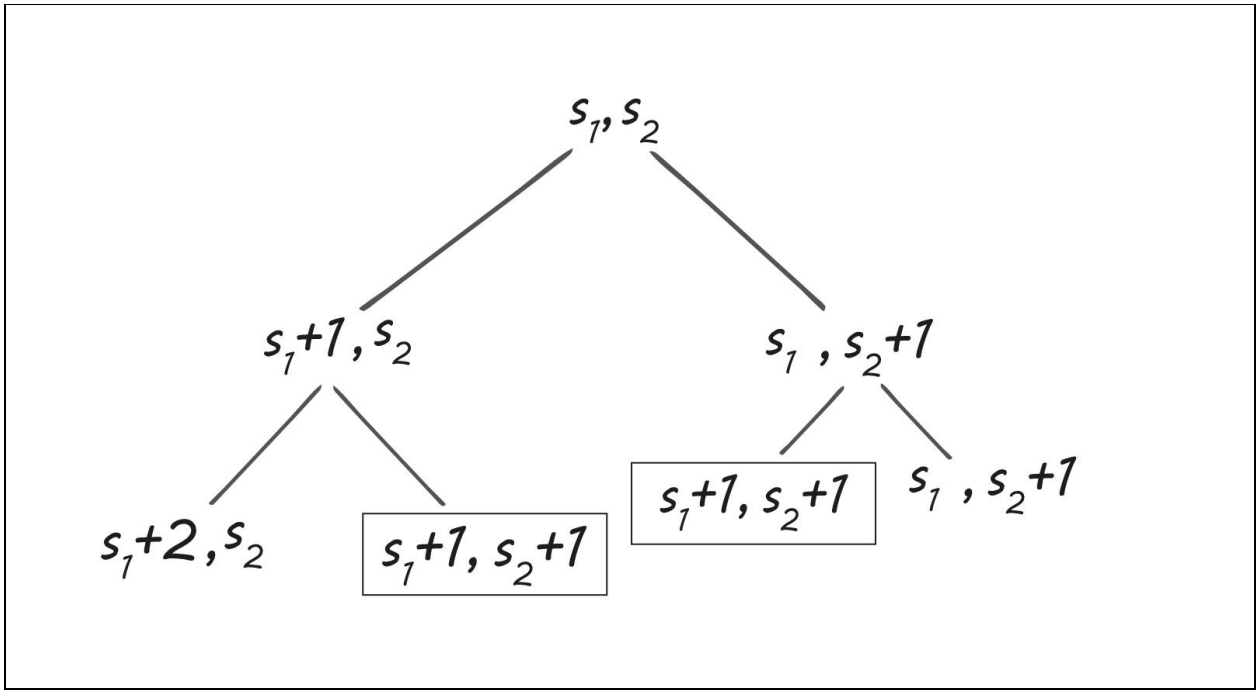
To take into consideration, the above three cases we need to find the answer from each of them and take the maximum for the final answer, i.e.,

max (LCS (s1 + 1, s2), LCS (s1, s2 + 1)).

Let's look at the code:

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

int lcs(char *s1, char *s2){
    if(s1[0] == '\0' || s2[0] == '\0'){
        return 0;
    }
    if(s1[0] == s2[0]){
        return 1 + lcs(s1+1, s2+1);
    } else {
        int option1 = lcs(s1, s2+1);
        int option2 = lcs(s1+1, s2);
        return max(option1, option2);
    }
}
int main(){
    char a[100];
    char b[100];
    cin >> a;
    cin >> b;
    cout << lcs(a,b) <<endl;
    return 0;
}
```

The above solution is very slow because there are multiple repetitive recursion calls happening.



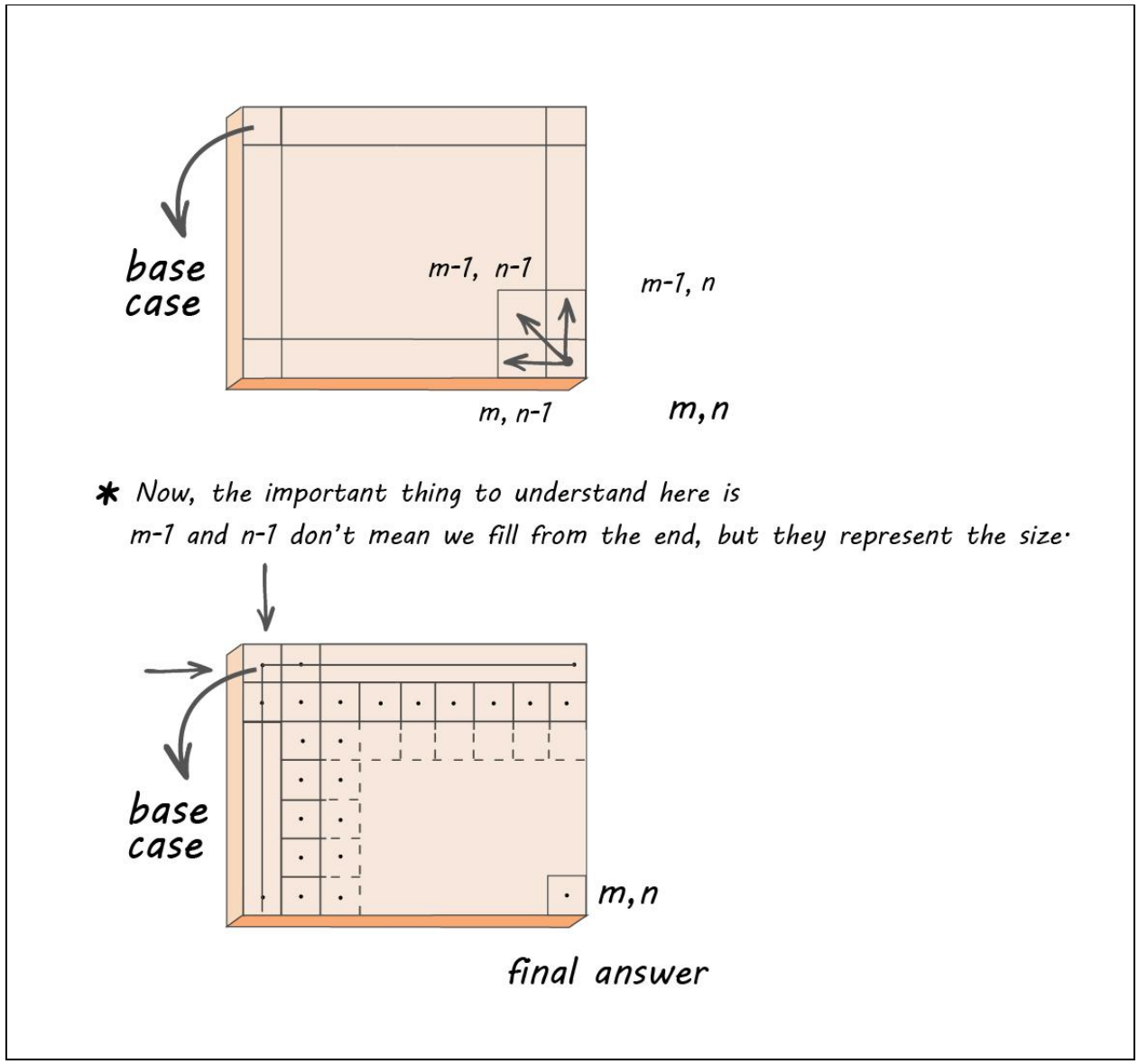
To improve the above solution, we can use memoization, i.e., we can store the data so that we can use that data instead of making a recursive call that has already been executed. For that, we create a helper function that takes a 2D array which stores -1 initially to tell that work has not yet been done on that position. After the recursive call at that position has been made, we store the anwer in the dp array.

Code:

```
int lcs2Helper(char *s1, char *s2, int m, int n, int **dp){
    if(m == 0 || n==0){
        return 0;
    }
    // if the work has already been done then return the answer stored
    if(dp[m][n] > -1){
        return dp[m][n];
```

```
    }
    int ans;
    if(s1[0] == s2[0]){
        ans =  1 + lcs2Helper(s1+1, s2+1, m-1, n-1, dp);
    } else {
        int option1 = lcs2Helper(s1, s2+1, m, n-1, dp);
        int option2 = lcs2Helper(s1+1, s2, m-1, n, dp);
        ans = max(option1, option2);
    }
    dp[m][n] = ans;
    return ans;
}
int lcs2(char *s1, char *s2){
    int m = strlen(s1);
    int n = strlen(s2);
    int **dp = new int *[m+1];
    for(int i=0; i<=m; i++){
        dp[i] = new int[n+1];
        for(int j=0; j<=n; j++){
            dp[i][j] = -1; // initializing with -1
        }
    }
    int ans = lcs2Helper(s1, s2, m, n, dp);
    // deallocating the memory
    for(int i=0; i<=m; i++){
        delete []dp[i];
    }
    delete []dp;
    return ans;
}
```

Iterative Approach:



Code:

```
int lcsI(char* s1, char* s2) {
    int m = strlen(s1);
    int n = strlen(s2);
    int** dp = new int*[m + 1];
```

```
    for (int i = 0; i <= m; i++) {
        dp[i] = new int[n + 1];
    }
    for (int i = 0; i <= m; i++) {
        dp[i][0] = 0;
    }
    for (int i = 0; i <=n ;i++) {
        dp[0][i] = 0;
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j<= n; j++) {
            if (s1[m - i] == s2[n - j]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
            }   else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    int ans =  dp[m][n];
    // delete 2d array
    for (int i =0; i <=  m; i++) {
        delete [] dp[i];
    }
    delete [] dp;
    return ans;
}
```

Knapsack Problem

Problem Statement: Given the weights and values of 'N' items, we are asked to put these items in a knapsack, which has a capacity 'C'. The goal is to get the maximum value from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

Explanation:

For example:

Items: {Apple, Orange, Banana, Melon}
Weights: {2, 3, 1, 4}
Values: {4, 5, 3, 7}
Knapsack capacity: 5

Possible combinations that satisfy the given conditions are:

Apple + Orange (total weight 5) => 9 value
Apple + Banana (total weight 3) => 7 value
Orange + Banana (total weight 4) => 8 value
Banana + Melon (total weight 5) => 10 value

This shows that **Banana + Melon** is the best combination, as it gives us the maximum value, and the total weight does not exceed the capacity.

Naive Approach: First-of-all, let's discuss the brute-force-approach, i.e., the **recursive approach**. There are two possible cases for every item, either to put that item into the knapsack or not. If we consider that item, then its value will be contributed towards the total value, otherwise not. To figure out the maximum value obtained by maintaining the capacity of the knapsack, we will call recursion over these two cases simultaneously, and then will consider the maximum value obtained out of the two.

If we consider a particular weight 'w' from the array of weights with value 'v' and the total capacity was 'C' with the initial value 'Val', then the remaining capacity of the knapsack becomes 'C-w', and the value becomes 'Val + v'.

Let's look at the recursive code for the same:

```
int knapsack(int *weight, int *values, int n, int maxWeight) {
    // Base case : if the size of array is 0 or we are not able to add any
more weight to the knapsack
    if(n == 0 || maxWeight == 0) {
        return 0;
    }
//If the particular weight's value extends limit of knapsack remaining
// capacity, then we have to simply skip it
    if(weight[0] > maxWeight) {
        return knapsack(weight + 1, values + 1, n - 1, maxWeight);
    }

    // Recursive calls
    //1. Considering the weight
    int x = knapsack(weight + 1, values + 1, n - 1, maxWeight - weight[0])
+ values[0];
    // 2. Skipping the weight and moving forward
    int y = knapsack(weight + 1, values + 1, n - 1, maxWeight);

    // finally returning the maximum answer among the two
    return max(x, y);
}
```

Now, the memoization and DP approach is left for you to solve. For the code, refer to the solution tab of the same. Also, figure out the time complexity for the same by running the code over some examples and by dry running it.

Subset Sum Problem

Problem Statement: Given an array of n integers, find if a subset of sum k can be formed from the given set. Print Yes or No.

Explanation:

Example: given array = {1, 2, 3} and k = 4, so subset {1, 3} has a sum of elements equal to k. Therefore, our output will be true.

Brute Force Approach: The naive approach that can be thought of for this problem is to generate all the subsets and check whether they satisfy the constraints given or not. This is very similar to what we did in the knapsack problem.

Iterative Approach: Taking an example, **val = [1, 3, 5, 7, 9] and sum = 8.**

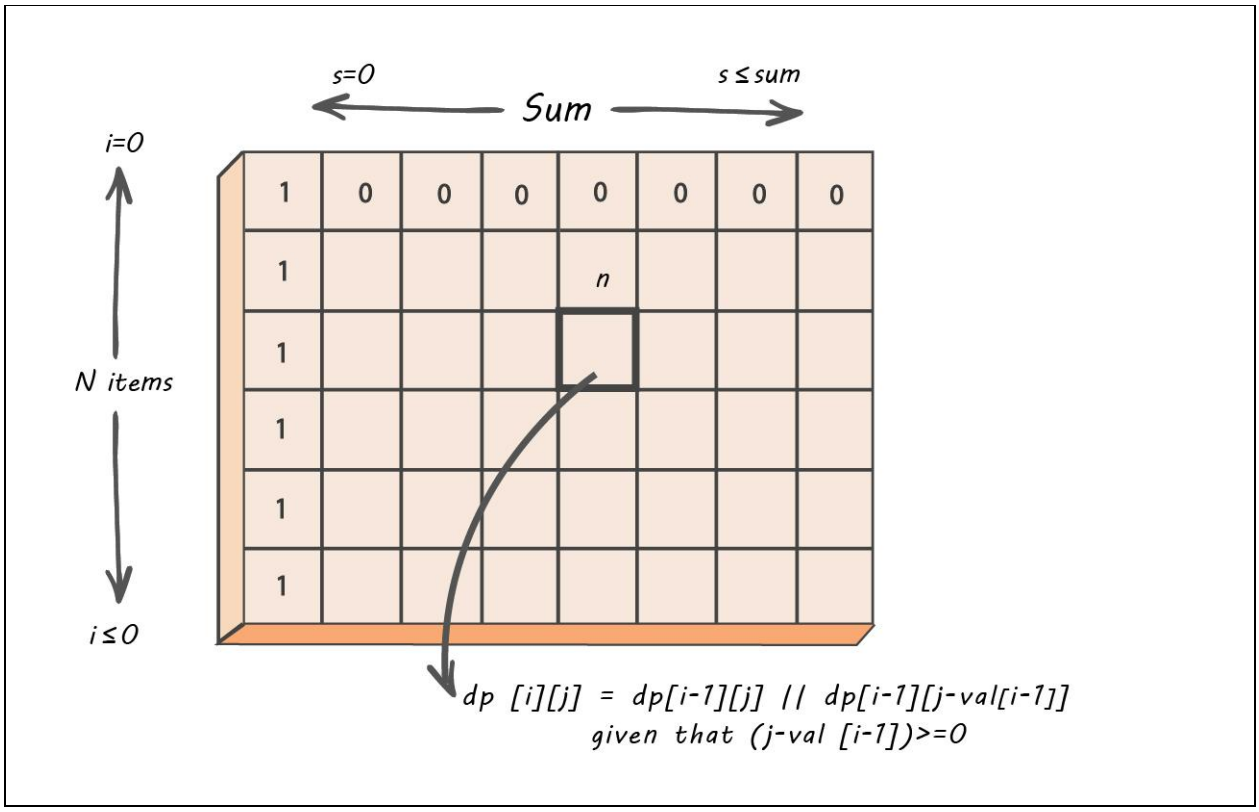
If we try to reduce this problem to a simpler one then there can be two cases:

- 9 in included in the subset sum :
    - For this case, the problem gets reduced to whether in **[1, 3, 5, 7]** there exists a subset whose sum is **8 - 9 = -1**.
- 9 is not included in the subset sum:
    - For this case the problem gets reduced to whether in **[1, 3, 5, 7]** there exists a subset whose sum is **8**.

Therefore, we are going to create a dp array that stores the answer of **dp[i][j] = dp[i-1][j] || dp[i-1][j-val[i-1]]**, given that **(j-val[i-1]) >= 0**.

The above expression means just what the two cases represent.

The first row of the dp array is initialized with 0 and the first column excluding the first element is initialized with 1, because the row is an indication of the items in the subset and the columns represent the sum values.



Code:

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

bool subsetSum(int *val, int n, int sum){

    bool **dp = new bool*[n+1];
    for(int i=0; i<=n; i++){
        dp[i] = new bool[sum+1];
    }
    // first row is initialized with false
    for(int i=0; i<=sum; i++){
        dp[0][i] = false;
```

```
    }
    // first column is initialized with true
    for(int i=0; i<=n; i++){
        dp[i][0] = true;
    }
    for(int i=1; i<=n; i++){
        for(int j=1; j<=sum; j++){

            dp[i][j] = dp[i-1][j];
            if(j>=val[i-1]){
                dp[i][j] = dp[i][j] || dp[i-1][j-val[i-1]];
            }
        }
    }
    bool ans = dp[n][sum];
    // delete dp array
    return ans;
}

int main(){
    int val[] = {1, 3, 5, 7, 9};
    int sum = 12;

    cout<<subsetSum(val, 5, sum);
    return 0;
}
```

Optimized Approach: In the above solution, for i^{th} row, we are using the previous row to compute or answer and the rows before that have no significance whatsoever. So,we could have made the dp array with just two rows and could have toggled around these two rows to calculate the solution.

Code:

```
bool subsetSum(int* val,int n,int sum){
```

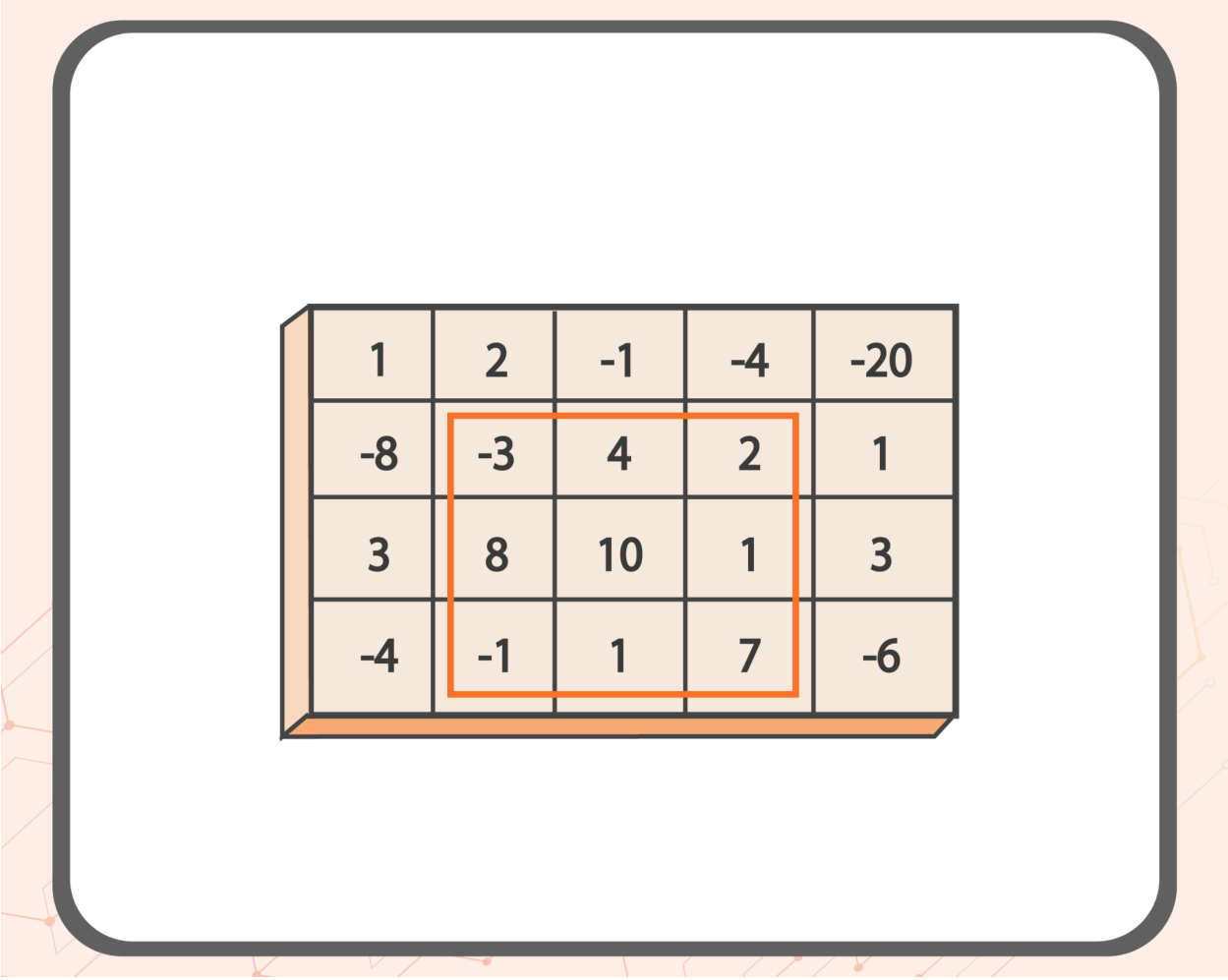
```
    bool** dp = new bool*[2];
    for(int i=0;i<=1;i++){
        dp[i] = new bool[sum+1];
    }

    for(int i=0;i<=sum;i++){
        dp[0][i] = false;
    }
    dp[0][0] = true;

    int flag = 1;

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

            dp[flag][j] = dp[flag^1][j];

            if(j>=val[i-1]){
                dp[flag][j] = dp[flag][j] || dp[flag^1][j-val[i-1]];
            }
        }
        flag = flag ^ 1;
    }
    bool ans = dp[flag^1][sum];
    // delete dp array
    return ans;

}
```

Maximum Sum Rectangle

Problem Statement: Given a 2D array, find the maximum sum rectangle in it. In other words, find the maximum sum over all rectangles in the matrix.

Explanation:

Example: in the following 2D array, the maximum sum subarray is highlighted with a blue rectangle and the sum of this subarray is 29.

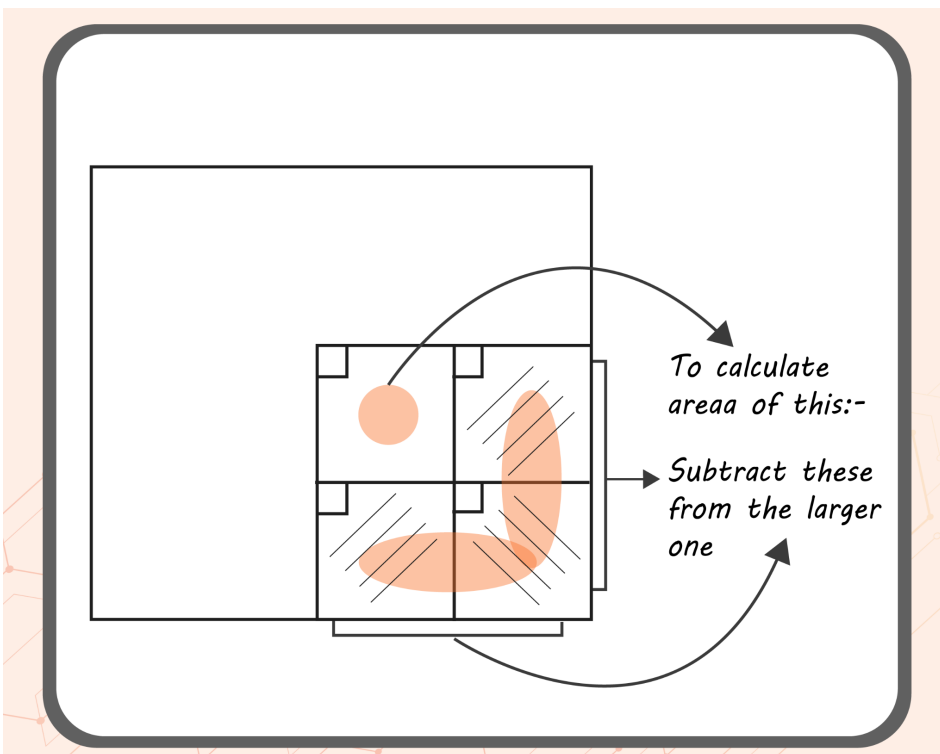


Naive Approach:

We can take a particular cell [i][j] as the upper left corner of the rectangle, the bottom right corner will be the last cell given to us and we can store the area of the rectangle thus formed in the cell [i][j]. We can do this for every cell present in the outer rectangle.

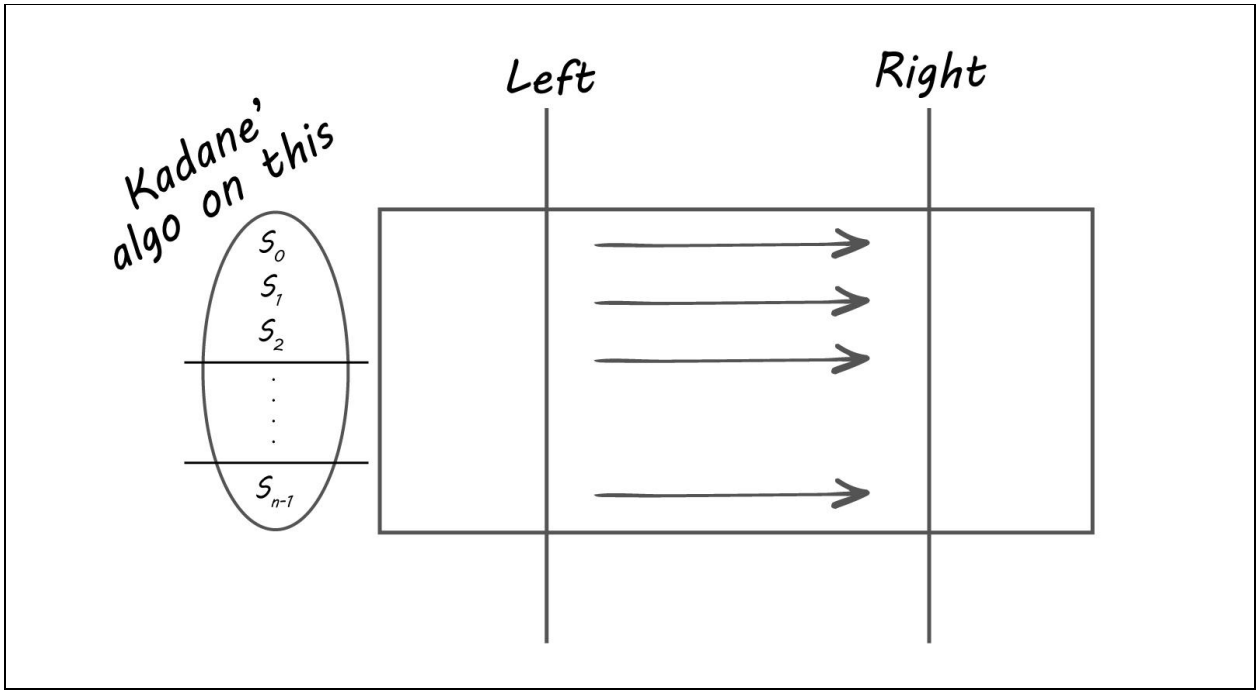
To calculate the area for a rectangle, that has no corner on any of the edges of the outer rectangle refer to the figure given below:

This approach will take **O(N^4)** time and hence is a very bad solution.



Optimized Approach:

We can take two points left and right with **O(N^2)** possibilities, and in between those two points, we find the sum of rows. Now, suppose we have the sum of rows S0, S1, S2 and so on, then we can apply Kadane's algorithm **O(N)** on these to find the top and bottom points that will help us find the coordinates of the rectangle that contribute to the maximum sum.



Now, suppose our left point increments by 1 then we need not find the entire sum array for the rows again, instead what we can do is add the numbers of the respected added column to the sum array elements. Time taken for this sub problem will also be **O(N).**

Therefore, the overall time complexity of the problem will be O(N^3) since for every pair of left and right, we are doing O(2N) work.

Bit Manipulation

Introduction

Until now, we were using only integers, floating-point numbers, etc for our codes, but now we are going to look at the bits present in these data types for our calculations and problem-solving.

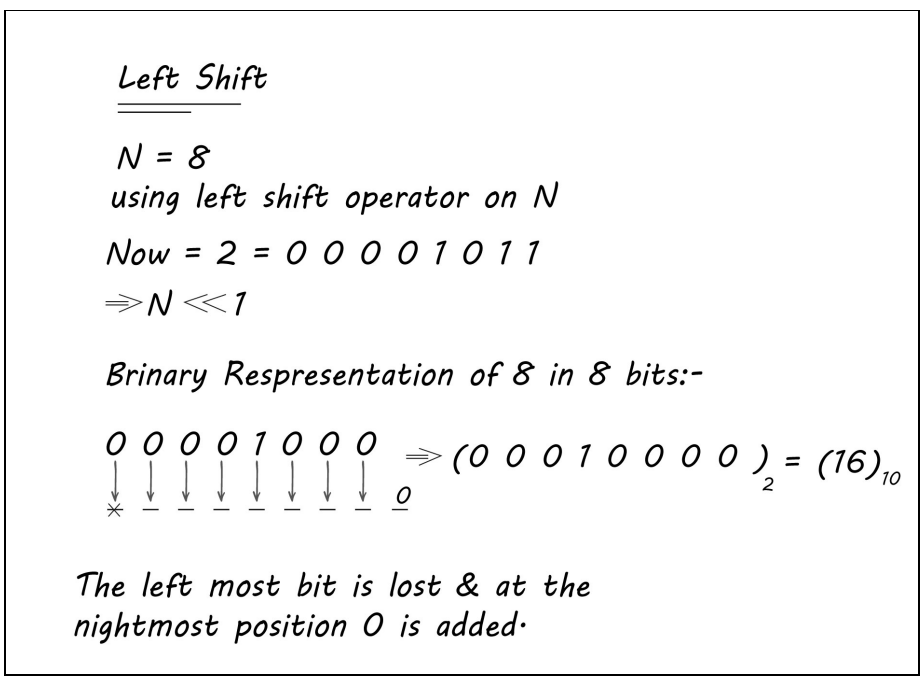
This is going to highly optimize our code, for example, an integer has 32 bits, now working on any of those 32 bits takes only O(1) time.

Shift Operators

Shift operators are of two types:

1. **Left Shift:** This operator is represented by **<<**. Example : **8 << 1 = 16**

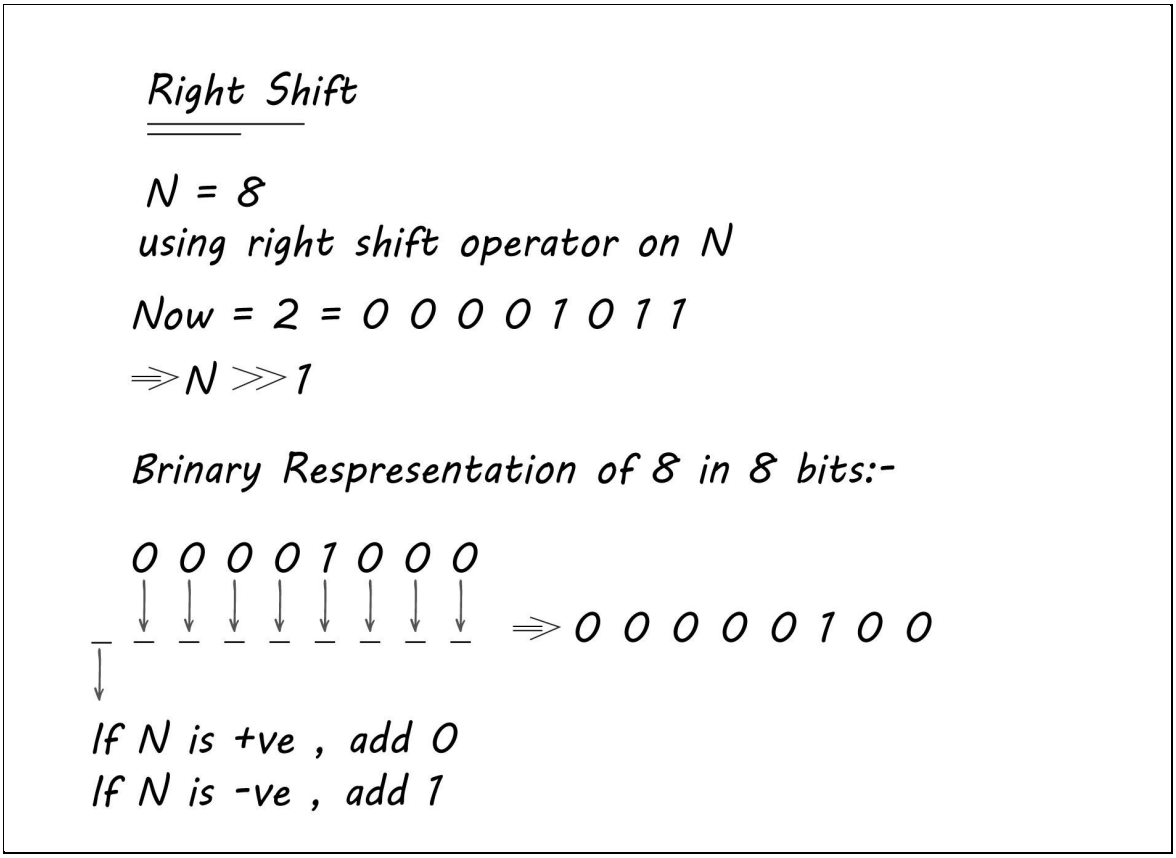
   This operator shifts every bit to its left position. For a clear understanding look at the image given below:



General Formula: N << i -> N*2^i

2. **Right Shift:** This operator is represented by **>>**. Example : **8 >> 1 = 4**

   This operator shifts every bit to its right position. For a clear understanding look at the image given below:



General Formula: N >> i -> floor(N/2^i)

Some other bitwise operators

1. **& (bitwise AND)** takes 2 numbers as operands and performs logical AND on every bit of the two numbers. The result of AND is 1 only if both bits are 1.
2. **| (bitwise OR)** takes 2 numbers as operands and performs logical OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. **~ (bitwise NOT)** takes 1 number and flips all the bits of it.
4. **^ (bitwise XOR)** takes 2 numbers as operands and performs XOR on every bit of 2 numbers. The result of XOR is 1 if the two bits are different.

A	B	A | B	A & B	A^B	~A
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

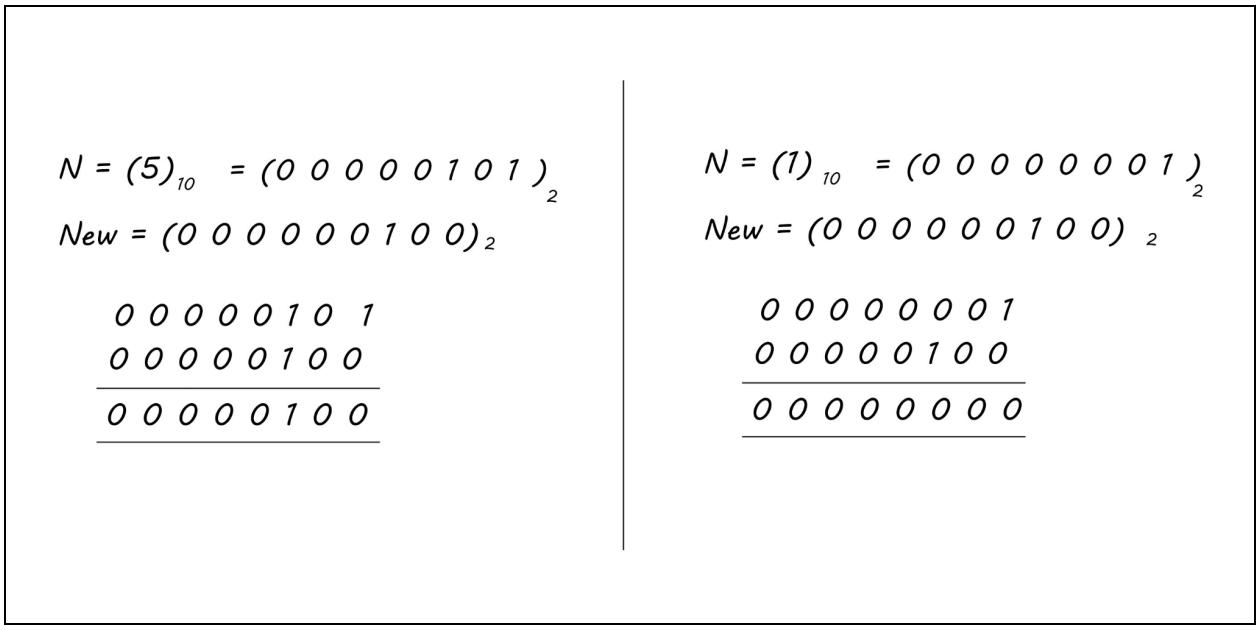
Check i^{th} bit

i^{th} bit is set if its value is 1 and it is not set if its value is 0. We will be given a number **N** and a position **i** and we will have to check if the bit at **i^{th}** position is set or not. Now, let's take a look at an example to understand how we are going to proceed:

Given : **N = 5, i = 2**

The binary representation of 5 in 8 bits is **0 0 0 0 0 1 0 1**. We make a new number in 8 bit format that has only one bit set and that bit is at the **i^{th}** position. Then we take the **&** of these two numbers. If the answer is zero, then the **i^{th}** bit is set, else it is not set.

In the diagram, we have also shown an example where ith is not set. The example has **N = 1, i = 2**.



Flip i^{th} bit

We know that :

1 ^ 0 = 1

0 ^ 0 = 0

Therefore, **X ^ 0 = X** and

0 ^ 1 = 1

1 ^ 1 = 0

Therefore, **X ^ 1 = ~X**

This means that to flip a bit we can take its XOR with $\mathbf{2^i}$.

Example:

$$N = 11, \; i = 2$$

$$N = 0\,0\,0\,0\,1\,0\,1\,1$$

$$N = 0\,0\,0\,0\,1\,0\,1\,1$$

$$Now = 2^1 = 0\,0\,0\,0\,1\,0\,1\,1$$

$$\underline{\underline{X\ OR}}$$

$$\begin{array}{r} 0\,0\,0\,0\,1\,0\,1\,1 \\ 0\,0\,0\,0\,0\,1\,0\,1 \\ \hline 0\,0\,0\,0\,1\,1\,1\,1 \end{array}$$

2^i can be calculated using the left shift operator, i.e., 2^i = 1 << i.

Check N: odd or even

The basic way to do this is taking modulo by 2:

```
If ( N % 2 == 0 ) {
        Even
} else {
        Odd
}
```

Optimized Approach: Sometimes, in competitive programming our solution might not work by 0.1 seconds or a very small time, so to optimize this we can use bitwise operators.

Let's take a look at the binary representation of a few numbers:

11 : **0 0 0 0 1 0 1 <u>1</u>**

8 : **0 0 0 0 1 0 0 <u>0</u>**

9 : **0 0 0 0 1 0 0 <u>1</u>**

10 : **0 0 0 0 1 0 1 <u>0</u>**

The last bit of every odd number is 1 and for every even number it is 0 because the last bit is represented by $\mathbf{2^0}$. So we can just check the last bit of a number to check whether a number is odd or even. Try to implement this by using the concept of checking i^{th} bit taught before.

Check N whether it is power of 2 or not

The basic way to do this is :

```
while ( ( N % 2 == 0) ) {
        N = N / 2;
}
If ( N == 1 ) {
        POWER OF 2
} else {
        NOT POWER OF 2
}
```

The above approach runs in **O($\log_2$(N))** time.

Optimized Approach :

Let's take a look at the binary representation of some numbers and see if we can infer anything from them :

2 : **0 0 0 0 0 0 0 1 0**

4 : **0 0 0 0 0 0 1 0 0**

8 : **0 0 0 0 0 1 0 0 0**

11 : **0 0 0 0 0 1 0 1 1**

16 : **0 0 0 0 1 0 0 0 0**

15 : **0 0 0 0 0 1 1 1 1**

There is one common pattern, that the numbers which are powers of 2 have only one bit set. Now take a look at the numbers 16 and 15 :

16 : **0 0 0 0 1 0 0 0 0**

15 : **0 0 0 0 0 1 1 1 1**

If we take **16 & 15**, then we get 0. This happens for all numbers **N and N-1** if N is a power of 2.

Generalized Formula: If N & (N-1) == 0, then N is a power of 2.

Remove all set bits from LSB to i :

LSB or Least Significant Bit is the rightmost bit.

Let's take a binary number N and **i = 3**,

N = 0 1 1 1 0 1 0 1 1 0

Now, take another number **A = 1 << (i+1)**,

A = 0 0 0 0 0 1 0 0 0 0

B = A-1 = 0 0 0 0 0 0 1 1 1 1

C = ~B = 1 1 1 1 1 1 0 0 0 0

Final answer = N & C = 0 1 1 1 0 1 0 0 0

Generalized Formula: N & M where M = ~((1 << (i + 1)) - 1)

DP and Bitmasking

Introduction

Bitmasking is when bits are used as a mask. To be more elaborate, when we use a collection of bits and in that collection the value of the bits and the different combinations that can be formed, are used to denote something relevant in a problem.

For example, in the problem to find the maximum sum of subset array bitmasking can be applied, where each subset could be represented as a combination of bits.

Suppose there is an array **Arr = [2, 5, 4, 3, 1]**. Now the subsets can be represented as :

0 0 0 0 0

0 0 0 0 1

0 0 0 1 0

0 0 0 1 1

.

.

.

1 1 1 1 1

Here, The length of the mask is equal to the length of the given array, and if the **i^{th}** bit is set, that means the **i^{th}** element of the array is included in the subset.

For an array of size N there can be **2^N** masks.

Solving subset sum using bitmasking:

for mask = 0 to 2^N-1 {

sum = 0

If the bits are set in the mask , then the corresponding element of the array to be added in the sum. // O(N)

If sum becomes equal to the given K, then countSubSet ++

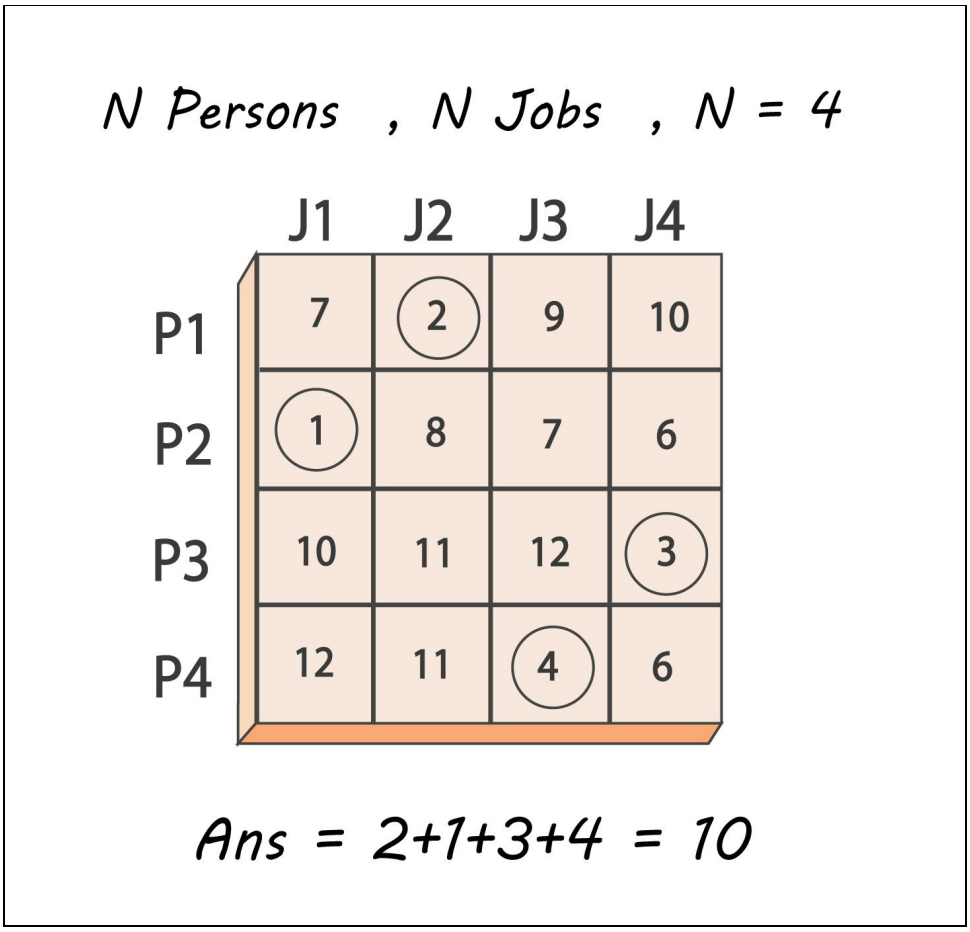
}

The time complexity of the solution using bitmasking will be **O(N*2^N)**.

DP with Bitmasking and Minimum Cost for Jobs

Let us consider a problem in which we are given **N Persons** and **N Jobs**. There is a **N*N matrix** in which the row number gives the person and the column number gives the job. There are values assigned to the cells of the matrix that define how much it will cost to assign the job to a person. There can be only one job for one person and our **goal is to minimize the overall cost**.

Example:



Output: 10

Explanation: In the above example, if **Job 1** is assigned to **person 2**, **job 2** is assigned to **person 1**, **job 3** is assigned to **person 4** and **job 4** is assigned to **person 3**, then we get the minimum cost which is **10**.

Now the combinations of jobs and persons can be represented using bitmasking

For example,

- **1 0 0 0 means that P1 is assigned Job 1**
    - 1 1 0 0 means that P1 is assigned Job 1 and P2 is assigned Job 2
    - 1 0 1 0 means that P1 is assigned Job 1 and P2 is assigned Job 3
    - 1 0 0 1 means that P1 is assigned Job 1 and P2 is assigned Job 4

- **0 1 0 0 means that P1 is assigned Job 2**
    - Here, 1 1 0 0 means that P1 is assigned Job 2 and P2 is assigned Job 1
    - 0 1 1 0 means that P1 is assigned Job 2 and P2 is assigned Job 3
    - 0 1 0 1 means that P1 is assigned Job 2 and P2 is assigned Job 4

- **And so on ...**

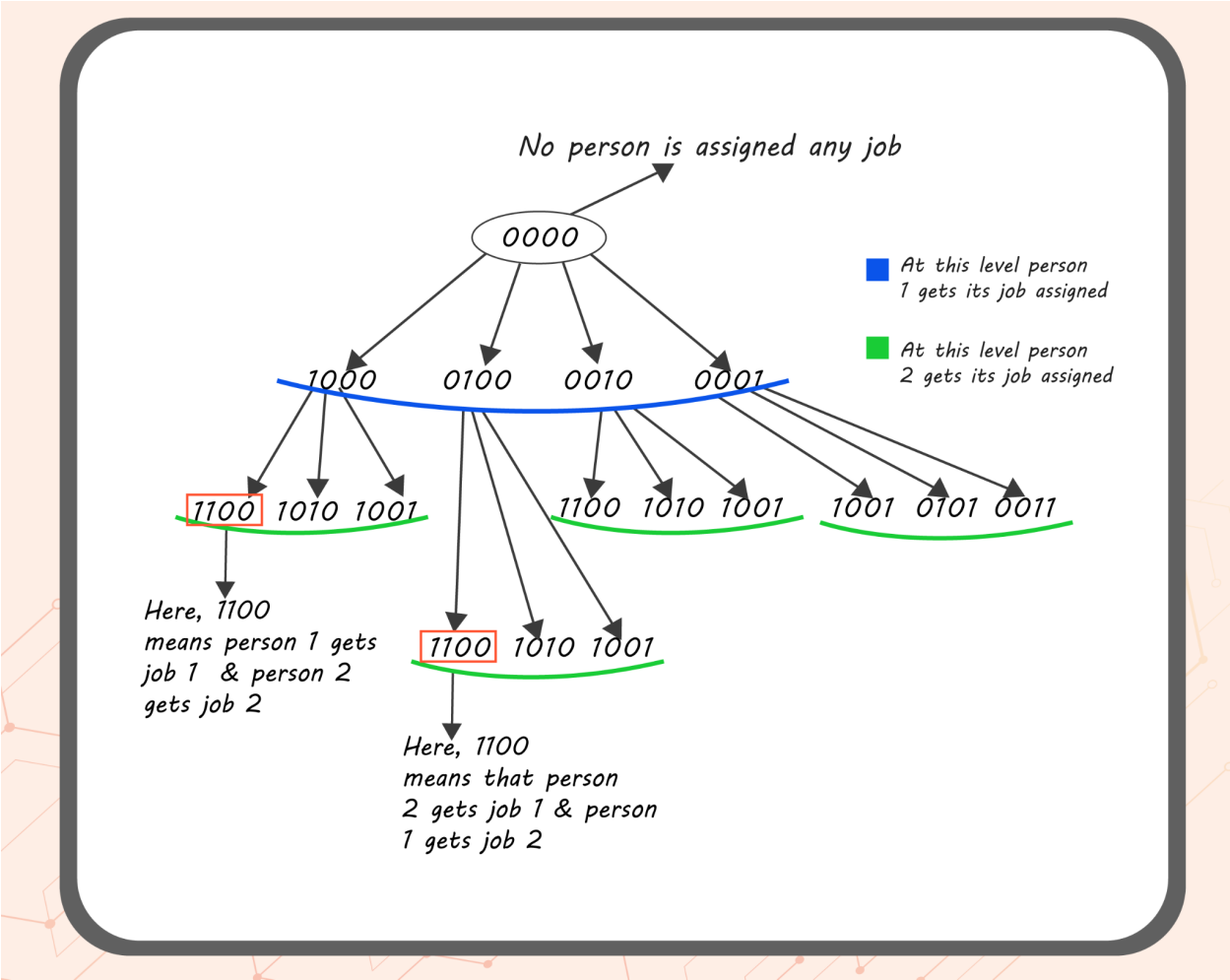
0000 means that no job is assigned to any person.

So we can call recursion with total four possibilities :

1. 1 0 0 0 : assigning P1 with Job 1
2. 0 1 0 0 : assigning P1 with Job 2
3. 0 0 1 0 : assigning P1 with Job 3
4. 0 0 0 1 : assigning P1 with Job 4

And further calling recursions on the above 4 possibilities.

Look at the image below for a better understanding



If we look carefully at the masks we notice that some of them are the same, for example, **1 1 0 0**, However they may represent different answers. This is because of the order of the recursion call.

The first 1 1 0 0 means that Person 1 is assigned Job 1 and Person 2 is assigned Job 2, whereas the second 1 1 0 0 means that Person 1 is assigned Job 2 and Person 2 is assigned Job 1.

This basic approach is not very efficient as there are a lot of repetitions happening, so we use memoization for a more optimized solution.

Code using memoization:

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

// parameters of the function are:
//  1. cost matrix
//  2. number of persons and jobs
//  3. person which has to be assigned a job
//  4. mask (initially zero)
//  5. dp array
int minCost(int cost[4][4], int n, int p, int mask, int*dp){

    if(p>=n){
        return 0;
    }

    if(dp[mask] !=INT_MAX){
        return dp[mask];
    }

    int minimum = INT_MAX;
    for(int j=0; j<n; j++){
        // check if the bit is set in the mask or not
        if(!(mask&(1<<j))){
            // if the bit is not set then call minCost for the next person
            int ans = minCost(cost, n, p+1, mask | (1<<j), dp) +
cost[p][j];
            if(ans < minimum){
                minimum = ans;
            }
        }
    }
    dp[mask] = minimum;
    return minimum;
}


int main(){
    int cost [4][4] = {{10,2,6,5}, {1,15,12,8}, {7,8,9,3}, {15,13,4,10}};

    int* dp = new int[1<<4];
```

```
    for(int i=0; i<(1<<4); i++){
        dp[i] = INT_MAX;
    }

    cout << minCost(cost, 4, 0, 0, dp)<<endl;
    cout<<dp[0];
    return 0;
}
```

Now, try to code the iterative solution on your own.

Dilemma

Problem Statement: Abhishek, a blind man recently bought N binary strings all of equal length. A binary string only contains '0's and '1's . The strings are numbered from 1 to N and all are distinct, but Abhishek can only differentiate between these strings by touching them. In one touch Abhishek can identify one character at a position of a string from the set. Find the minimum number of touches T Abhishek has to make so that he learns that all strings are different .

Explanation:

Let's take an example to understand what exactly touching a string means,

String1 = "1 1 1 0 1 0"

String2 = "1 0 0 1 0 0"

We start iterating the two strings from index 0. Since, String1[0] == Sring2[0], therefore, we move on to the next index and increase the touch count from 0 to 1.

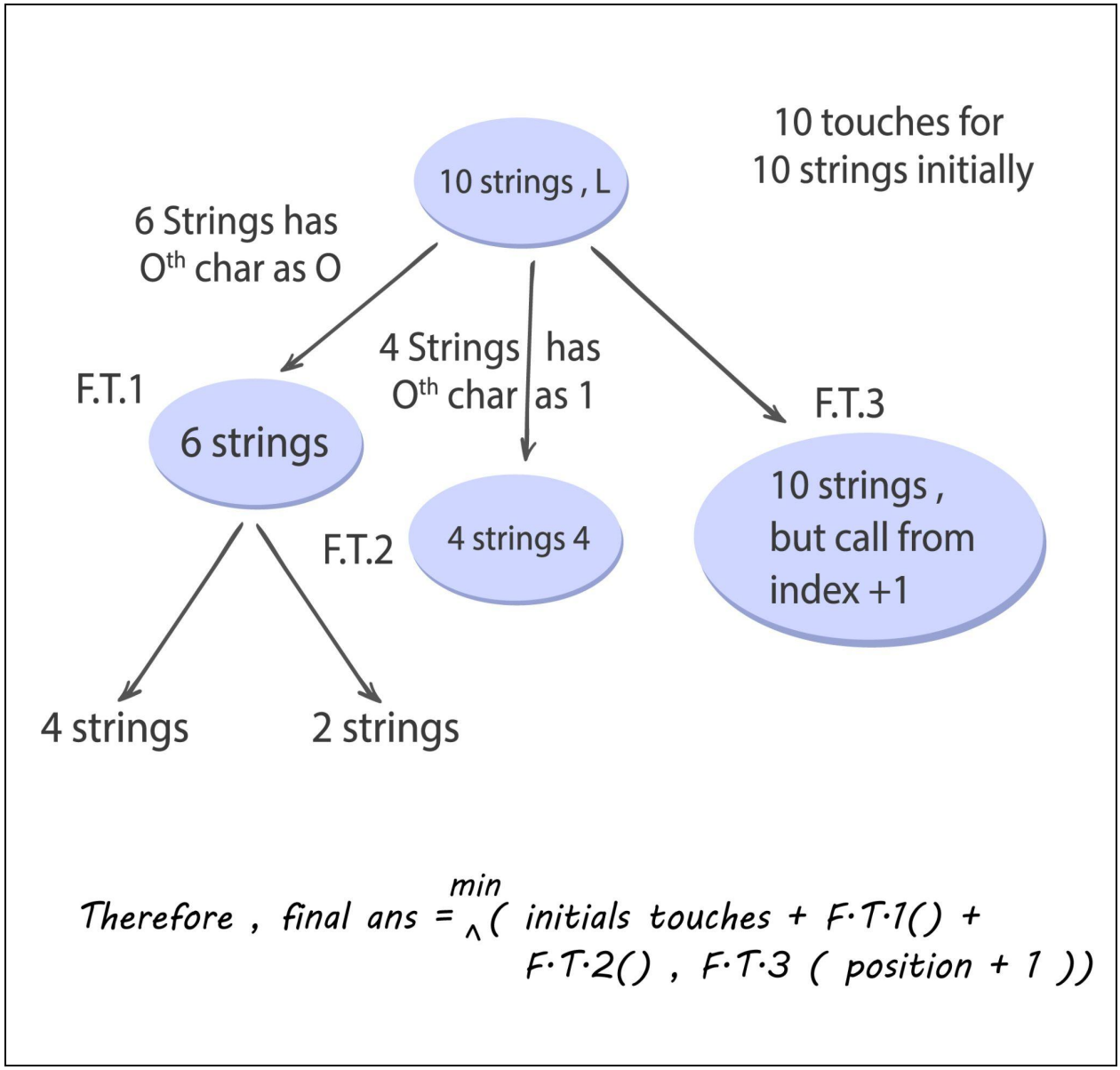
Now, at index 1, String1[1] != String2[1], therefore we know that the two strings are different and we increase the touch count from 1 to 2 for iteration over index 1.

Therefore, total touches = 2.

Let's take a look at the approach for this problem:

- Suppose there are 10 Strings of length L.
- At the 0th index there will be a total of 10 touches for 10 strings.
- Now, let's say that 6 strings have 0th character as 0 and 4 strings have 0th character as 1. And let's call the final answers to these two recursive calls as **FT1 and FT2.**
- Similarly there can be more recursive calls.
- Another case would be that we don't check the 0th character initially to avoid those 10 initial touches and call recursion from index 1 rather than 0 on all 10 strings. Let's call the final answer of this recursion call to be FT3.

- So, our final answer will be **min(initial_touches + FT1 + FT2, FT3)**.



Applying bitmasking:

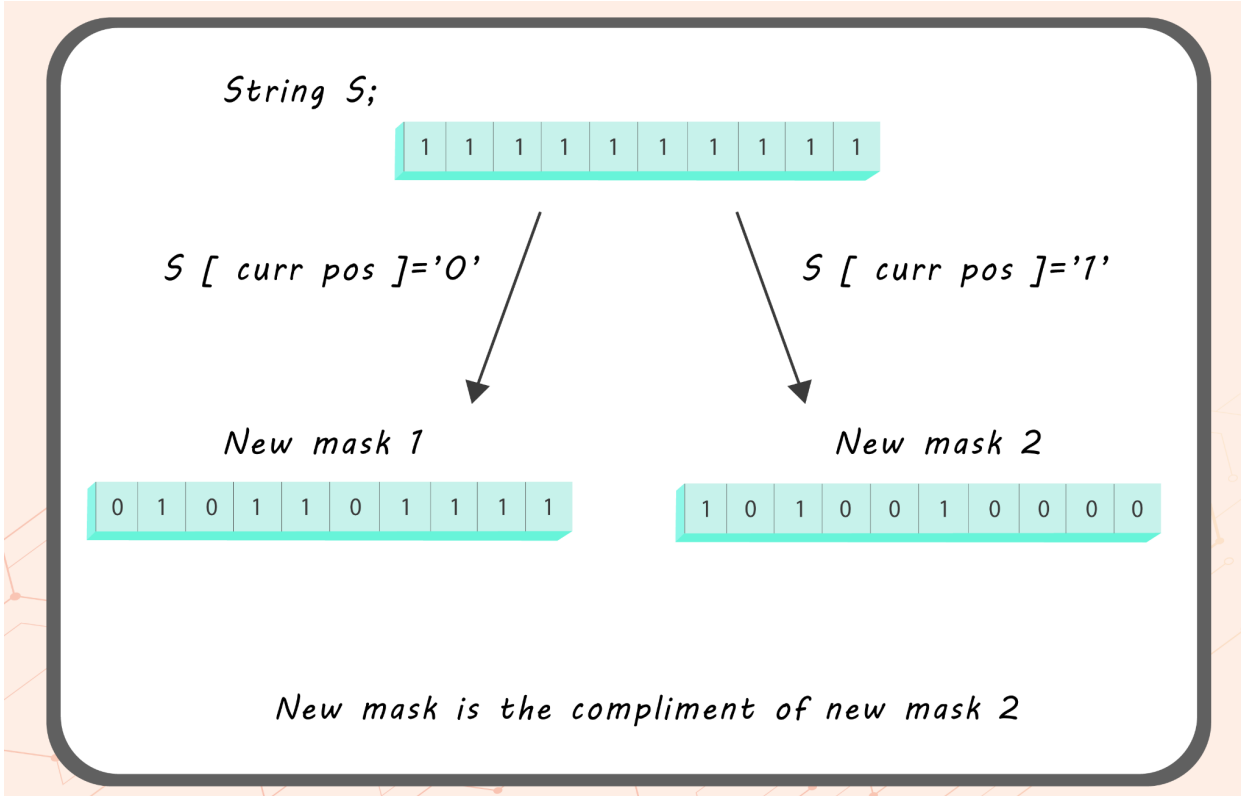
Now, we need to figure out a way with which we can keep the track of strings that we have distuinughed by touching so far.

For this, we will use the concept of BITMASKING.

The maximum number of strings that can be given in a test case according to the question constraints is 10. Thus, in this problem, we maintain a mask of 10 bits, where each bit would represent the state of the string. If i^{th} bit is not set, it means that i^{ith} string has been successfully distinguished on the basis of current check(touch). Conversely, if the ith bit is set, it means that this string has not yet been successfully distinguished on the basis of touching the value at the current position, and we need to check(touch) the following positions to distinguish them.

You may check(touch) the string either from the 0th position or the $(L - 1)^{th}$ position, where **L** is the length of the strings. This means that, while calling recursion, you may begin the recursive calls from **pos = 0** and end them using base case **pos = L**, or you may begin recursive calls from **pos = L - 1** and end them using base case **pos = -1**.

Using the mask, we can easily find how many touches will be used to distinguish the strings. How? The number of set bits in the mask will give us the number of touches that are required to further distinguish the strings. At each level we will check the value of the string at the current position. Depending on whether it is 1 or 0, we will create 2 masks recursively, which would give us the number of touches required at that level.



Now, for the base cases:

1. If mask has only one set bit then return 0
2. If position < length then return INT_MAX because we will not be able to distinguish anything.

Code:

```
/*
 Time complexity: O(2^N * ( L ))
 Space complexity: O(L * (2 ^ N))
 where N is the number of strings;
 and L is the length of the string
*/
#include <bits/stdc++.h>
using namespace std;
int dp[105][1 << 11];  // acc to the constraints, you can have max 10
```

```
strings in a single test case, hence to be on safe side we shall use 10+1
as the size.
int find_touches(int pos, int mask, vector<string> &v) {
    if (!(mask & (mask - 1)) && mask) {
        return 0;
    }
    if (pos == -1 || mask == 0) {
        return 1000000;
    }
    if (dp[pos][mask]) {
        return dp[pos][mask];
    }
    int newmask1 = 0, newmask2 = 0, touches = 0;
    for (int i = 0; i < v.size(); i++) {
        if ((mask >> i) & 1) {
            touches++;
            if (v[i][pos] == '0') {
                newmask1 |= (1 << i);
            } else { newmask2 |= (1 << i);
            }
        }
    }
    return dp[pos][mask] = min(find_touches(pos - 1, newmask1, v) +
find_touches(pos - 1, newmask2, v) + touches, find_touches(pos - 1, mask,
v));
}
int solve(int n, vector<string> v) {
    return find_touches(v[0].size() - 1, (1 << n) - 1, v);
}
int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        vector<string> v;
        memset(dp, 0, sizeof(dp));
        for (int i = 0; i < n; i++) {
            string s;
            cin >> s;
```

```
            v.push_back(s);
        }
        cout << solve(n, v) << endl;
    }
}
```

Number Theory 1

Introduction

According to Wikipedia, number theory is a branch of pure mathematics devoted primarily to the study of the integers and integer-valued functions.

Find Prime Numbers from 1 to N

Example: If we take **N = 10**, then there are a total 4 prime numbers between 1 to 10, i.e., **2, 3, 5, 7**.

Basic Approach: One thing we can do is to iterate from 1 to **N**, and apply some logic **checkPrime(i)** to find out whether **i** is a prime number or not.

For checkPrime(**n**), we can use the logic that **any prime number has only 2 factors**.

For example,

2 → 1 * 2, 2 * 1

3 → 1 * 3, 3 * 1

5 → 1 * 5, 5 * 1

So for every **n**, we can calculate how many factors of **n** are there from 1 to **n**.

Therefore, in the worst case, **n** can be equal to **N**, and it will take **O(N)** time for checkPrime() logic. Additionally, there is an outer loop iterating from 1 to **N**, hence the overall time complexity of this approach will be **O(N^2)**.

Optimized Approach:

Consider a pair **a * b = N**. Here, **a** and **b** are the factors of **N**.

For instance,

N = 12

(a, b) can be (1, 12), (2, 6), (3, 4)

On observing the equation, we can infer that the maximum value of **a** and **b** can be the square root of **N.**

√N * √N = N

Since, if both of the values go beyond **sqrt(n)**, then **a * b** would be greater than **N**.

Also, if **a** is less than **√N**, then **b** will be greater than **√N.** Similarly, if **a** is greater than **√N**, **b** will be smaller than **√N.**

We can conclude that one of the numbers is <= **sqrt(n)**, and the other one is >= **sqrt(n)**.

To prove that **N** is prime, we just need to find one of the numbers : **a** or **b**.

If no such number exists, it means that **N** is not prime.

Hence, to do the primality test, we need not run a loop till **N**, this can be done by running the loop till **√N** itself.

Example, for 25, if we check for factors from 1 to 5 then we can get all the factors without repeating any answer.

25 → 1 * 25

25 → 5 * 5

While checking the factors, remember to accordingly modify the count.

If the factor is **√N** then count = count + 1, otherwise count = count + 2.

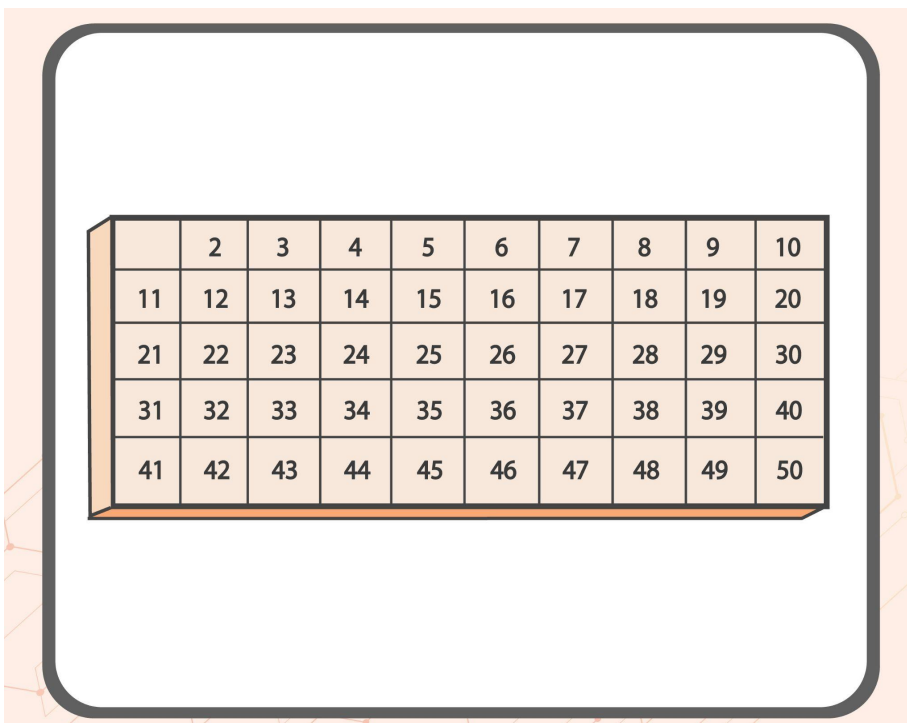
For example, while finding primes for integer 10, we encounter 2 which is a factor. We also encounter a factor 5, for which we won't check because it is greater than **√N .** Hence +2 is done in count, to consider both these factors.

Time Complexity of this approach is O(N√N).

Now , let's take a much more optimized approach in the next section.

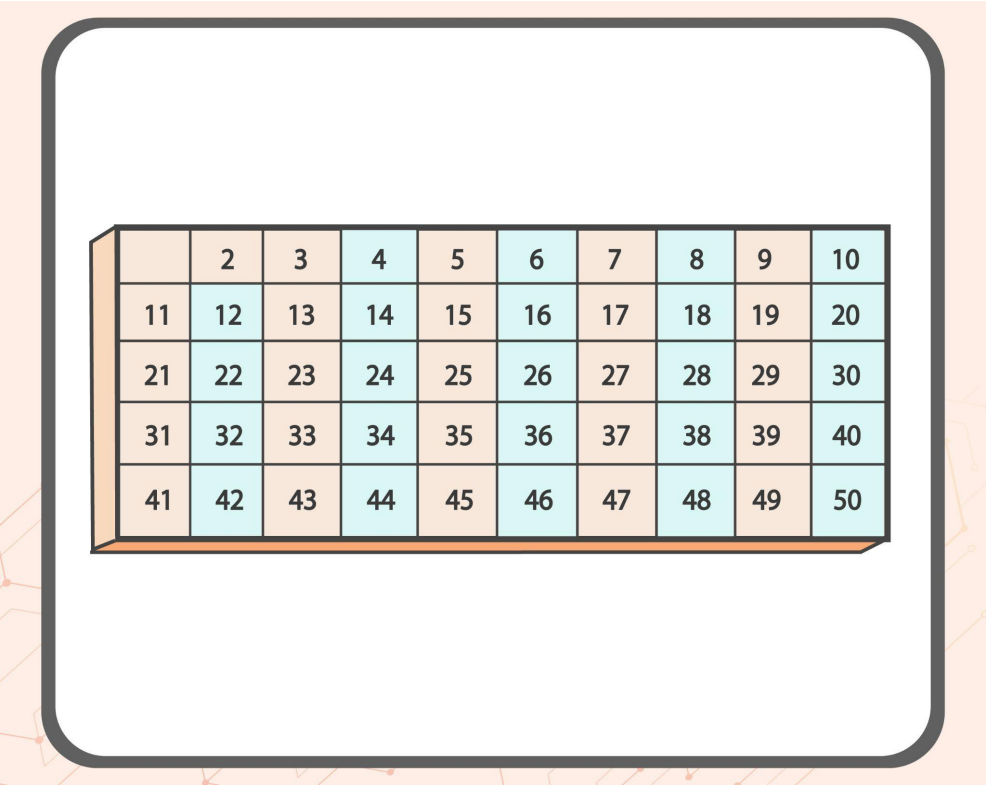
Sieve of Eratosthenes

Suppose, we have to find the prime numbers from 1 to 50, then we make an array of size 50. Then we start from 2, and on each encounter of a prime number, we mark its multiples as composite.



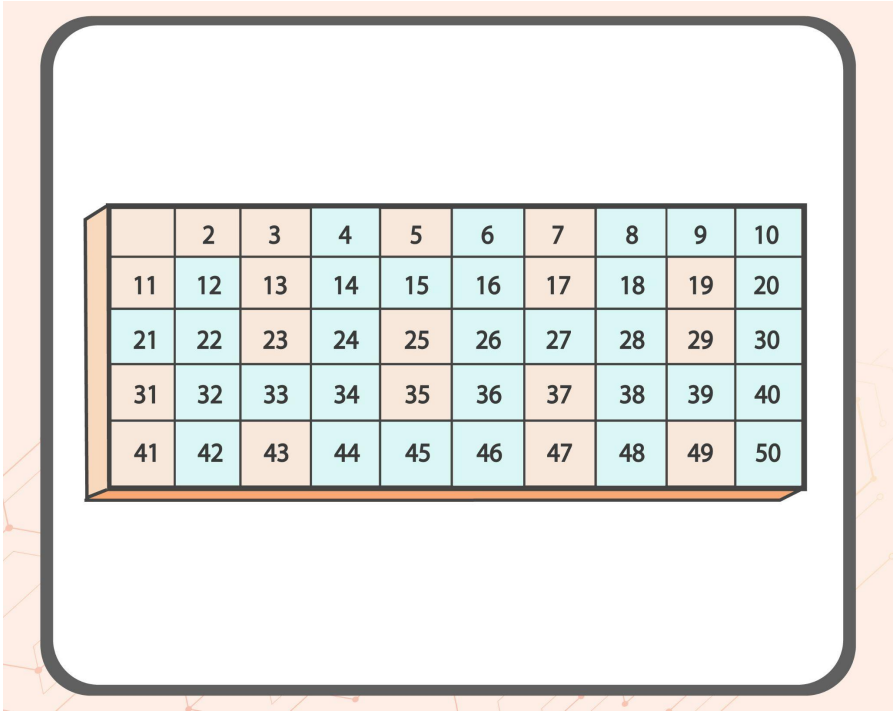
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Now , when we encounter 2 and find out that it is a prime number, we mark all the multiples of 2 in the array.



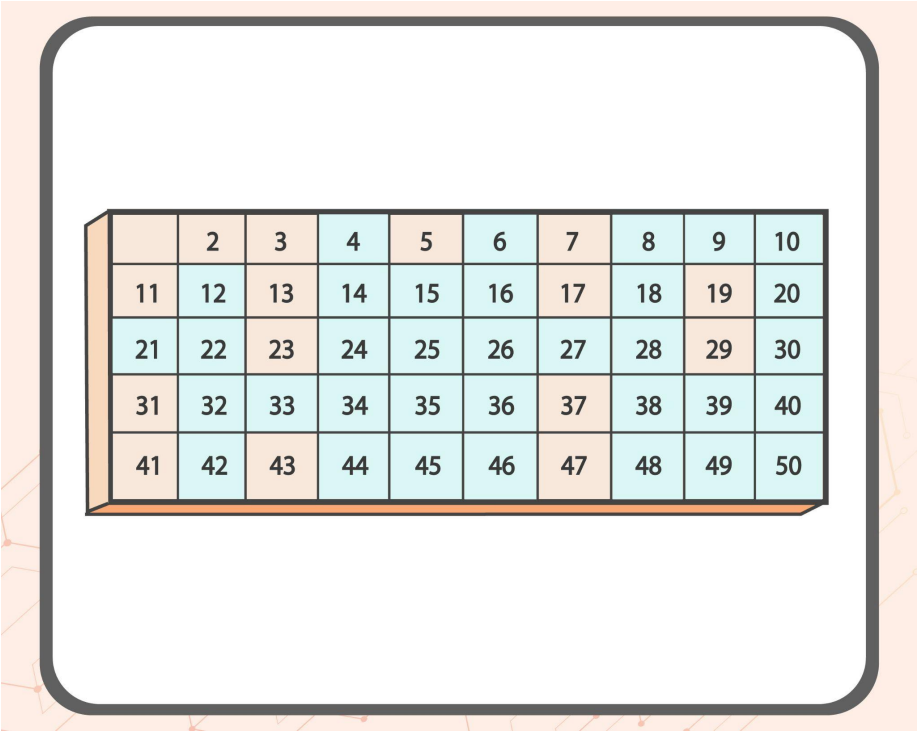
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

When we encounter 3 and find out that it is a prime number, we mark all its multiples. Now, if we look carefully, 6 is a multiple of 3 but it is already marked. This is because it was marked during marking the multiples of 2. So whenever we start marking the multiples of a prime number, we start from its square.



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Similarly for 5 and 7 and all the remaining prime numbers.



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Now, we are left with all the prime numbers as unmarked.

Time Complexity: O(N log log N)

Space Complexity: O(N)

Code:

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

int makeSieve(int n){

    bool isPrime[n+1];
    for(int i=0;i<=n;i++){
        isPrime[i] = true;
    }
    isPrime[0] = false;
    isPrime[1] = false;

    for(int i=2;i*i<=n;i++){
        if(isPrime[i] == true){
            for(int j=i*i;j<=n;j+=i){
                isPrime[j] = false;
            }
        }
    }

    int count = 0;
    for(int i=0;i<=n;i++){
        if(isPrime[i] == true){
            count++;
        }
    }
    return count;

}
int main(){
    int n;
    cin >> n;
```

```
    int nPrimes = makeSieve(n);
    cout<<nPrimes <<endl;
    return 0;
}
```

Greatest Common Divisor

Let's try to understand the algorithm to finding GCD with the help of an example:

Find GCD (10, 6).

Basic Approach: Now, one common divisor of both 10 and 6 is 1. So, we start from one and we go till 6, and for every number we iterate on, we are going to check whether it divides both 10 and 6, while maintaining a maximum variable.

Now, let's say that we have **GCD (16131, 13733).** In this case we have to iterate till 13733 which is a very huge number, so let's try to come up with a better solution.

Euclid's Algorithm:

Euclid's algorithm states that **GCD (A, B) = GCD (B, A%B)**, given that **A > B**.

In the above algorithm, GCD (B, A%B) is a recursive call, so our **base case is when the second parameter i.e, A%B becomes zero, we return B.**

For example,

GCD (16, 10) = GCD (10, 6)

GCD (10, 6) = GCD (6, 4)

GCD (6, 4) = GCD (4, 2)

GCD (4, 2) = GCD (2, 0) → Ans = 2

Time Complexity of Euclid's algorithm :

GCD (A, B) → GCD (B, A%B)

Let's say that **R = A%B**

GCD (B, R) → GCD (R, B%R)

Now, we have **0 <= R <= B-1**

Also, **R = A%B = A - B * int (A/B)**

Example, 10 % 7 = 10 - 7 * int (10/7) = 10 - 7*1 = 3

Also, the minimum value of A-B = 1, so **R <= A-B** and we have **R < B**.

Adding the two equations, we get **R < A/2.**

So at every step the value of the second parameter/ number get's halved. Therefore the time complexity of this algorithm is **O (log (max (a,b)).**

Diophantine Equations

A Diophantine equation is a polynomial equation, usually in two or more unknowns, for which the integral solutions exist.

For example, in the equation **a * x + b * y = c**, if **a = 3, b = 6, c = 9**, then there exist **x = 1** and **y = 1**.

For linear Diophantine equation equations, integral solutions exist if and only if the GCD of coefficients of the two variables divides the constant term perfectly. In other words the integral solution exists if GCD(a ,b) divides c.

Extended Euclid's Algorithm

While the Euclidean algorithm calculates only the greatest common divisor (GCD) of two integers a and b, the extended version also finds a way to represent GCD in terms of a and b, i.e. coefficients x and y for which:

$$a*x + b*y = gcd(a, b)$$

It's important to note, that we can always find such a representation, for instance gcd(55, 80) = 5, therefore we can represent 5 as a linear combination with the terms 55 and 80:

$$55*3 + 80*(-2) = 5$$

We have discussed these equations in the previous topic, Diophantine Equations.

Algorithm:

We will denote the GCD of **a** and **b** with **g** in this section.

The changes to the original algorithm are very simple. If we recall the algorithm, we can see that the recursive algorithm ends with **b = 0** and **a = g (base case, as when b = 0, GDC is g = a)**. For these parameters we can easily find coefficients, namely

$$g * 1 + 0 * 0 = g.$$

Starting from these coefficients **(x, y) = (1, 0)**, we can go backwards up the recursive calls. All we need to do is to figure out how the coefficients **x** and **y** change during the transition from **(a, b)** to **(b, a % b)**.

Let us assume we found the coefficients (x_1, y_1) for **(b, a % b)**:

$$b*x_1 + (a \% b)*y_1 = g$$

and we want to find the pair **(x, y)** for **(a, b)**:

$$a*x + b*y = g$$

We can represent **(a % b)** as:

$$(a \% b) = a - [a/b] * b$$

Substituting this expression in the coefficient equation of (x_1, y_1) gives:

$$g = b * x_1 + (a \% b) * y_1$$

$$= b * x_1 + (a - [a/b] * b) * y_1$$

and after rearranging the terms we get:

$$g = a * y_1 + b * (x_1 - y_1 * [a/b])$$

Thus we have found the values of x and y.

$$x = y_1$$

$$y = x_1 - y_1 * [a/b]$$

Code:

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

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

    // Update x and y using results of
```

```
    // recursive call
    *x = y1;
    *y = x1 - y1 * (a/b);


    return gcd;
}
```

The recursive function above returns the GCD and the values of coefficients to x and y (which are passed by reference to the function).

This implementation of the extended Euclidean algorithm produces correct results for negative integers as well.

Iterative Code:

It's also possible to write the Extended Euclidean algorithm in an iterative way. Because it avoids recursion, the code will run a little bit faster than the recursive one.

```
int gcdExtended(int a, int b, int *x, int *y)
{
    *x = 1;
    *y = 0;
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}
```

No algorithm is complete without some practice. Here are links to a few practice problems that you should try.

- 10104 - Euclid Problem
- GYM - (J) Once Upon A Time
- UVA - 12775 - Gift Dilemma

Multiplicative Modulo Inverse

Let's try to understand this problem with the help of an example,

Given: (A * B) % m = 1, A = 5, m = 17

To find: B (multiplicative modulo inverse)

Steps:

(5 * 1) % 17 = 5

(5 * 2) % 17 = 10

(5 * 3) % 17 = 15

(5 * 4) % 17 = 3

(5 * 5) % 17 = 8

(5 * 6) % 17 = 13

(5 * 7) % 17 = 1

Solution: This problem can be solved using the extended euclid algorithm.

(A * B) % m = 1

Since A * B ≅ 1 (% m)

(A * B - 1) % m = 0

(A * B - 1) = m*q where q is an integer.

(A * B) - m * q = 1

(A * B) + m *(-q) = 1

(A * B) + m *(Q) = 1

Now, according to **linear diophantine equations**,

B will exist only **if GCD (A, M) divides 1**.

i.e., GCD (A, M) = 1

=> A, M should be co-prime.

Code:

```
Triplet extendedEuclid(int a, int b){

    // base case
    if(b == 0){
        Triplet ans;
        ans.gcd = a;
        ans.x = 1;
        ans.y = 0;
        return ans;
    }

    // extended euclid algorithm
    Triplet smallAns = extendedEuclid(b, a%b);
    Triplet ans;
    ans.gcd = smallAns.gcd;
    ans.x = smallAns.y
    ans.y = smallAns.x - (a/b)*smallAns.y;
    return ans;
}
int mmInverse(int a, int m){
    Triplet ans = extendedEuclid(a, m);
```

```
    return ans.x;
}
```

Number Theory 2

Sachin and Varun Problem

Problem Statement: Given two weights of **a** and **b** units, in how many different ways you can achieve a weight of **d** units using only the given weights? Any of the given weights can be used any number of times (including 0 number of times).

Explanation:

Let's take an example to understand the problem more clearly:

a = 2

b = 3

d = 7

So, one possible way could be **2 * 2 + 3 = 7**

Another could be **2 * 5 - 3 = 7.**

So, basically we have to find the solutions to a*x + b*y = d, where x and y are always greater than zero.

Additional Info: For the maximum value of **x**, we need to keep y as zero i.e, **x <= d/a**.

Similarly, **y <= d/b**.

Approach 1: One approach will be to iterate over all possible x and y values. Time complexity of this approach will be **O (d^2 / a * b).** Since in the question constraints,

we are given that in the worst case **a** would be equal to **b.** Hence the worst case time complexity would be **O (d^2 / b^2).**

Approach 2: Let us understand the equations a bit more.

$$a * x + b * y = d$$

$$a * x = d - b * y$$

We know that for one value of **y** there is at-most one **x** which will satisfy this equation. So, iterate over all values of **y** in the range and check if **d – b * y** is divisible by **a** or not.

Similarly we could also check that is **d – a * x** divisible by **b** or not as

$$b * y = d - a * x$$

Thus, the time complexity of this approach is **O(d/b)** or **O(d/a).** This is better than the previous solution's complexity but still not good enough to pass within the time limit.

Approach 3: Beginning of Optimized Approach

In the above approach, the value of **y** is changing. Hence, if we could find the smallest value for **y** for which we get our conditions satisfied, we can easily find out our next value of **y**. That is, we can add integer multiples of **a** to satisfy the conditions.

Let y_1 be the smallest non-negative number such that $d - b * y_1$ is divisible by **a**. We can say that y_1 is the first value of **y** such that **d – b * y** is divisible by a.

If $d - b^*y$ is divisible by **a**, then $d - b^*y + a$ is also divisible by **a**.

If we have $\mathbf{y_1}$, the next value of **y** will be $\mathbf{y_1 + a}$. Let us prove this.

Proof: Let $\mathbf{y = y_1 + a}$. Then,

$$\mathbf{d - b * y = d - b - (y_1 + a)}$$

$$\mathbf{= d - b * y_1 - b * a}$$

We know that $\mathbf{d - b * y_1}$ is divisible by **a.** Additionally, **b * a** is also divisible by **a.** Hence we may conclude that $\mathbf{d - b * (y_1 + a)}$ will also be divisible by **a.**

Thus the values of y will be $\mathbf{y_1, y_1 + a, y_1 + 2a, y_1 + 3a, y_1 + 4a, \ldots, y_1 + n * a,}$ where

$$\mathbf{y_1 + n * a <= d/b}$$

The value of **y** seems to be an Arithmetic Progression. So we can easily calculate the number of terms in the Arithmetic progression using $\mathbf{y_1 + n * a <= d/b}$ equation.

$\mathbf{n = (d/b - y_1) / a.}$

So the number of terms in the Arithmetic progression is equal to n + 1.

Now the only part that's left is to find the value of $\mathbf{y_1}$. An obvious solution is to iterate over all values of **y** and break as soon as we find the first value such that **d – b * y** is divisible by **a**. Time complexity of this solution is **O(d/b)** in the worst case. But this solution is faster than previous solutions in other cases.

Approach 4: Optimisation over Approach 3

A small optimization is to divide both sides of the equation **a * x + b * y = d** by the GCD(a, b). Why ?? Think about it.

a * x + b * y = d is a linear Diophantine equation in two variables. The solution of this equation exists if and only if d is divisible by the GCD(a, b).

We can compute the first value of $\mathbf{y_1}$ using Modulo Inverse (Extended Euclid).

We have, **a * x + b * y = d.**

$$y = (d - a*x) / b$$

Taking modulo **a** on both sides we get

$$y_1 = y \% a = [(d - a*x) / b] \% a$$

Now we need to check if $\mathbf{(d - b * y_1)}$ is divisible by a.

We can compute $\mathbf{y_1}$ in **O(log(max(a, b)).** The number of terms used can be computed using the formula of the previous solution $\mathbf{(n = (d/b - y_1) / a)}$ in **O(1)**.

Hence, the overall time complexity of this solution is **O(log(max(a, b))).**

Code:

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

using namespace std;
typedef long long ll;

class Triplet{
public:
    ll gcd;
    ll x;
    ll y;
};
Triplet gcdExtendedEuclid(ll a,ll b){
    //Base Case
    if(b==0){
        Triplet myAns;
        myAns.gcd = a;
        myAns.x = 1;
        myAns.y = 0;
```

```
        return myAns;

    }
    Triplet smallAns = gcdExtendedEuclid(b,a%b);
    //Extended euclid says

    Triplet myAns;
    myAns.gcd = smallAns.gcd;
    myAns.x  = smallAns.y;
    myAns.y = (smallAns.x - ((a/b)*(smallAns.y)));
    return myAns;
}

ll modInverse(ll A, ll M)
{
    ll x = gcdExtendedEuclid(A, M).x;
    return (x % M + M) % M;
}

int main(){

    int t;
    cin>>t;
    while(t--){
        ll a,b,d;

        cin >> a >> b >> d;
        ll g = __gcd(a,b);
//Special Cases
        if(d%g){
            cout << 0 << endl;
            continue;
        }

        if(d == 0){
            cout << 1 << endl;
            continue;
        }
```

```
        a/=g;
        b/=g;
        d/=g;


        ll y1  = ((d%a) * modInverse(b,a) ) %a;
        ll firstValue = d/b;

        if(d < y1*b){
            cout << 0 <<endl;
            continue;
        }

        ll n = (firstValue - y1)/a;

        ll ans = n +1 ;
        cout << ans <<endl;

    }

    return 0;
}
```

Advanced GCD

Problem Statement: Write a more efficient way to calculate GCD(a, b).

$0 < b < 10^{250}$ and $0 < a < 40000$

Explanation: Our function to calculate GCD earlier was:

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

We know that **gcd(b, a)** is the same as **gcd(a, b%a)**.

Thus, suppose had we needed to calculate **gcd(10^{240}, 40),** we may write this as

$$\mathbf{gcd(40,\ 10^{240}\ \%\ 40)\ \text{------}\ Eq1}$$

Now, since **b** can be very large, it can't be stored in integers. So, we need to think of a different way. We can store **b** in a string. We need to figure out how to take the modulo of a string. Taking modulo with a small number like **a** would enable us to store the resultant in an integer variable, thus satisfying the usage of the above formula **Eq1.**

Important Note: **(a * b) % m** is same as **(a%m * b%m) % m.**

Keeping in mind the above note, we may break our string into effective integers and calculate modulo. Let us understand the process. Suppose we have 104. Let us deconstruct this integer.

104 = 10*10 + 4

If we are supposed to find 104 % m, we may write this like

((10*10) % m + (4 % m)) % m

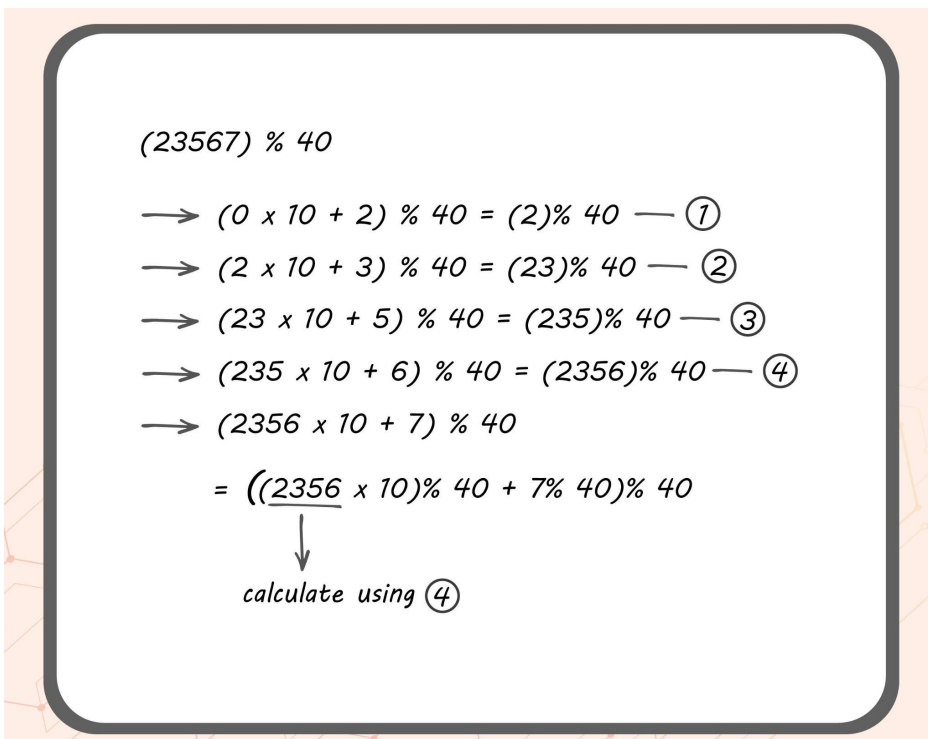
((100) % m + (4 % m)) % m

Similarly, we may deconstruct 100 like

(10 * 10) + 0

This 10 can be deconstructed into **1 * 10 + 0.**

Now, instead of breaking this big number into a small number, let's have a look at it in the reverse direction. Given a string, we shall take each digit separately from the start, and begin forming our big integer, all the while, performing modulo on it to maintain it inside an integer variable. Example for this is explained below in the image.



Extracting each digit from a string can be easily done using a for loop. In fact, you may use the same for loop for the purpose of building your larger integer.

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

{

b = (10 * b + s[i] - '0') % a;

}

Let us now write the code for the same.

Code:

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

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

    int t;
    cin >> t;

    while (t--)
    {
        int a;
        string s;
```

```
        cin >> a >> s;

        if (a == 0)
        {
            cout << s << "\n";
            continue;
        }

        int n = s.size();
        int b = 0;
        for (int i = 0; i < n; i++)
        {
            b = (10 * b + s[i] - '0') % a;
        }
        int ans = gcd(a, b);
        cout << ans << endl;
    }

    return 0;
}
```

Divisors of a Factorial

Problem Statement: We are given **N!** and we have to tell the total number of divisors for it.

Explanation:

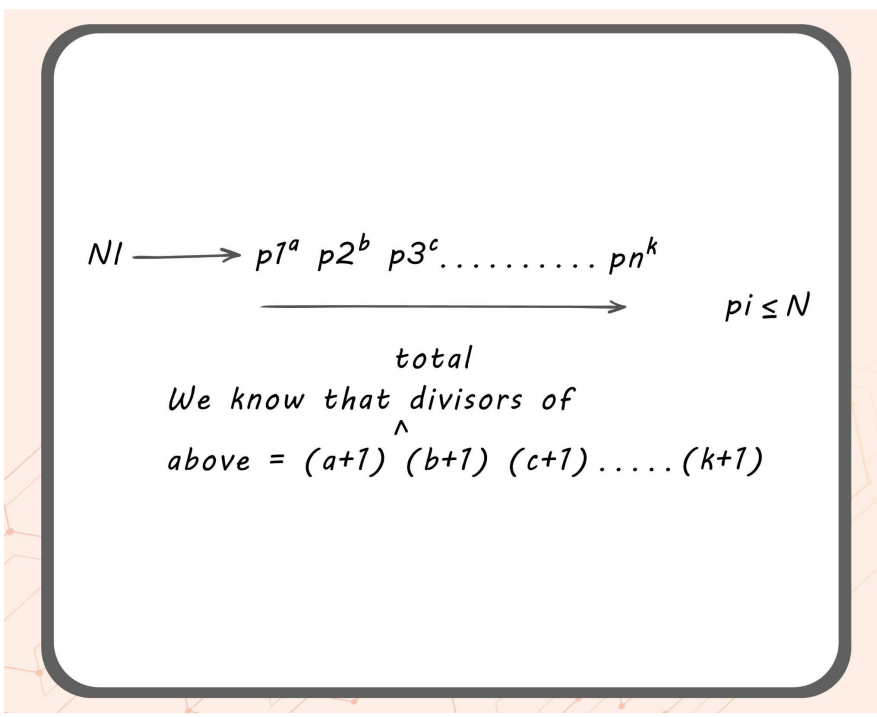
Basic Approach: The basic approach to solve this problem would be to calculate the value of **N!** Then we traverse from 1 to N! and count the number of divisors.

Time complexity of this will be **O(N!)** which will give time limit exceeded.

Optimized Approach:

We know that any number can be expressed in the form of prime numbers, that is, $\mathbf{N \rightarrow 2^a 3^b 5^c ...}$

For example, $\mathbf{10 \rightarrow 2^1 5^1}$

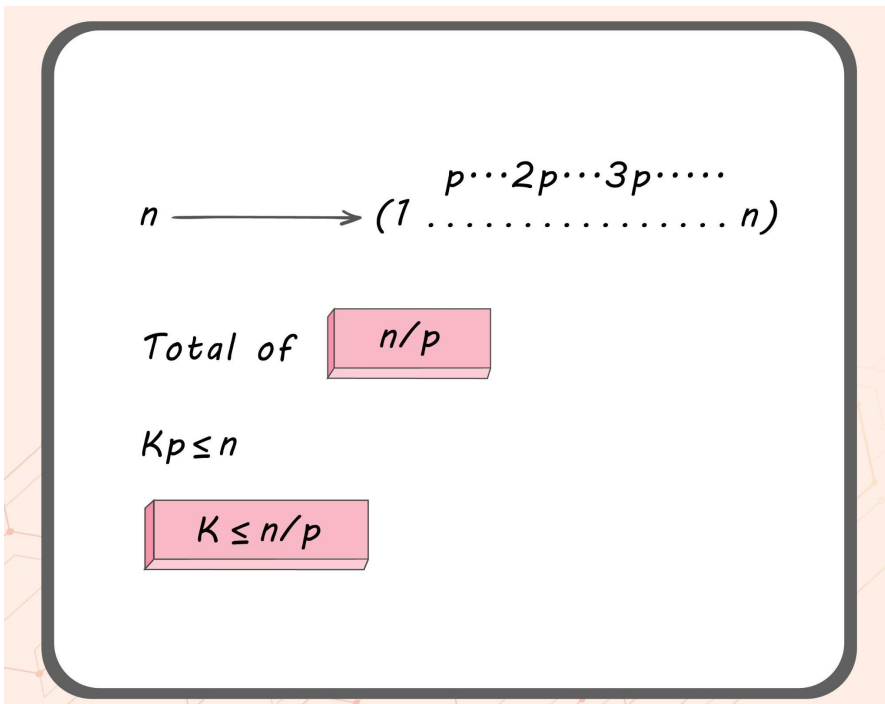


Now, how to express any number in the form of prime numbers ?

Suppose we are given 100! It is obvious that it won't contain any prime number that is greater than 100.

To simplify the problem even more, we can think of it in a way that we just need to calculate the powers of the prime numbers less than 100 that make up 100!

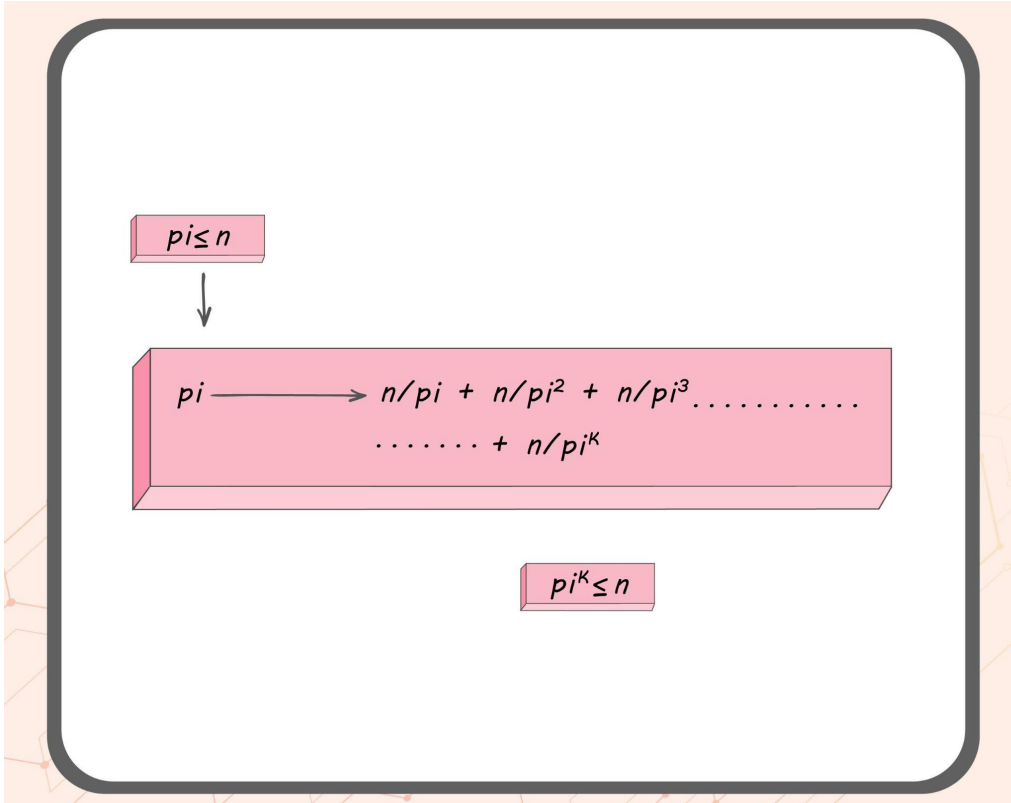
Now, we need to calculate the count of prime number p_i.



So let's say that **2** comes **n/p** times

Therefore $\mathbf{2^2}$ comes $\mathbf{n/p^2}$ times.

$\mathbf{2^3}$ comes $\mathbf{n/p^3}$ times and so on ...



We shall pre compute the values of primes using Sieve to save time. Let us now write the code for the same.

Code :

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

#define MAX 500001
#define MOD 1000000007
#define pb push_back

typedef long long ll;

vector<int>* sieve(){
    bool isPrime[MAX];

    vector<int>* primes = new vector<int>();
```

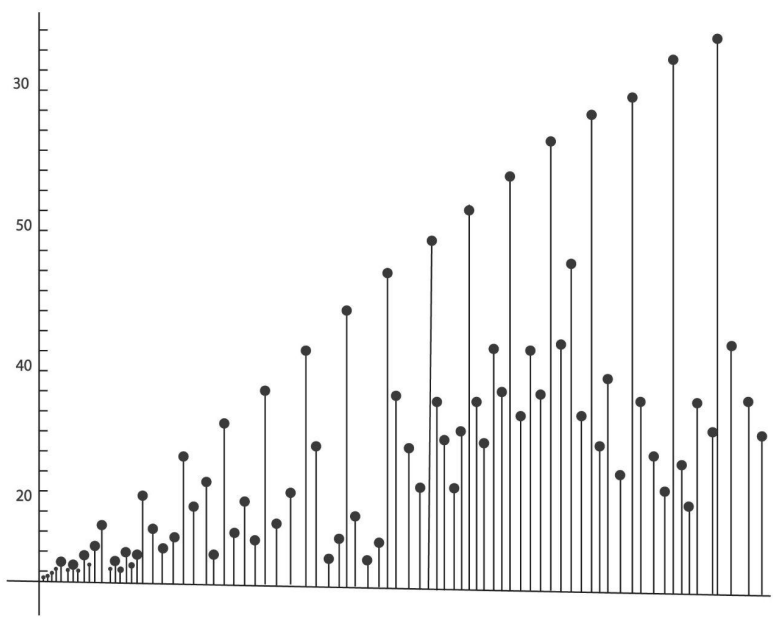
```
    for(int i=2;i<=MAX;i++){
        isPrime[i] = true;
    }
    for(int i=2;i*i<=MAX;i++){

        if(isPrime[i]){
            for(int j=i*i;j<=MAX;j+=i){
                isPrime[j] = false;
            }
        }
    }

    primes->pb(2);
    for(int i=3;i<=MAX;i+=2){
        if(isPrime[i]){
            primes->pb(i);
        }
    }
    return primes;

}

ll divisors(int n,vector<int>* & primes){


    ll result = 1;
    for(int i=0;primes->at(i)<=n;i++){
        int k = primes->at(i);
        ll count = 0;
        while((n/k)!=0){
            count = (count + (n/k))%MOD;
            k = k*primes->at(i);
        }
        result = (result * ((count+1)%MOD))%MOD;
    }
    return result;
}
int main(){
    vector<int>* primes = sieve();
    int t;
```

```
    cin >> t;
    while(t--){
        int n;
        cin >> n;
        cout << divisors(n,primes) << endl;
    }
    return 0;
}
```

Number Theory 3

Euler's Totient Function

Euler's totient function gives us the count of positive integers less than n that are co prime to n. That is, **ϕ(n)** is the number of **m ∈ N** such that **1 ≤ m < n** and **gcd(m,n)=1**.



The value of ϕ (n) for $n \leq 100$

For example, Find **ϕ(15)**.

Among positive numbers less than 15, eliminate multiples of 3 or 5, which are {3, 5, 6, 9, 10, 12}. The remaining numbers are **{1,2,4,7,8,11,13,14}**, so **ϕ(15) = 8**.

According to a property of euler totient function :

ϕ(a * b) = ϕ(a) * ϕ(b), given that gcd(a, b) = 1

And we know that any number can be written using prime numbers as **N → 2^a 3^b 5^c**

Therefore, $\phi(n) = \phi(p_1^a * p_2^b * p_3^c \ldots\ldots\ldots\ldots * p_k^k)$

$$\rightarrow \quad \phi(p_1^a) * \phi(p_2^b) * \phi(p_3^c) \ldots\ldots\ldots\ldots * \phi(p_k^k)$$

Now, $\phi(p^a)$ means that count of numbers from 1 to p^a that are co-prime to p^a.

And those will be excluding the multiples of p, that is, 1, p, 2p, 3p and so on ..

The total count of these multiples will be p^{a-1}. Now,

$$\phi(p^a) = p^a - p^{a-1}$$

$$\phi(p^a) = p^a (1 - (1/p))$$

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right)$$

Where k is the number of distinct prime factors.

Basic Approach :

The simple solution is to iterate through all numbers from 1 to N-1 and count numbers whose GCD(Greatest Common Divisor) with N is 1 and we can find the GCD of two numbers in an efficient manner using the Euclidean algorithm.

Approach 2:

The better solution is to use Euler's product formula which states that the total number of integers between 1 and N-1 inclusive which are coprime to N can be found using this formula Euler Product Formula
N*(1-(1/p1))*(1-(1/p2))*.....*(1-(1/pk)) where p1,p2...pk are the prime factors of N.

So now the problem is to find all the prime factors and use the above formula.

To find the prime factors and count the number of integers between 1 and N which are coprime to N below are the steps.

Steps:

1. Initially declare a variable total = N.
2. Then start a loop from p = 2 to sqrt(N) and do the following:
    1. if(N%p==0) then
        1. While p divides N:
            1. Divide N by p.
        2. total =total- total/p
3. If N is a prime number and is greater than 1, then:
    1. total = total-total/N
4. At last, return the total.

OR, we could also use a sieve instead of carrying out the above steps.

Code:

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


void eulerPhi(int n){

    int phi[n+1];

    for(int i=1;i<=n;i++){
        phi[i] = i;
    }

    for(int i=2;i<=n;i++){
        if(phi[i] == i){
            phi[i] = i-1;
            for(int j=2*i;j<=n;j+=i){
                phi[j] = (phi[j]*(i-1))/i;
            }
        }
    }

    for(int i=1;i<=n;i++){
        cout << "Euler Totient Phi For " << i << "Is :" << phi[i]<<endl;
    }

}
int main(){

    eulerPhi(10);
    return 0;
}
```

Sum of LCM

Problem Statement : Given n, calculate and print the sum :

LCM(1,n) + LCM(2,n) + .. + LCM(n,n)

where LCM(i,n) denotes the Least Common Multiple of the integers i and n.

Explanation:

$$S = L{\cdot}C{\cdot}M\,(1,n) + L{\cdot}C{\cdot}M\,(2,n) + \cdots + L{\cdot}C{\cdot}M(n,n) \longrightarrow n$$

$$S-n = L{\cdot}C{\cdot}M\,(1,n) + L{\cdot}C{\cdot}M\,(2,n) + \cdots L{\cdot}C{\cdot}M(n-1,n)$$

$$S-n = L{\cdot}C{\cdot}M\,(n-1,n) + \cdots\cdots\cdots L{\cdot}C{\cdot}M\,(1,n)$$

Also, we know that,

1) $$L{\cdot}C{\cdot}M\,(a,n) = \frac{a * n}{gcd\,(a,n)}$$

2) $$L{\cdot}C{\cdot}M\,(n-a\,,\,n) = \frac{n * (n-a)}{gcd\,(n-a\,,n)}$$

These two values are equal

If we add 1 and 2,

$$\frac{\cancel{a * n} + n^2 - \cancel{a * n}}{gcd\,(a,n)} = \frac{n^2}{gcd\,(a,n)}$$

Therefore,

$$2s - 2n = \sum_{i=1}^{n-1} \frac{= n^2}{gcd\,(i,n)}$$

Segmented Sieve

We have to calculate the number of primes in a particular range from L to R. Now earlier we solved this question using a sieve of size R, but if R is given to be greater than 10^8 we can't use that approach. But what we can do is make a sieve of size L - R which is well given in our acceptable range, and adjust the indexing accordingly.

Everything else will work according to the previous discussion of sieve of eratosthenes.

Code:

```
#include<bits/stdc++.h>
using namespace std;
#define MAX 100001
vector<int>* sieve(){

    bool isPrime[MAX];
    for(int i=0;i<MAX;i++){
        isPrime[i] = true;
    }
    for(int i=2;i*i<MAX;i++){
        if(isPrime[i]){
            for(int j=i*i;j<MAX;j+=i){
                isPrime[j] = false;
            }
        }
    }
    vector<int>* primes = new vector<int>();
    primes->push_back(2);
    for(int i=3;i<MAX;i+=2){
        if(isPrime[i]){
            primes->push_back(i);
        }
    }
    return primes;
}
void printPrimes(long long l,long long r,vector<int>* & primes){
```

```
    bool isPrime[r-l+1];

    for(int i=0;i<=r-l;i++){
        isPrime[i] = true;
    }

    for(int i=0;primes->at(i)*(long long)primes->at(i) <= r;i++){
        int currPrime = primes->at(i);
        // Just smaller or equal value to l
        long long base = (l/(currPrime))*(currPrime);
        if(base < l){
            base = base + currPrime;
        }

        // Mark all mutliples within L To R as false
        for(long long j = base;j<=r ;j+= currPrime){
            isPrime[j-l] = false;
        }

        // There may be a case where base is itself a prime number .
        if(base == currPrime){
            isPrime[base-l] = true;
        }
    }

    for(int i=0;i<=r-l;i++){
        if(isPrime[i] == true){
            cout << i + l << endl;
        }
    }
}
int main(){
    vector<int>* primes = sieve();
    int t;
    cin >> t;
    while(t--){
        long long l,r;
        cin>>l>>r;
        printPrimes(l,r,primes);
    }
```

```
    return 0;
}
```

Optimized Power Function

We have to find out the **pow (x, n)**.

Basic Approach : We can solve this problem recursively by **x * pow (x, n)**.

Optimized Approach :

If n is even :

Val = pow (x, n)

Pow (x, n) = Val * Val

Example: $\mathbf{x^8 = x^4 * x^4}$

If n is odd :

Val = pow (x, n/2)

pow (x, n) = Val * Val * x

Example: $\mathbf{x^9 = x^4 * x^4 * x}$

Time complexity of this approach will be O(log(n)).

Modular Exponentiation

If in the previous topic of optimized power function, we have to find $\mathbf{2^{1024}}$ then it will be out of the range of integers or long long.

We can prevent the answers to go out of range using modulo arithmetic

(a * b) % c = ((a % c) * (b % c)) % c

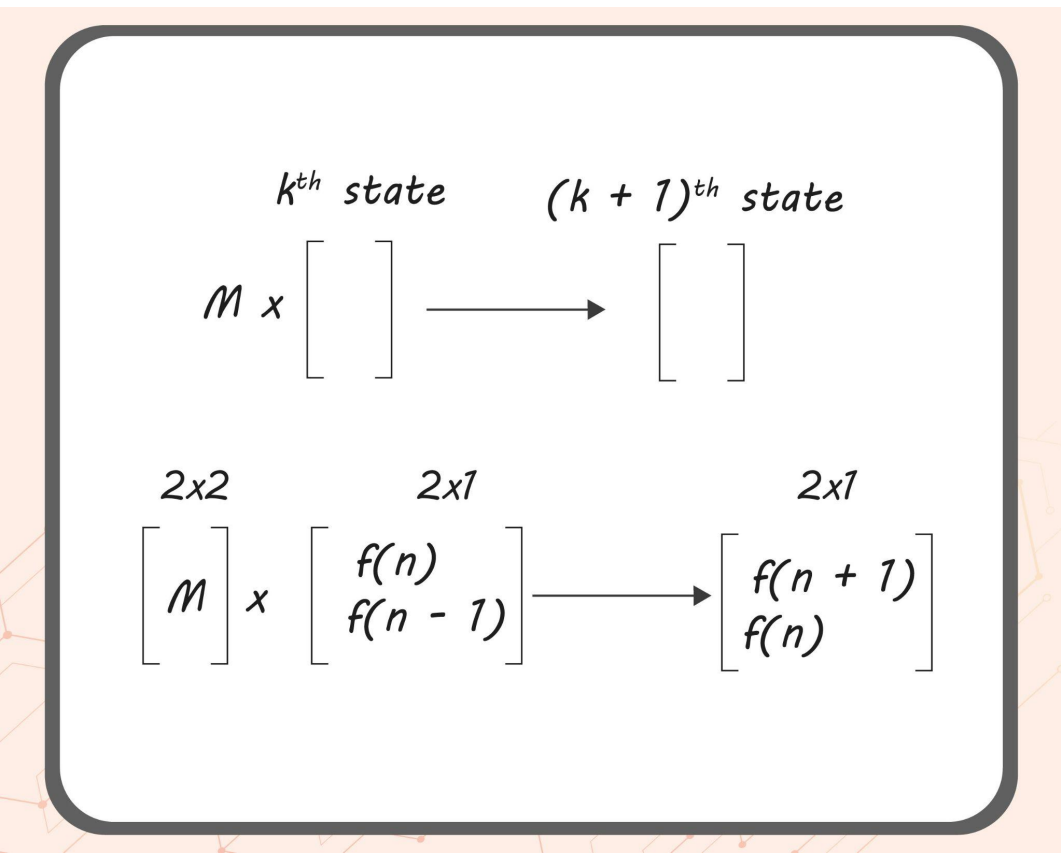
Number Theory - 4

Matrix Exponentiation

Matrix exponentiation says that we have to find a matrix such that our recurrence relation at k^{th} state when multiplied by the matrix gives the $(k + 1)^{th}$ state of the recurrence relation.

Example : **f (n) = f (n-1) + f (n-2)**

Since there are two unknowns therefore our matrix will be of size **2 X 2**.

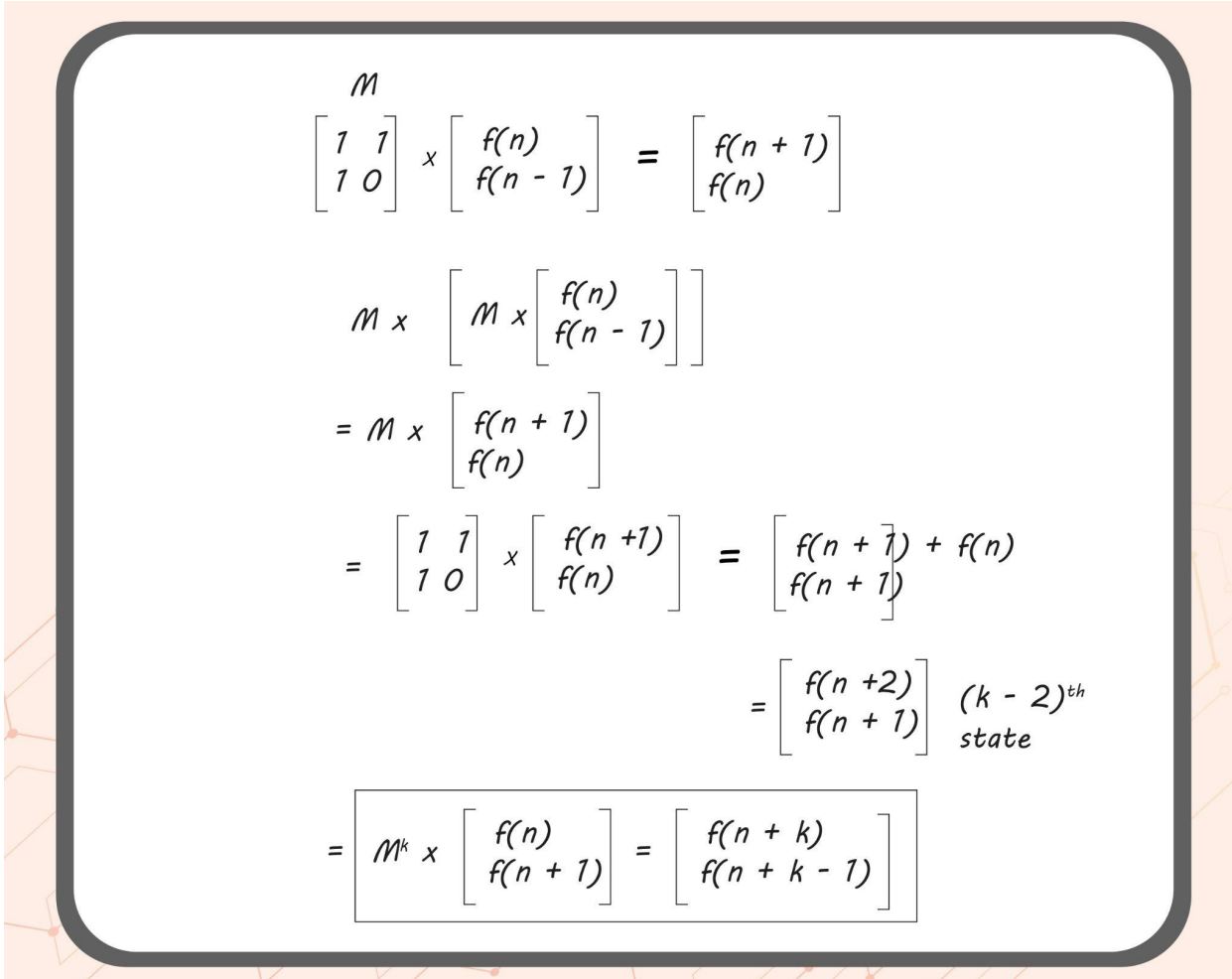


$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \end{bmatrix}$$

$$a \times f(n) + b \times f(n-1) = f(n+1)$$

$$c \times f(n) + d \times f(n-1) = f(n)$$

Now, since **f (n) = f (n -1) + f (n - 2)** can be written as **f (n + 1) = f (n) + f (n - 1)**, therefore, **a , b = 1**. Also, **c = 1 and d = 0**.



Time complexity of finding Nth fibonacci using matrix exponentiation will be **O(log(N))**.

Code:

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


void multiply(int A[2][2],int M[2][2]){

    int firstValue = A[0][0] * M[0][0] + A[0][1] * M[1][0];
    int secondValue = A[0][0] * M[0][1] + A[0][1] * M[1][1];
    int thirdValue = A[1][0] * M[0][0] + A[1][1] * M[1][0];
```

```
    int fourthValue = A[1][0] * M[0][1] + A[1][1] * M[1][1];

    A[0][0] =firstValue;
    A[0][1] = secondValue;
    A[1][0] = thirdValue;
    A[1][1] = fourthValue;

}
void power(int A[2][2],int n){
    if(n==1){
        return;
    }
    power(A,n/2);
    multiply(A,A);
    if(n%2 !=0){
        int F[2][2] = {{1,1},{1,0}};
        multiply(A,F);
    }
}
int getFibonacci(int n){
    if(n==0 || n==1){
        return n;
    }
    int A[2][2] = {{1,1},{1,0}};
    power(A,n-1);
    return A[0][0];
}
int main(){
    int n;
    cin >> n;
    cout << getFibonacci(n)<<endl;
    return 0;
}
```

Some examples of Recurrence Relations

1. **f (n) = a * f (n-1) + b * f (n-2).**

$$\begin{array}{ccc} & k^{th} \text{ state} & (k+1)^{th} \text{ state} \\ M \times & \begin{bmatrix} \\ \\ \end{bmatrix} & = \begin{bmatrix} \\ \\ \end{bmatrix} \end{array}$$

$$\begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \end{bmatrix}$$

2. **f (n) = f (n-1) + f (n-2) + c**

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f(n) \\ f(n-1) \\ c \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \\ c \end{bmatrix}$$

3. **f (n) = f (n-1) + f (n-3)**

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f(n) \\ f(n-1) \\ c \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \\ c \end{bmatrix}$$

4. **f (n) = a * f (n-1) + b * f (n-2) + c * f (n-3) + d * f (n-4) + e**

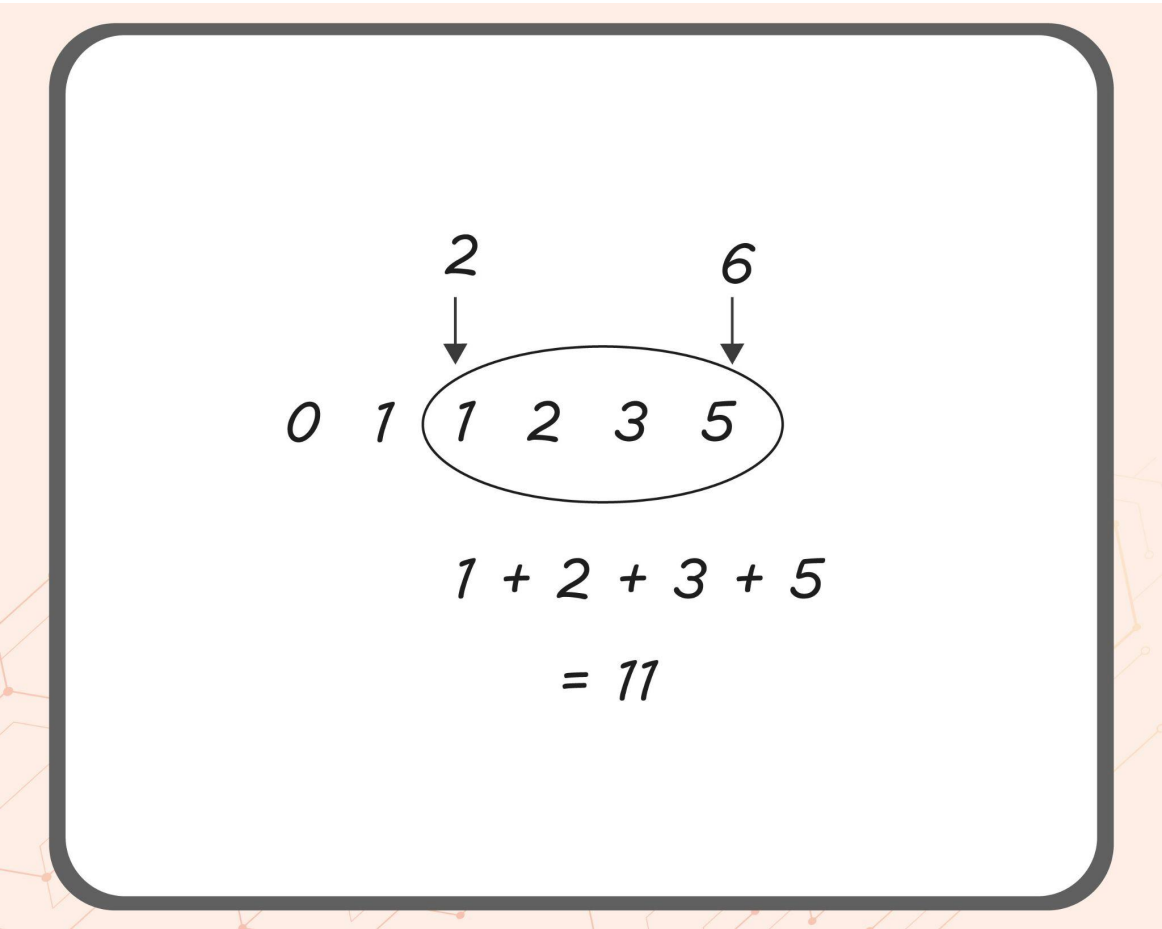
$$\begin{bmatrix} a & 0 & c & d & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f(n) \\ f(n-1) \\ f(n-2) \\ f(n-3) \\ e \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ f(n-2) \\ e \end{bmatrix}$$

FiboSum

Problem Statement : Find the sum between N^{th} and M^{th} fibonacci numbers.

Explanation :

Example : Taking N = 2 and M = 6,



Now, **M, N <= 10^9** and a maximum of 10^8 operations per second are allowed so we can't calculate the sum in **O (M)** or **O (N)**.

Optimized Approach: Sum of N fibonacci numbers is $\mathbf{S_N = S_{N-1} + F_N}$

$$F_n = F_{n-1} + F_{n-2}$$

$$F_{n-1} = F_{n-2} + F_{n-3}$$

$$F_{n-2} = F_{n-3} + F_{n-4}$$

$$\vdots$$

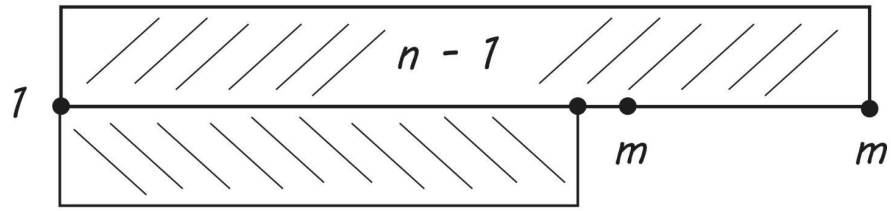
$$F_1 = 1$$

→ $F_N = F_{N-2} + F_{N-3} + \ldots\ldots + F_1 + 1$

→ $F_{N-1} = F_{N-2} + F_{N-3} + \ldots\ldots + F_1 = S_{N-2}$

Therefore, $S_N = F_{N+2} - 1$

$$S(m) - S(n-1) \qquad F(m+2) - 1$$



$$F(n = 1) - 1$$

$$\therefore (F(m+2) - 1) - F(n+1) - 1)$$

$$= F(m+2) - F(n+1)$$

Fermat's Little Theorem

Fermat's little theorem is a fundamental theorem in elementary number theory, which helps compute powers of integers modulo prime numbers.

The theorem states that :

Let **p** be a prime number, and **a** be any integer. Then $\mathbf{a^p - a}$ is always divisible by **p**.

In modular arithmetic notation, this can be written as $\mathbf{a^p \equiv a \bmod p}$.

Example: 11 is prime so $2^{11} - 2 = 2046$ is divisible by 11 by Fermat's little theorem.

$$a^{p} = a \pmod{p}$$

$$\rightarrow a^{p-1} = 1 \pmod{p}$$

$$\rightarrow (a^{p-1}) = (\bmod\ p) = 1$$

$$\rightarrow a^{-1}\,(a^{p-1}) = (\bmod\ p) = a^{-1}$$

$$\rightarrow (a^{-1}\,(a^{p-1})\ (\bmod\ p)\ \bmod\ p = (a^{-1})\ \bmod\ p$$

$$\rightarrow ((a^{-1}) \times (a^{p-1}))\ \bmod\ p = (a^{-1})\ \bmod\ p$$

$$\rightarrow \boxed{(a^{p-1})\ \bmod\ p}$$

$\mathbf{(a^{p-2})}$ mod p can be calculated using modular exponentiation.

Wilson's Theorem

Wilson's theorem states that a positive integer **p > 1** is a prime if and only if **(p−1)!≡−1(modp)**.

Let's verify Wilson's theorem using some examples:

n	(n-1)!	(n-1)! (mod n)	Is Prime?
2	1	1	yes
3	2	2	yes
4	6	2	no
5	24	4	yes
6	120	0	no
7	720	6	yes
8	5040	0	no
9	40320	0	no
10	362880	0	no

Uses :

Consider the problem of computing factorial under modulo of a prime number which is close to the input number, i.e., we want to find the value of "n! % p" such that n < p, p is a prime and n is close to p. For example (25! % 29). From Wilson's theorem, we know that 28! is -1. So we basically need to find [(-1) * inverse(28, 29) * inverse(27, 29) * inverse(26)] % 29. The inverse function inverse(x, p) returns the inverse of x under modulo p.

Income on the Nth day

Problem Statement:

Daulat Ram is an affluent businessman. After demonetization, an IT raid was held at his accommodation in which all his money was seized. He is very eager to gain his money back, he started investing in certain ventures and earned out of them. On the first day, his income was Rs. X, followed by Rs. Y on the second day. Daulat Ram observed his growth as a function and wanted to calculate his income on the nth day.

The function he found out was **F(n) = F(n-1) + F(n-2) + F(n-1)×F(n-2)**

Given his income on day 0 and day 1, calculate his income on the nth day (yeah It's that simple).

Explanation:

Manipulating the given expression a little bit:

$$F(n) = F(n-1) + F(n-2) + F(n-1) \times F(n-2)$$

$$F(n) = F(n-1)\ (1 + F(n-2) + F(n-2)$$

$$F(n) = F(n-1)\ (1 + F(n-2) + 1 + F(n-2) - 1)$$

$$F(n) = (1 + F(n-2))\ (1 + F(n-1)) - 1$$

$$= 1 + F(n) = (1 + F(n-1))\ (1 + F(n-2))$$

Take $G(n) = 1 + F(n)$

$$= \boxed{G(n) = G(n-1) * G(n-2)}$$

Now take **G(0) = a** and **G(1) = b**.

→ **G(2) = ab**

→ **G(3) = ab^2**

→ **G(4) = a^2b^3**

→ **G(5) = a^3b^5**

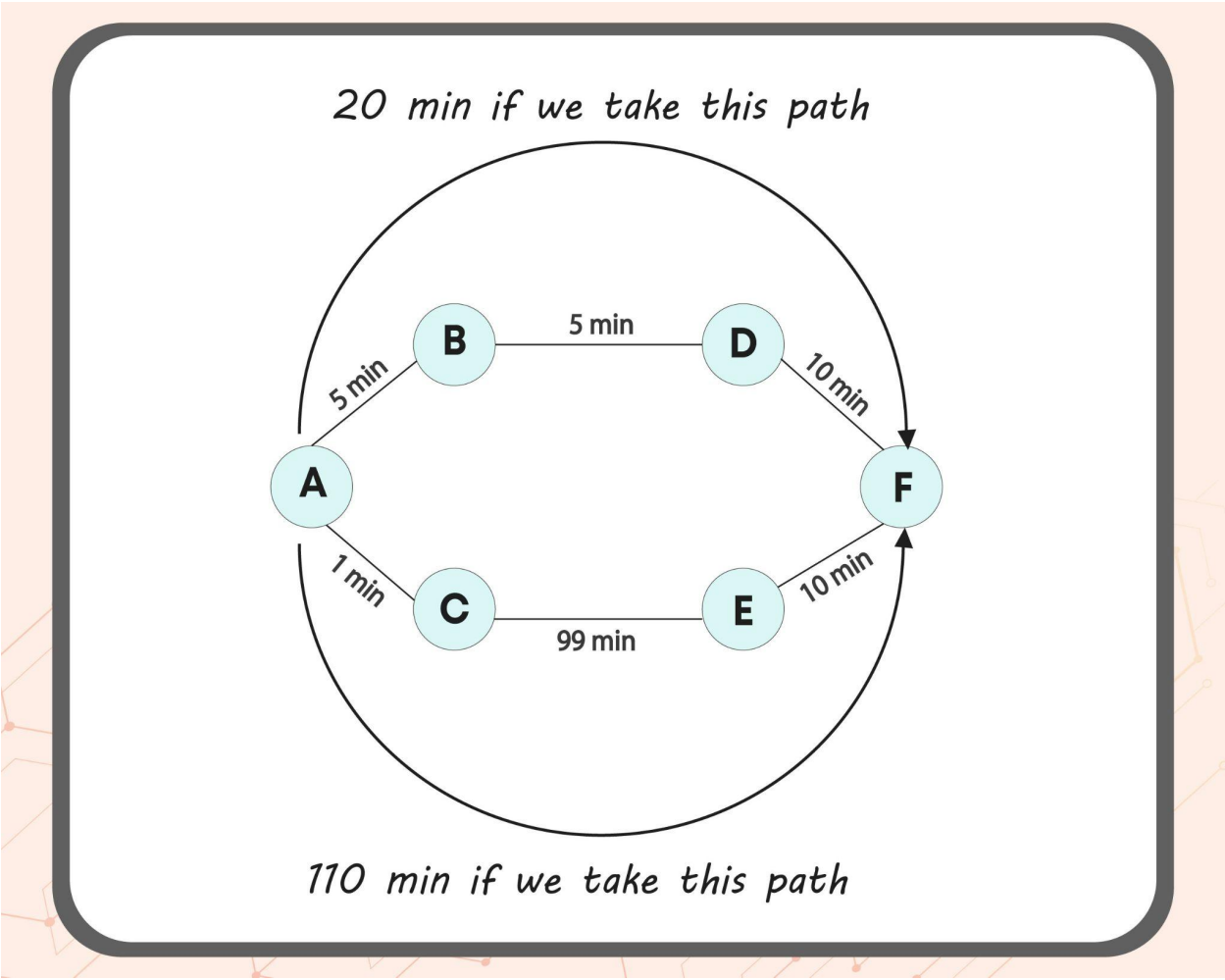
→ **G(n) = $a^{fib(n-1)}\ b^{fib(n)}$**

Now that we know the general expression of G(n), we can easily find out the general expression for F(n).

Greedy Problems

Introduction

Suppose we have 6 places A, B, C, D, E, F and we are given the time to reach from one destination to another, and we have to tell the shortest time required to reach from A to F.



Now, if we take the path A → B → D → F that will be our DP solution since we know that it is the overall shorter path, but if we take path A → C → E → F, then it will be greedy approach since we only know and judge our decision on the basis of path A → B and path A → C.

Greedy solution is not giving us the right answer in this problem but it might be a useful approach in others.

Activity Selection

Problem Statement: We are given the starting time and the finishing time of activities and we have to tell how many activities can be completed in a given time interval. We can not do more than one activity simultaneously.

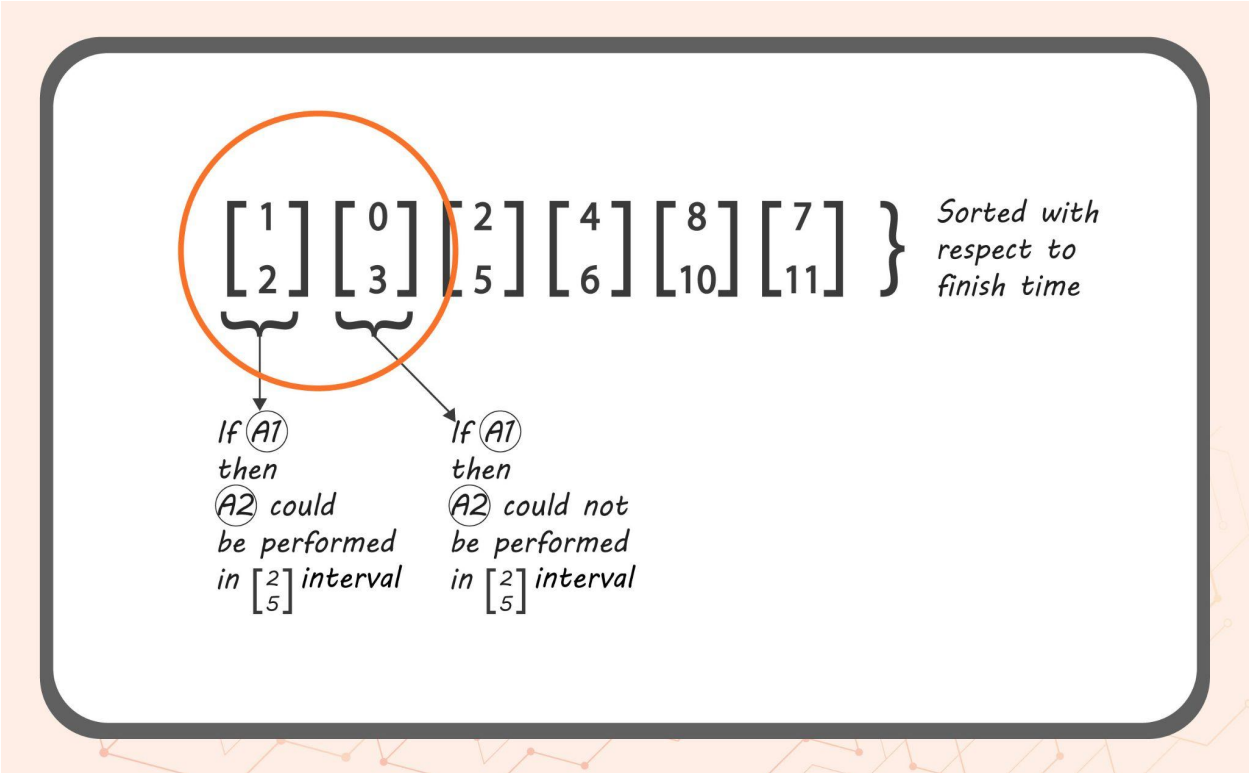
Example:

Given time interval to perform the activities: **0 to 11**.

Output: **3** (because the first activity will be performed in the time interval **1 to 2**, the second activity will be performed in the time interval **2 to 5** and the third activity will be performed in interval **8 to 10**.

	A1	A2	A3	A4	A5	A6
1 Start	[0	1	2	4	7	8]
A finish	[3	2	5	6	11	10]

The greedy approach to solving this problem starts by thinking to finish an activity as soon as possible and start with a new one. So we sort these two arrays on the basis of their finishing time or rather than saying that we sort the two arrays, we can say that we make a structure that has both the start time and finish time and we sort that on the basis of the finish time.



Now, A1 could be finished early if we do it in [1, 2] time interval which leaves us the option to start A2 in [2, 5] time interval.

So we say that greedy technique is applied when we start from an optimal answer from the first step and continue to look for the optimal answers in the next steps also.

The code to this solution is left for the reader as an exercise.

Minimum Absolute Difference in Array

Problem Statement: We are given an array and we have to tell the minimum absolute difference between two elements.

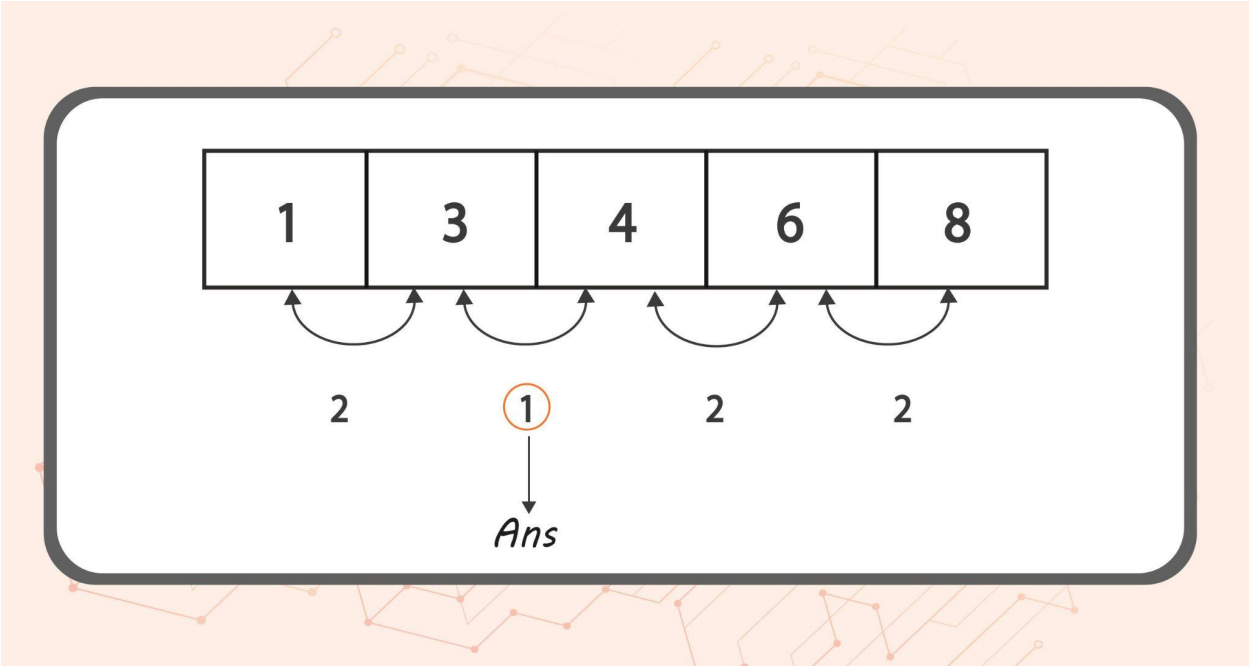
Explanation:

For example, **Arr = [1, 3, 2, 5, 4, 11, 7]**. Now, the difference between 1 and 3 is 2 and the difference between 3 and 2 is 1. So in this example the minimum absolute difference is 1.

Another example is, **Arr = [1, 4, 6, 3, 8]** then also the answer will be 1 i.e., the difference between 3 and 4.

Basic Approach: The basic approach will take $O(N^2)$ time as we will have to generate all the differences of elements by traversing using two loops.

Greedy Approach: If we sort this array then we can think of the greedy technique. After sorting, the array becomes **Arr = [1, 3, 4, 6, 8]**. Now if we start traversing the elements we need not use two loops because we just have to check the minimum difference in the consecutive elements only.



The code to this solution is left for the reader as an exercise.

Fractional Knapsack

Problem Statement: This problem is just like the original knapsack problem in which we were given two arrays: one with weights of the items and the other with the values of these items and we were given a maximum weight that we can put in the knapsack. Now in this we can take the fractional weight of an item by adding only a fraction of the item.

Explanation:

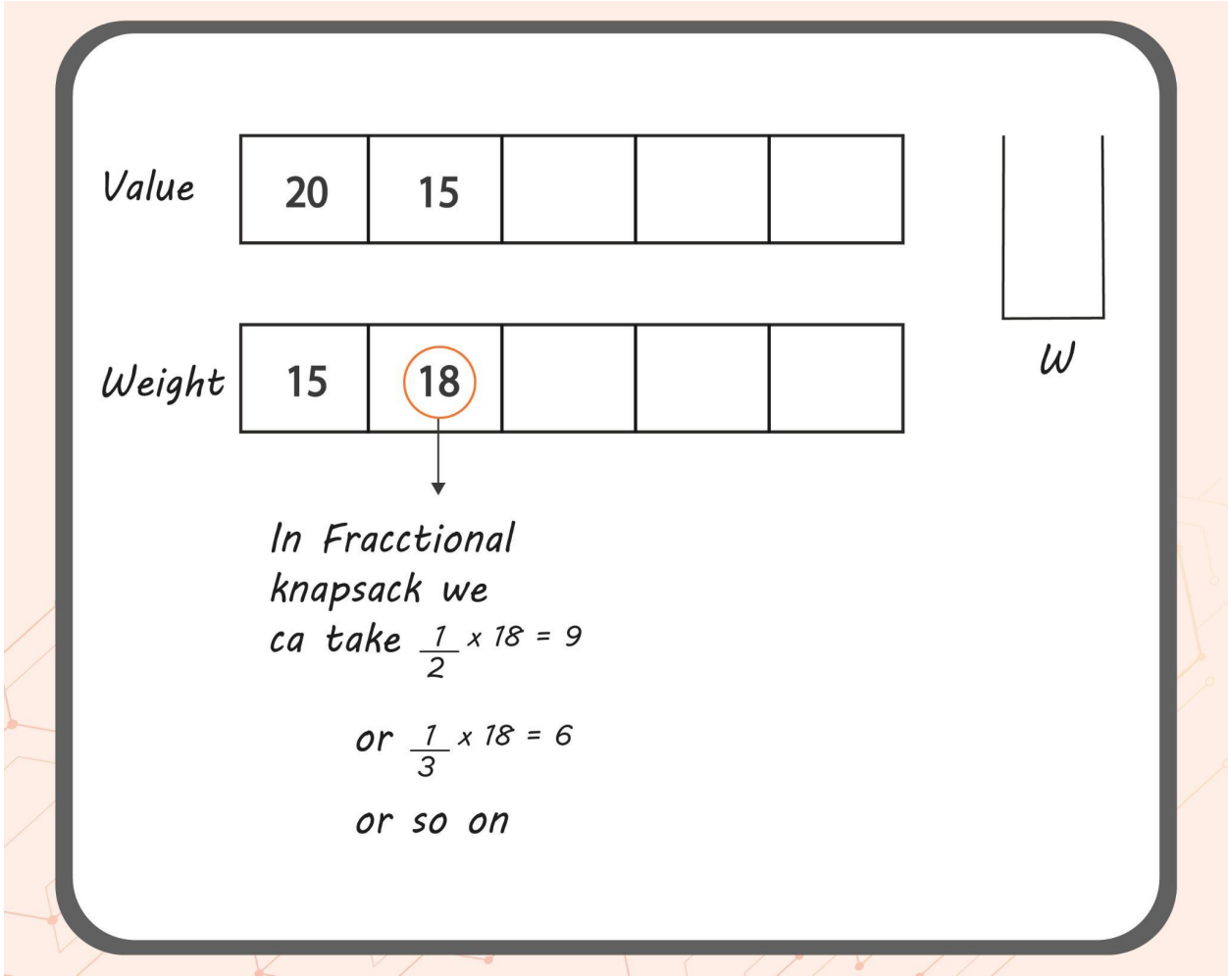
Example 1:

Let us consider the different items that need to be put in the knapsack as dry fruits like Cashews, Raisins, Almonds in quantities 6 kg, 3 kg, and 4 kg respectively and we have a knapsack with weight capacity as 5 kg. The relative prices of these are in the order Cashew > Raisins > Almonds. Now, as we can include fractional weight of items, hence, we will obviously want to fill our knapsack with the items with maximum relative price i.e. Cashews.

Now, let's think for a second that what if these items were not dry fruits and were statues, then we would not have the option to take a part of the total weight of the item as breaking a statue into parts will result in the loss of its value and we could not have chosen the 6 kg statue with the maximum value because our knapsack's capacity is only 5 kg. Therefore, if we have statues as items, then it becomes a case of 0-1 Knapsack.

But since dry fruits can be easily divided and measured according to the weight, that means we now have the option to take only 5 kg out of 6 kg Cashews and still maximize the value of items in the knapsack.

Let's take a look at the image below with another example to understand fractional knapsack.



So, in this problem, we will have to make an array that contains the value per weight of the items and sort that in descending order and then start filling the knapsack accordingly. Therefore we are applying a greedy approach by choosing the items that have the maximum value per weight.

Let's look at the dry run of another example:

Weight of Knapsack = 60 kg

ITEM	WEIGHT	VALUE	VALUE / WEIGHT
I1	5	30	6
I2	10	40	4

I3	15	45	3
I4	22	77	3.5
I5	25	90	3.6

Now we sort all the items in decreasing order of their value per weight.

I1	I2	I5	I4	I3

We are ready to start filling the knapsack

Remaining Knapsack Weight	**Items in Knapsack**	**Cost**
60	NIL	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now, the remaining knapsack weight is 20kg but I4 weighs 22kg. Therefore we add only 20 kg of I4 which makes the total cost of our knapsack to be 160 + (20/22)*77 = 230.

Segment Tree

Introduction

Suppose we have a problem in which two operations are performed:

1. **Range Query** (**finding sum of elements in a range**).
2. **Update Array element(s)**

For example, in array = [1, 3, 5, 7, 6, 8] we have to perform the following operations:

Operation	Brute Force Time Complexity of Operation
1. Sum (array[0] to array[4])	O(N)
2. array[2] = 6	O(1)
3. array[3] = 9	O(1)
4. Sum (array[1] to array[4])	O(N)

Now, if we have **K** number of queries and **L** number of update operations then the overall time complexity of the solution becomes **O(K*N) + O(L*1)**. This increases with respect to **K**. Therefore, this approach is only effective when **K** is small.

Another approach to solve this is by maintaining another array in which the ith element will store the sum from 0th to the ith element in our original array.

For example, ArrSum[2] = Sum(array[0] to array[2]).

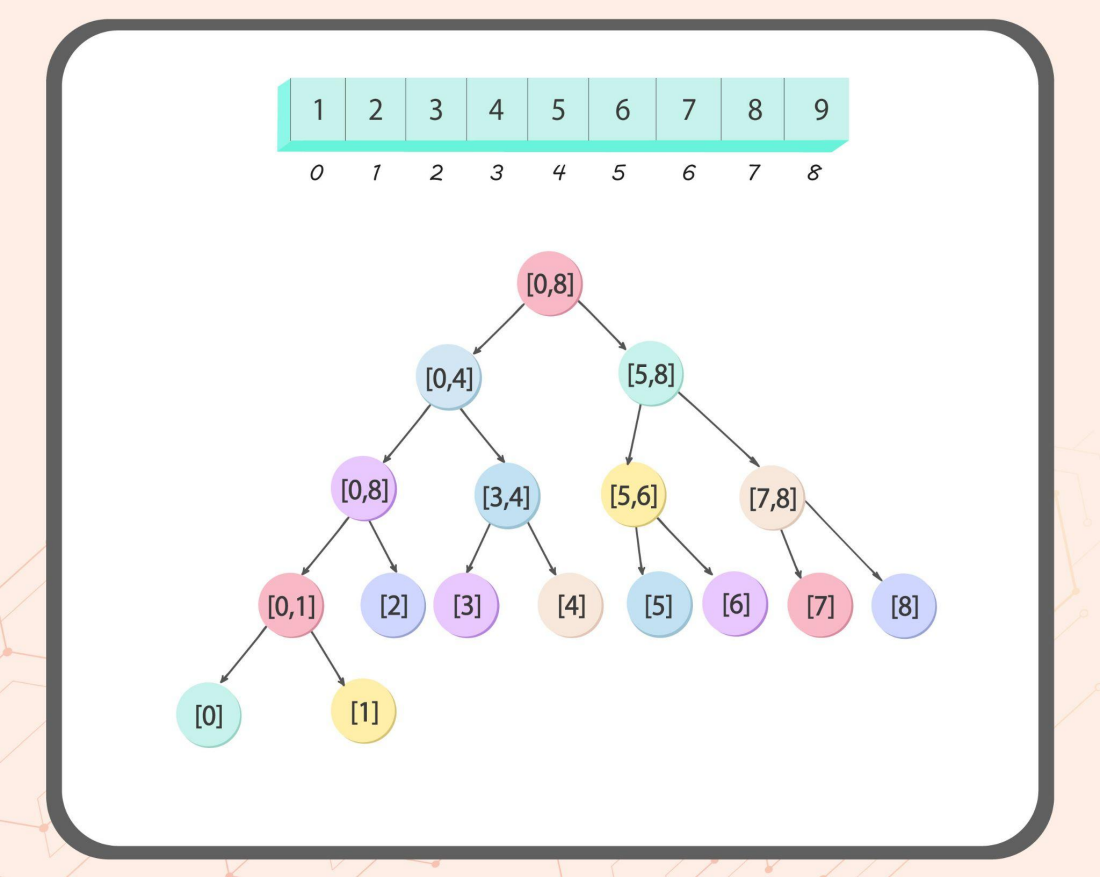
Array = [1, 3, 5, 7, 6, 8]
ArrSum = [1, 4, 9, 16, 22, 30]

Sum(array[2] to array[4]) = Sum(array[0] to array[4]) – Sum(array[0] to array[1])

= ArrSum[4] – ArrSum[1]

Therefore, the time complexity of solving the query reduces to **O(1)** but for update operations, in the worst case we have to traverse the entire "**ArrSum**". Hence, the time complexity of update operation increases to **O(N)**.

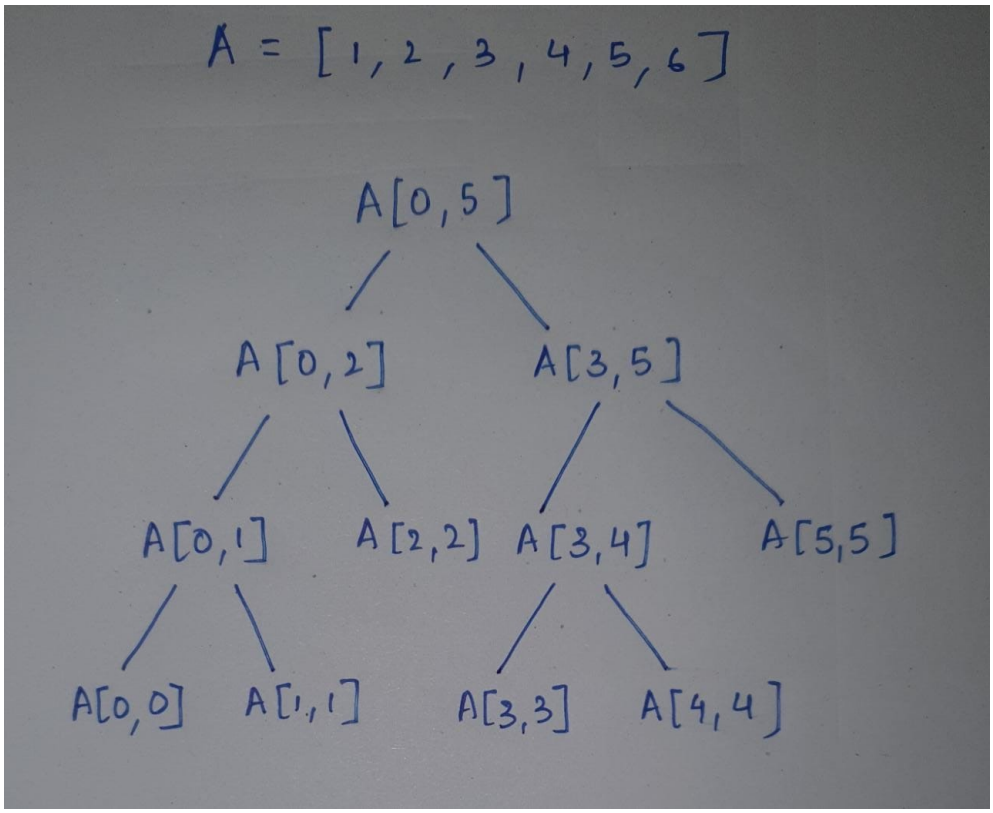
Segment Tree helps us to perform both Query and Update Operations in **O(log(N))** time.



In the Segment Tree, most of the solutions to queries are already present at its nodes, for those queries whose solutions are not directly present like Sum(array[5] to array[7]), we can calculate it by using the [5, 8] and [8] nodes.

Even for the worst-case scenario, that is, traversing through the entire tree, the time complexity of the array will be equal to **O(log(N))** which is equal to the height of the tree.

Building the Segment Tree



Building a Segment tree requires **O(N)** time because there are **N** leaf nodes in a segment tree and the remaining nodes are equal to **N – 1**. Therefore, the total number of nodes = **(N) + (N – 1)** = **2N – 1**.

```
int main () {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    // segment tree is given the size of 2*size of arr
```

```
    int *tree = new int[18] ;

    buildTree(arr, tree, 0, 8, 1);
}
```

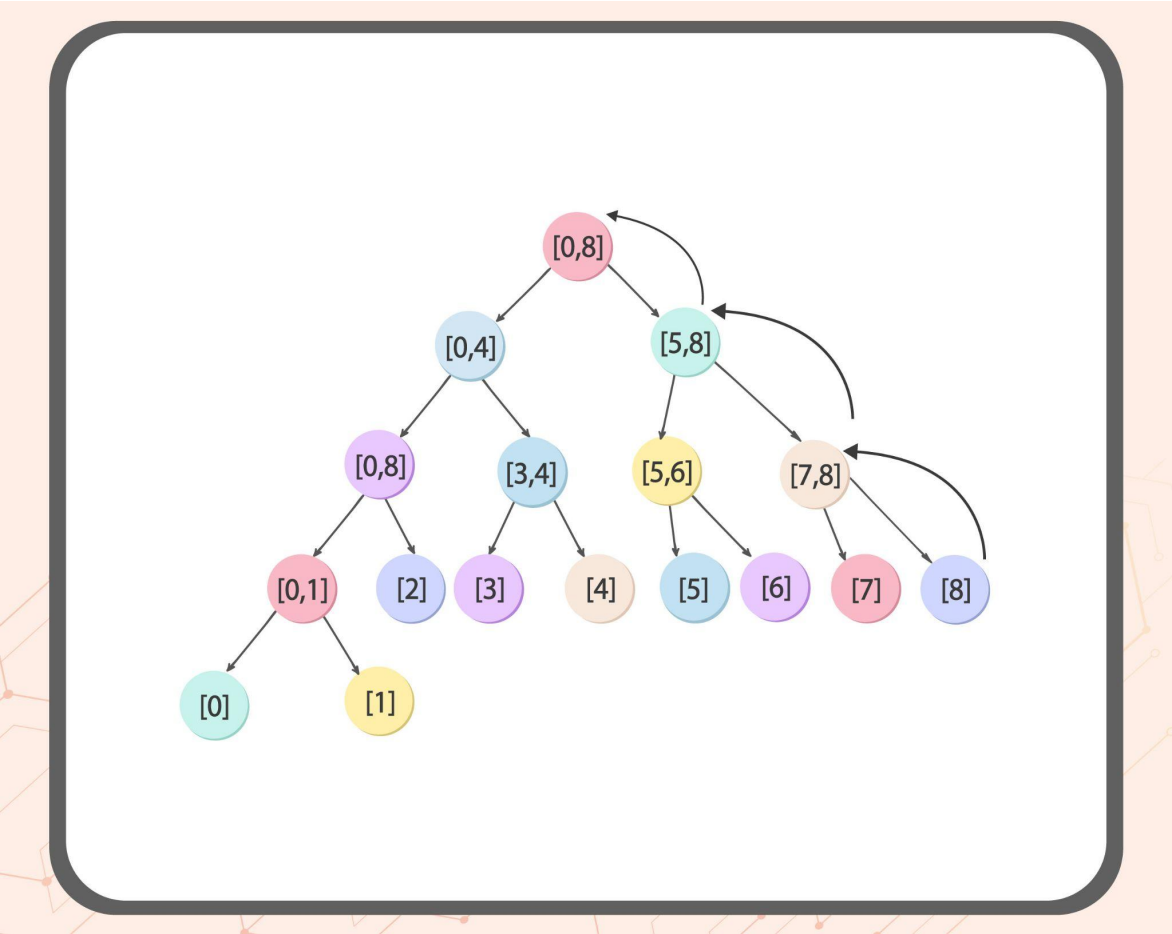
```
void buildTree(int* arr, int* tree, int start, int end, int treeNode){

    // BASE CASE
    if(start == end){
        tree[treeNode] = arr[start];
    }

    int mid =(start+end)/2;
    // two recursive calls to build the left half and
    // right half respectively
    buildTree(arr, tree, start, mid, 2*treeNode);
    buildTree(arr, tree, mid+!, end, 2*treeNode+1);
    // the answer to the recursive calls gets stored in
    // the treeNode
    tree[treeNOde] = tree[2*treeNode] + tree[2*treeNode + 1]
}
```

The size of the segment tree is twice the size of our initial array and the indexing starts from 1 instead of 0.

Here there are two recursive calls happening for every ***treeNode*** and the sum of their answers gets stored inside the ***treeNode***.

Updating the Segment Tree

First we find mid, then decide either to go left or right, that is, if the index where we have to update the value is greater than mid then we go right, else we go left.
Even if the values at other nodes get affected by updating a single node, we only travel the whole height of the tree in the worst case scenario, hence, the time complexity for update operation is **O(log(N))**.

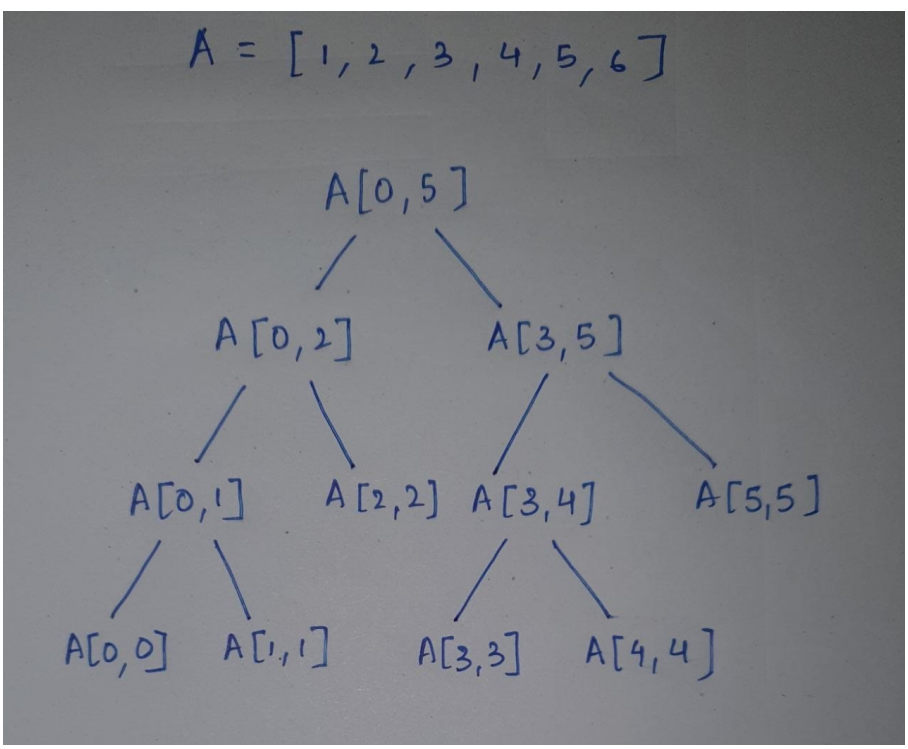


Example: For the update function at [8] or 9th index of the segment tree, only the values at 4 nodes need to be changed.

```
void updateTree(int* arr, int* tree, int start, int end, int treeNode, int idx
, int value){

    int mid = (start+end)/2;
    if(idx > mid){
        updateTree(arr, tree, mid+1, end, 2*treeNode+1, idx, value);
    } else {
        updateTree(arr, tree, start, mid, 2*treeNode, idx, value);
    }
    tree[treeNode] = tree[2*treeNode] + tree[2*treeNode+1];
}
```

Query on the Segment Tree

Example : A = [1, 2, 3, 4, 5, 6] **Query: Sum(2 to 4)**



- At A[0, 5] : the value of this node doesn't give us the answer to our query. Hence, we have to call recursion, now our query requires a sum

from 2 to 4th index so it is partially inside A[0, 2] and A[3, 5]. Therefore, a recursion call is made to both the left and right subtree.

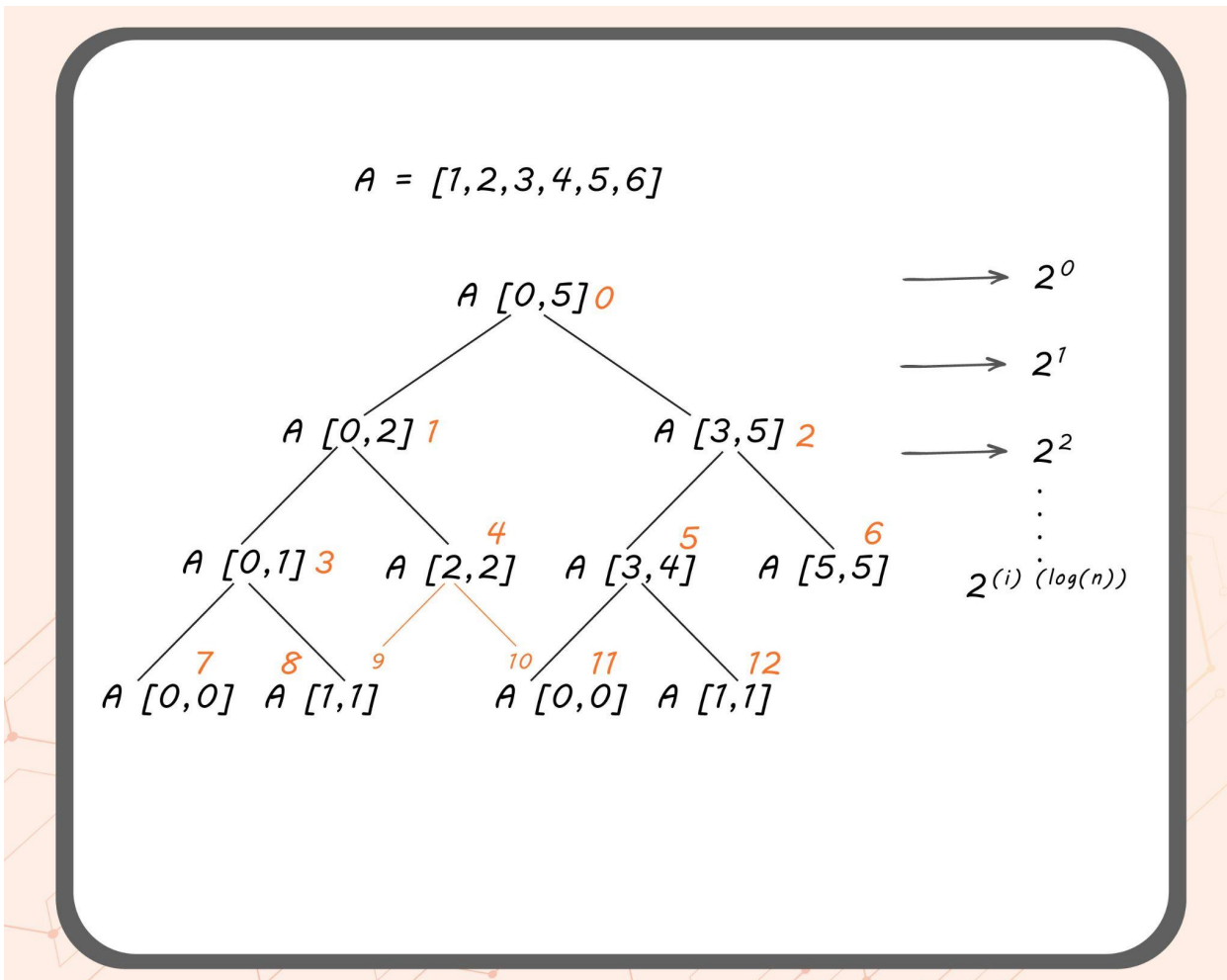
- At A[0, 2] : recursion call to A[0, 1] is not made because 0 and 1 are completely outside the range of our query and the answer of A[2, 2] which is completely inside the range of our query is returned using a recursive call.

- At A[3, 5] : recursion call to A[5, 5] is not made because 5 lies completely outside the range of our query while the answer to A[3, 4] which is completely inside the range of our query is returned using a recursive call.

```
int query(int *tree, int start, int end, int treeNode, int left, int right){
    // completely outside the given range
    if(start > right || end < left){
        return 0;
    }
    // completely inside the given range
    if(start >= left && end <= right){
        return tree[treeNode];
    }
    // Partially inside and partially outside
    int mid = (start+end)/2;
    int ans1 = query(tree, start, mid, 2*treeNode, left, right);
    int ans2 = query(tree, mid+1, end, 2*treeNode+1, left, right);

    return ans1 + ans2;
}
```

Size of the Segment Tree

Earlier we made the segment tree to be of size **2N,** because we were storing **2N** nodes. This doesn't always work because like in the figure shown below A[3, 3] and A[4, 4] are supposed to be stored at the 11th and 12th index rather than the 9th and 10th index.



- Total number of nodes = $2^0 + 2^1 + 2^2 + ... + 2^{\text{ceil (log(N))a}}$

$$= (1)(2^{\text{ceil (log(N)) +1}} -1) /(2-1)$$

$= 2*2^{\text{ceil}(\log(N))}$

= **4N** (approx)

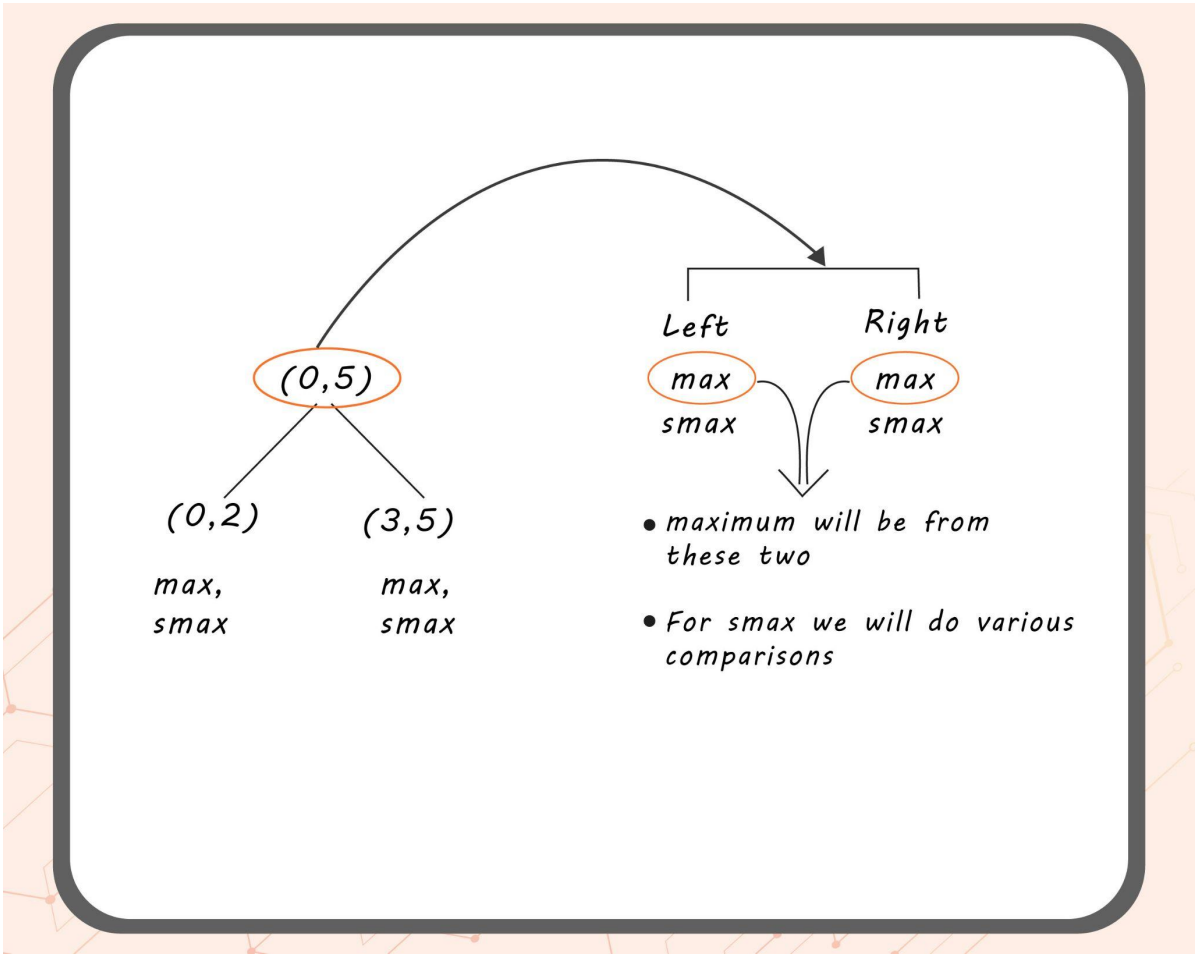
Maximum Pair Sum

Problem Statement: Suppose we are given an array = [2, 3, 1, 5, 7, 6] and we have to find a pair that gives us the maximum sum.

Explanation:

Basic Approach: If we need a pair that has the maximum sum then that clearly means the elements in the pair are maximum and second maximum in the array.
Therefore, we can sort this array or use a two pointer approach to solve this question.

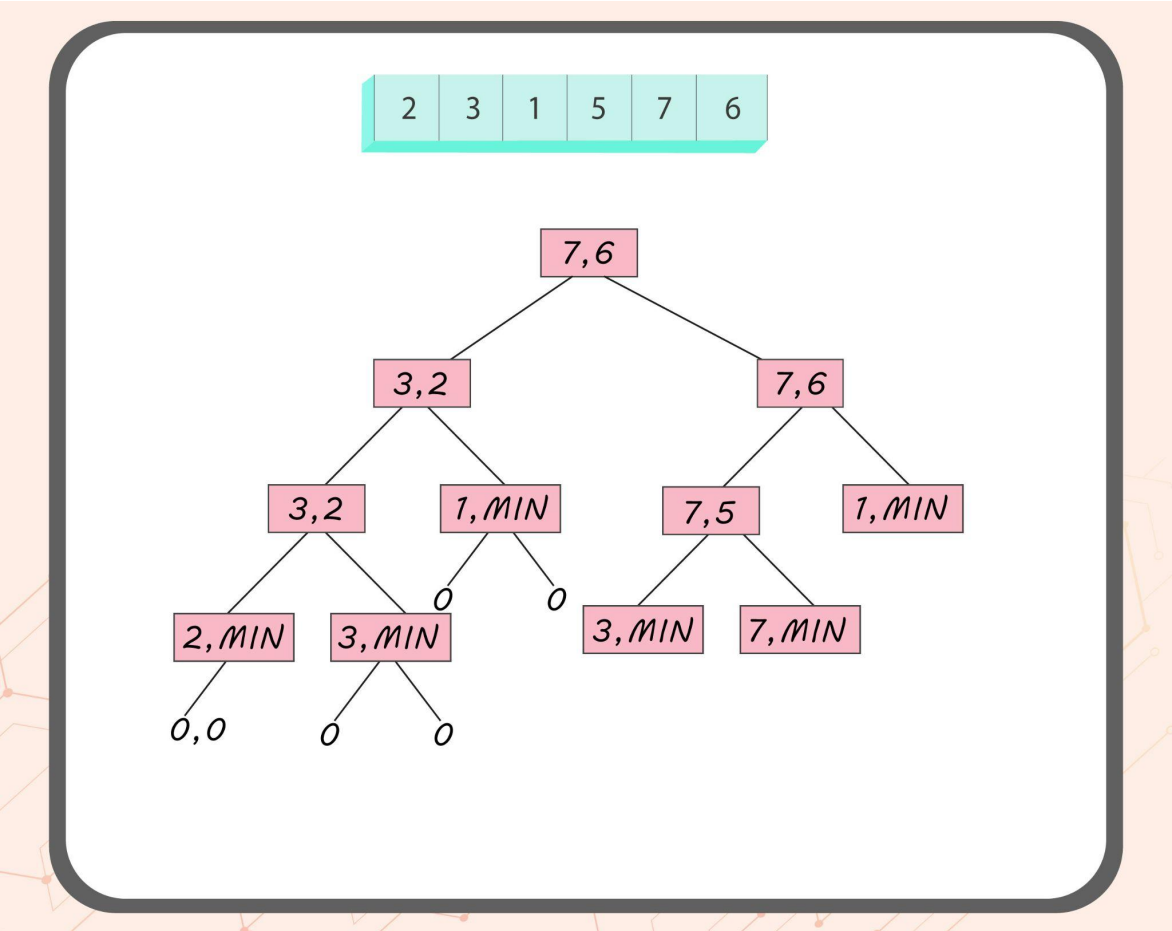
Segment Tree Approach: We can use the segment tree in the above problem by calculating the maximum(**max**) and second maximum(**smax**) at each node of the segment tree.



For the node that stores the answer of (0, 5), we have one left node and one right node and we have the max and smax from both left and right nodes.

Now, the max of their parent node will be from the max of the left or max of the right node.
If the right node's max is chosen then smax of the parent node will be the max of the left node or smax of the right node.
And vice versa.

The tree after the build will look like this:



The code is pretty similar to what we have done until now so try to do it on your own.

Lazy Propagation

Earlier we used to update the value on a single index but we use lazy propagation when we have to update in a range of indexes.

When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required.

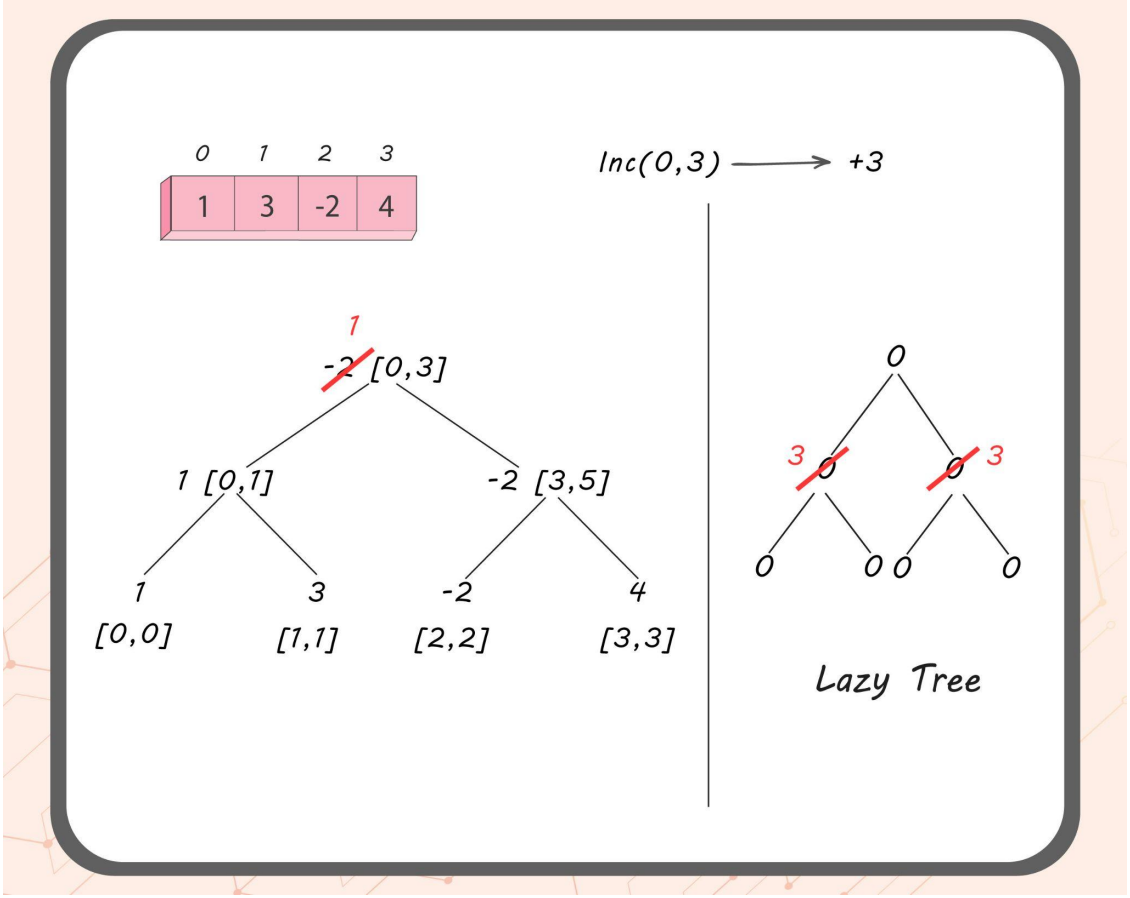
We make a lazy tree with the same structure as our segment tree. A value of 0 at any node represents that there are no pending updates on that node. A non-zero value represents the amount to be added to the node in the segment tree before making any query to the node.

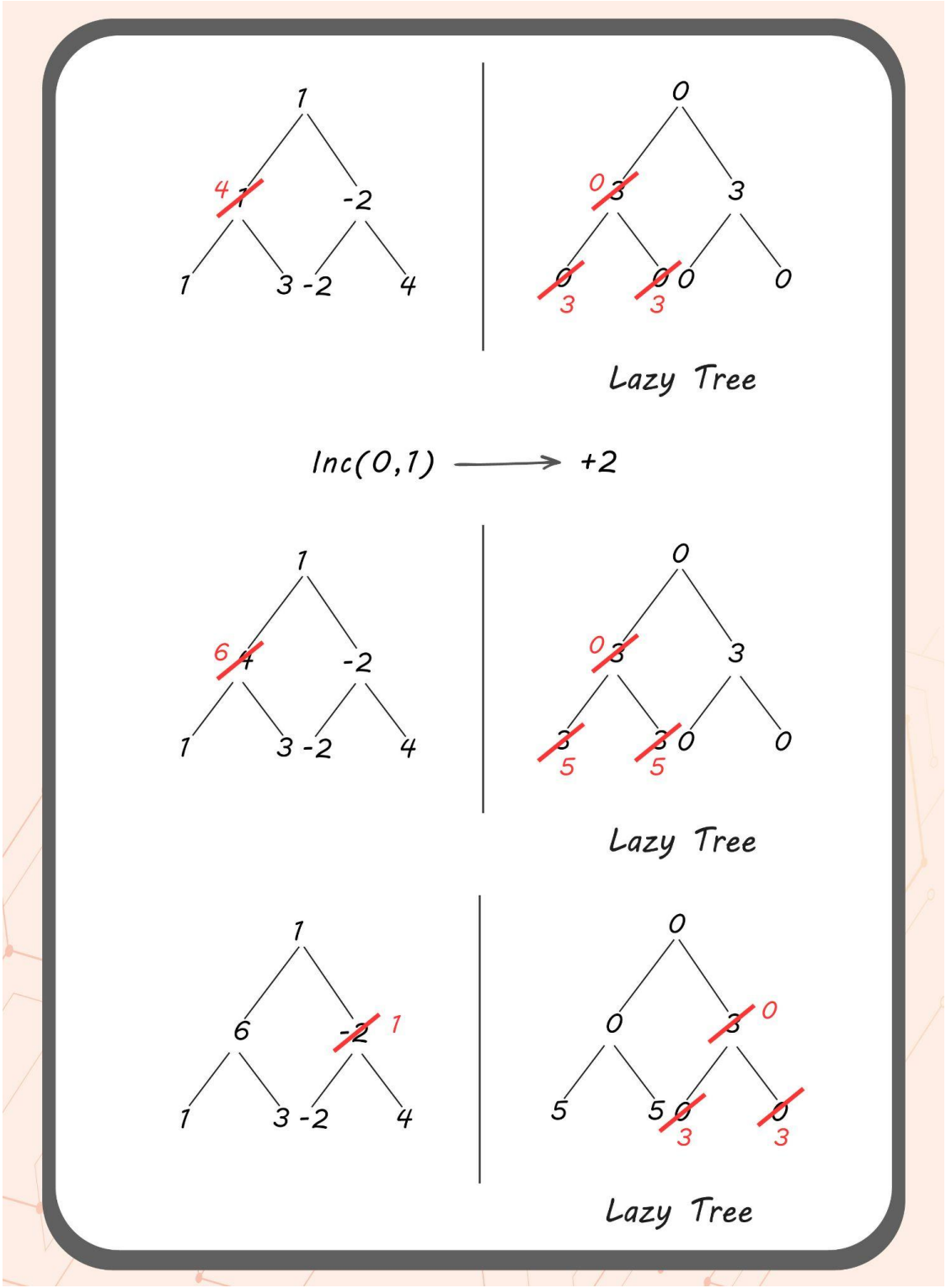
Algorithm:

// To update segment tree for change in array
// values at array indexes from l to r.

updateRange(l, r)

1. If the current segment tree node has any pending update, then first add that pending update to the current node.
2. If the current node's range lies completely in the update query range:
    a. Update current node
    b. Postpone updates to children by setting lazy values for children nodes.
3. If the current node's range overlaps with the update range, follow the same approach as above.
    a. Recur for left and right children.
    b. Update current node using results of left and right calls.


0 1 2 3
1 3 -2 4
Inc(0,3) → +3
1
-2 [0,3]
1 [0,1]
-2 [3,5]
1
[0,0]
3
[1,1]
-2
[2,2]
4
[3,3]
0
3 0
0 3
0 0 0 0
Lazy Tree


Lazy Tree
Inc(0,1) → +2
Lazy Tree
Lazy Tree

Code :

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

void buildTree(int* arr,int* tree,int start,int end,int treeNode){

    if(start == end){
        tree[treeNode] = arr[start];
        return;
    }
    int mid = (start+end)/2;

    buildTree(arr,tree,start,mid,2*treeNode);
    buildTree(arr,tree,mid+1,end,2*treeNode+1);
    tree[treeNode] = min(tree[2*treeNode],tree[2*treeNode+1]);
}

void updateSegmentTreeLazy(int* tree,int* lazy,int low,int high,int
startR,int endR,int currPos,int inc){

    if(low > high){
        return;
    }

    if(lazy[currPos] !=0){
        tree[currPos] += lazy[currPos];

        if(low!=high){
            lazy[2*currPos] += lazy[currPos];
            lazy[2*currPos+1] += lazy[currPos];
        }
        lazy[currPos] = 0;
    }

    // No overlap
    if(startR > high || endR < low){
        return;
    }

    // Complete Overlap
```

```
    if(startR<= low && high <= endR){
        tree[currPos] += inc;
        if(low!=high){
            lazy[2*currPos] += inc;
            lazy[2*currPos+1] += inc;
        }
        return;
    }

    // Partial Overlap

    int mid = (low+high)/2;
    updateSegmentTreeLazy(tree,lazy,low,mid,startR,endR,2*currPos,inc);

updateSegmentTreeLazy(tree,lazy,mid+1,high,startR,endR,2*currPos+1,inc);
    tree[currPos] = min(tree[2*currPos],tree[2*currPos+1]);
}

int main(){
    int arr[] = {1,3,-2,4};
    int* tree = new int[12]();
    buildTree(arr,tree,0,3,1);
    int* lazy = new int[12]();
    updateSegmentTreeLazy(tree,lazy,0,3,0,3,1,3);
    updateSegmentTreeLazy(tree,lazy,0,3,0,1,1,2);

    cout<< "Segment Tree" <<endl;
    for(int i=1;i<12;i++){
        cout<<tree[i]<< endl;
    }

    cout<< "Lazy Tree" <<endl;
    for(int i=1;i<12;i++){
        cout<<lazy[i]<< endl;
    }
    return 0;
}
```

Fenwick Tree

Introduction

A Fenwick tree or binary indexed tree is a data structure used for **Range Query problems**.

Fenwick tree has the following benefits over segment trees:

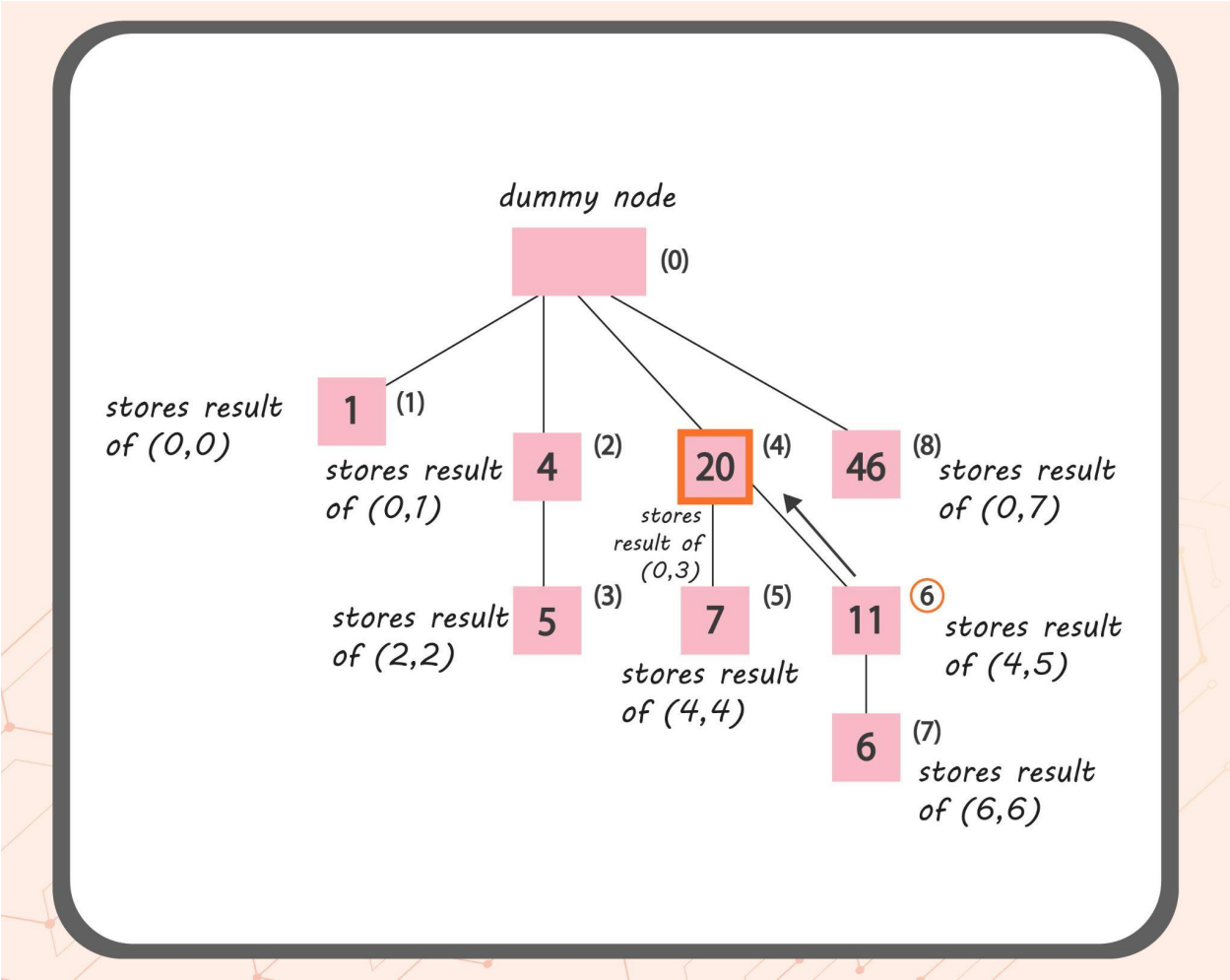
1. Code for the Fenwick tree is much simpler than that of the segment tree.
2. It takes very less space as compared to segment tree.

Let's see how a binary indexed tree is constructed.
Suppose we have an array, **Arr = [1, 3, 5, 11, 7, 4, 6, 9]** and we need to make a binary indexed tree from this.
First of all we should know that if an array has **N** elements, then the corresponding binary indexed tree will have **N + 1** nodes. The 0th node does not contain an answer to any query. In the above array there are 8 elements, so the corresponding binary indexed tree will have 9 nodes.

The first node is a dummy node which contains nothing. Thus, indexing actually starts from 1.

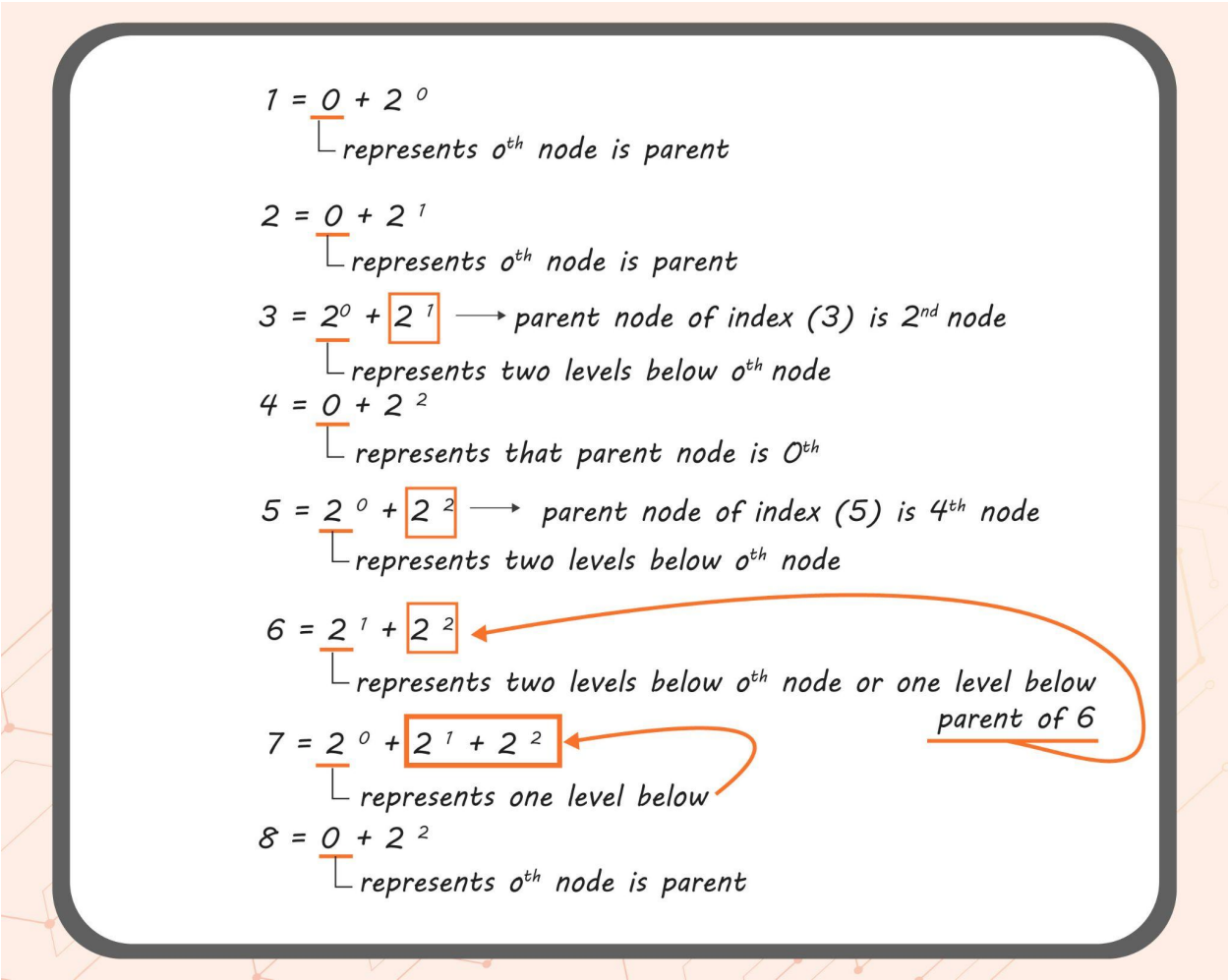
Let's see how a Fenwick Tree looks and how we query on it.

Example :
Suppose we have to find the result of the query (0, 5).
Since we have started our indexing from 1, we go to index 5 + 1 = 6 **($2^2 + 2^1$)**.
Thus, the 6th node stores the answer to the query (4, 5), and its parent node which is at index 4 stores the answer to the query (0, 3). Hence our final answer / result will be the sum of results at index 6 and its parent index 4.
So the final answer of the query (0, 5) = 11 + 20 = 31.

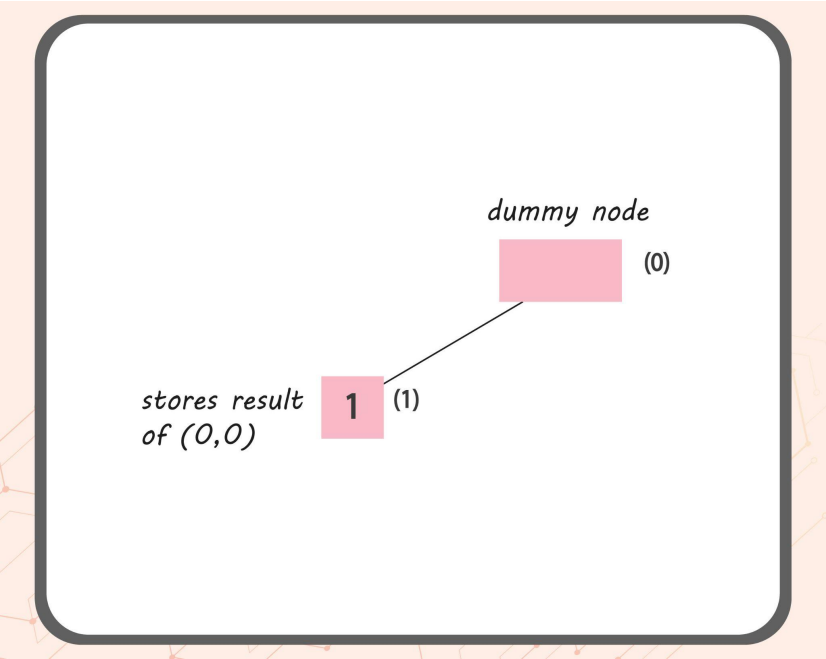

dummy node
(0)
stores result of (0,0)
1
(1)
stores result of (0,1)
4
(2)
20
(4)
46
(8)
stores result of (0,7)
stores result of (0,3)
stores result of (2,2)
5
(3)
7
(5)
11
6
stores result of (4,5)
stores result of (4,4)
6
(7)
stores result of (6,6)

Now, let's take a look behind the concept of building a Fenwick Tree.

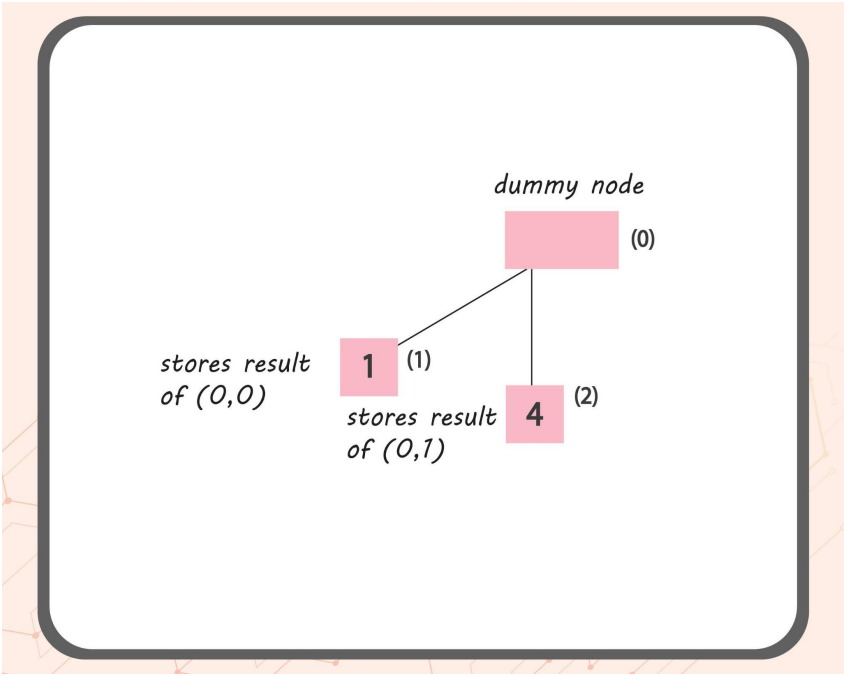
We can represent the indices in the form of powers of 2 such as -



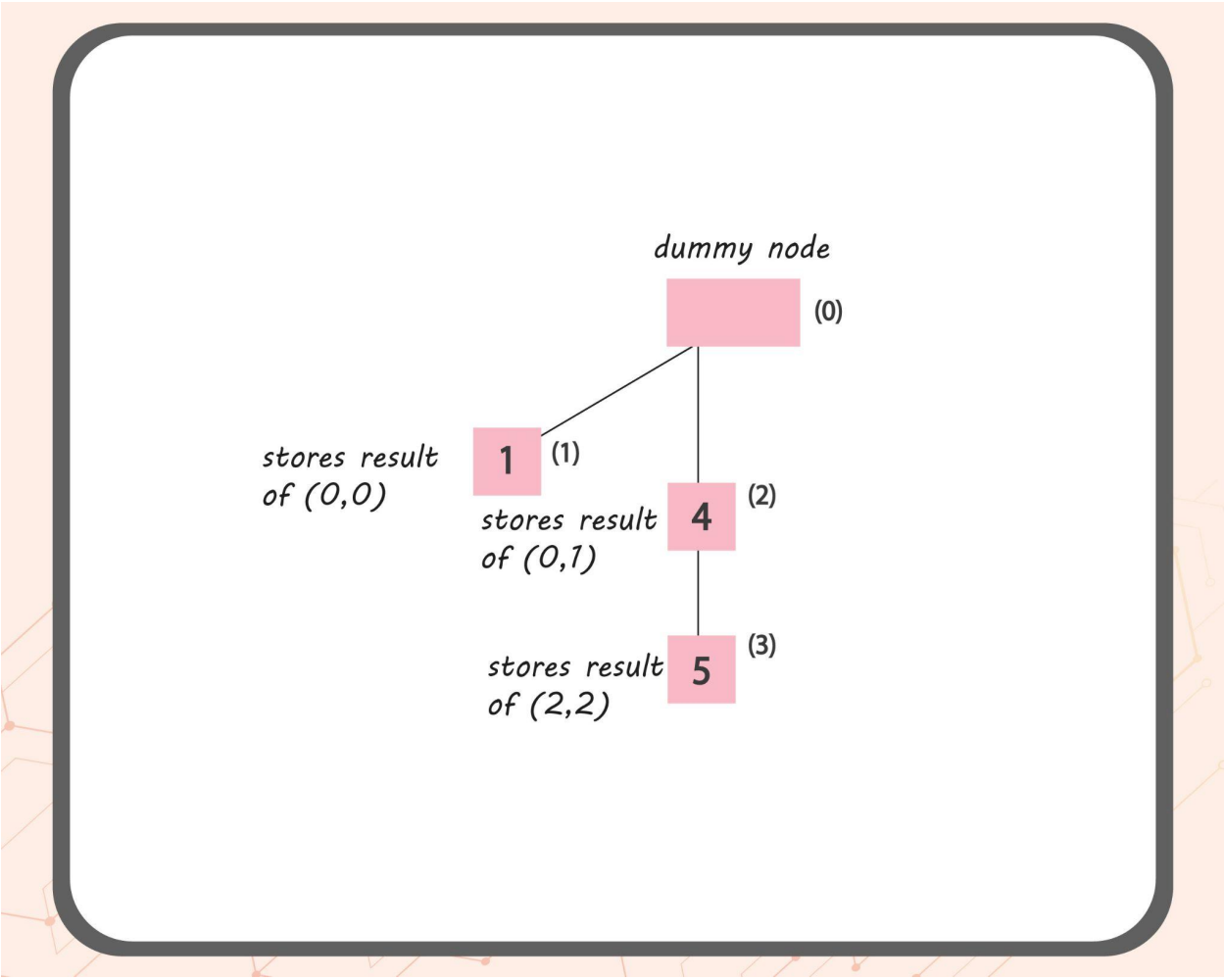
If we look at index **1 = 2^0 + 0,** the 0 represents the level or height at which the parent node of index 1 is. That is, the parent node of index 1 is the 0th node (which is empty). Index 1 stores the result of (0,0).



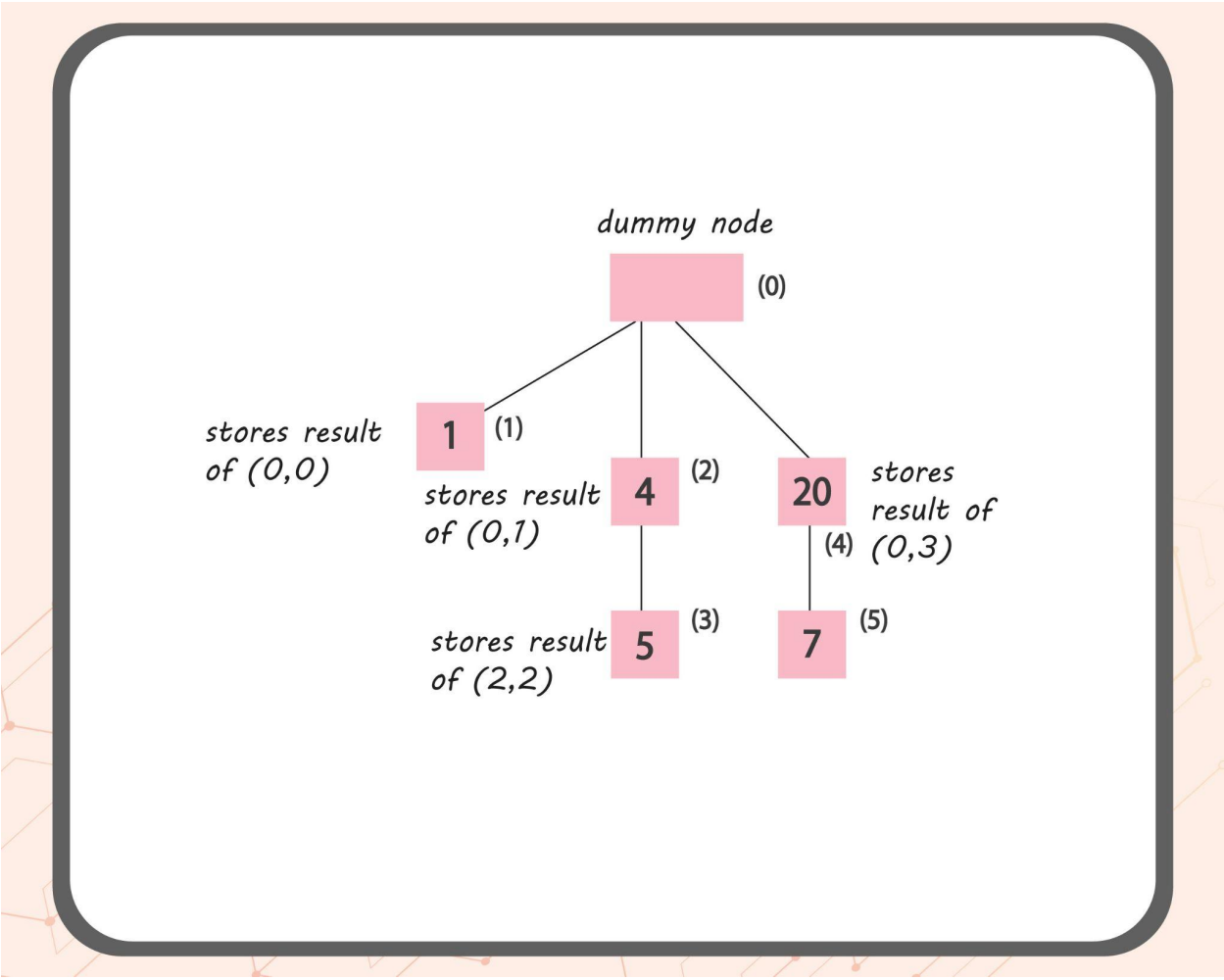
Now, index **2 = 2^1 + 0**. The 0 represents that it is the direct child of the 0th node and 2^1 represents that it will store the result of (0, 1).



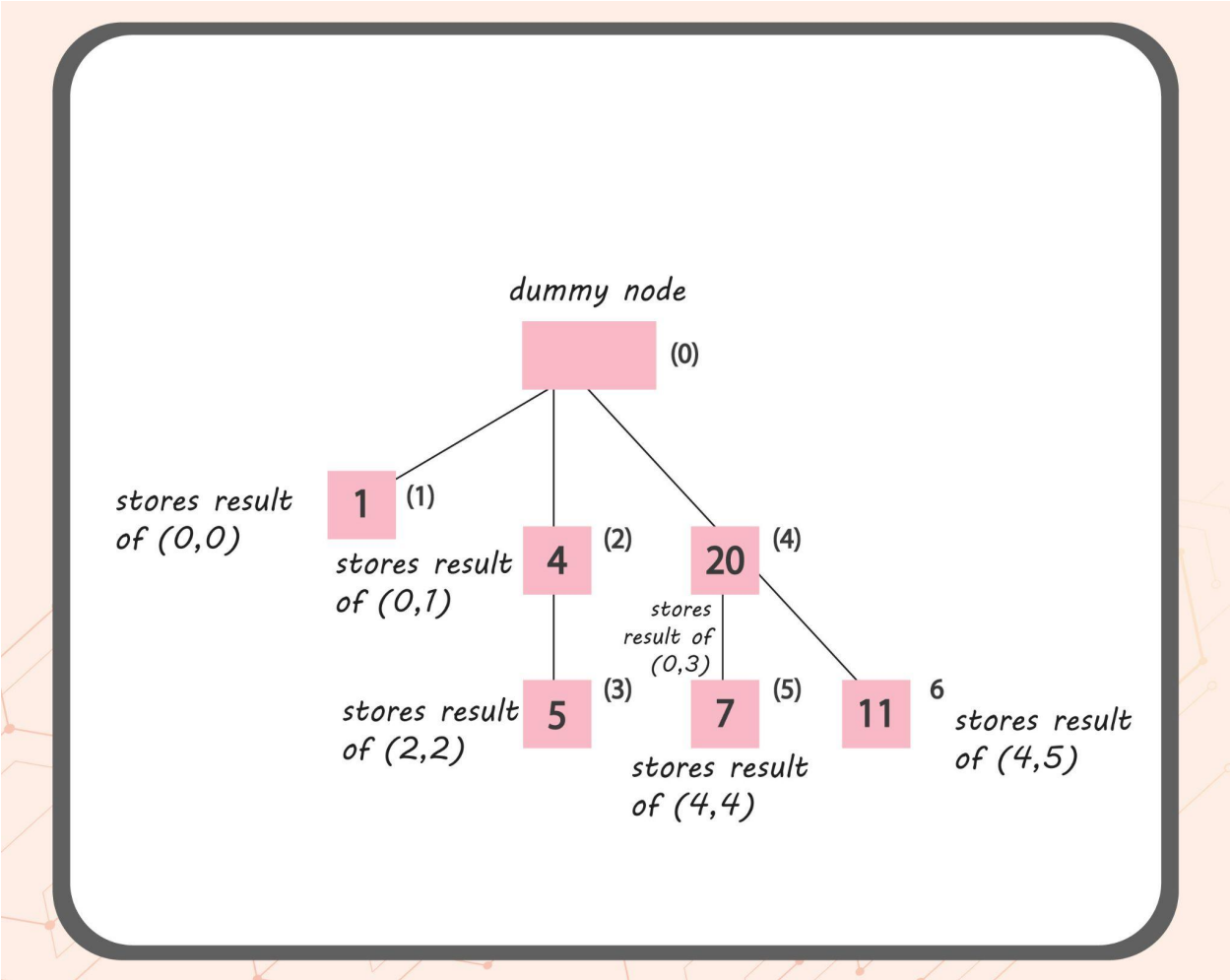
Index **3 = $2^1 + 2^0$**. Here, 2^1 represents that the parent of index 3 will be index 2 and since the result of (0,1) is stored at index 2, therefore index 3 will store the result of (2, 2).



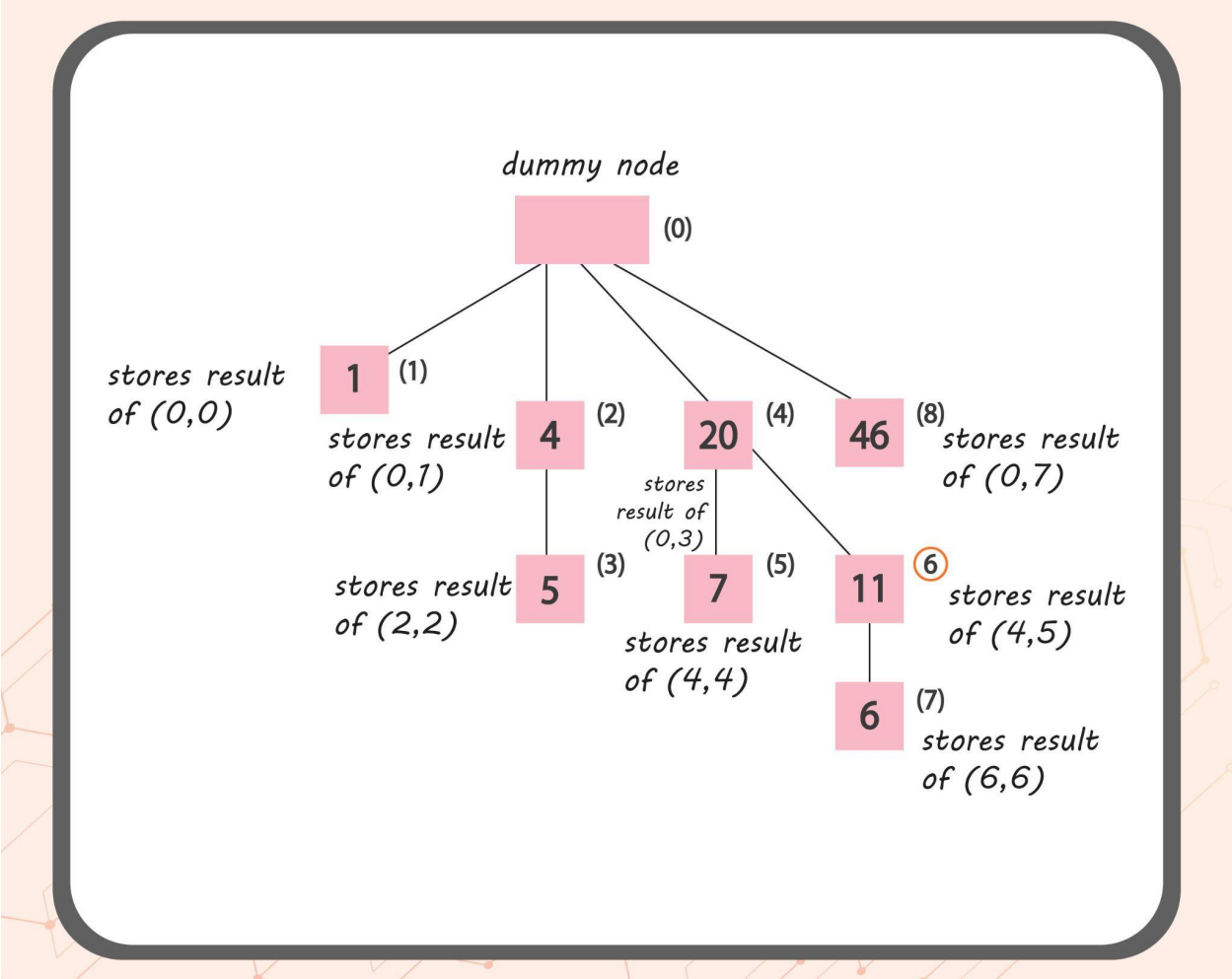
Index **4 = 2^2 + 0**, so 0 represents that index 4 will be a direct child of the 0th node and it will store the result of (0, 3).



Index **5 = 2^2 + 2^0**. 2^2 represents index 4, therefore index 5 will be the child of index 4 and will store the result of (4, 4).
Index **6 = 2^2 + 2^1**. Since 2^2 = index 4, then index 6 will be a child of index 4 and will store the result of (4, 5).



Similarly we can add the nodes for the remaining indices and our final tree will look something like -



Query on Fenwick tree

Example: Suppose we have to find the result of the query (0, 6).
Since we have started our indexing from 1 we go to index 6 + 1 = 7 **($2^2 + 2^1 + 2^0$)**.
If we subtract 2^0 from the above we get $2^2 + 2^1$ (Index 6), which is the parent of index 7.
Similarly, if we subtract 2^1 from the above result, we get 2^2 (Index 4), which is the parent of index 6.

Thus, we can conclude that, by removing / subtracting the smallest power of 2 (or the rightmost set bit), we reach the parent of the current node.

Update on Fenwick tree

Suppose we have to update the value at index 2 in the array **Arr = [1, 3, 5, 11, 7, 4, 6, 9]**. This essentially means that index 3 of the fenwick tree will be updated. But there will also be updates at index 4 and index 8 because, since 4 (2^2) and 8 (2^3) are powers of 2, they will be the direct children of the 0th node and will contain results to the queries with start index 0 thus affected by a change in the index 2 of the array .

Let's take a look at the code for building, updating and query on a fenwick tree.

Code:

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


void update(int index,int value,int* BIT,int n){

    for(;index <= n;index += index&(-index)){
        BIT[index] += value;
    }
```

```
}

int query(int index,int* BIT){
    int sum=0;
    for(;index >0;index -= index&(-index)){
        sum += BIT[index];
    }
    return sum;
}
int main(){

    int n;
    cin >> n;

    int* arr = new int[n+1]();
    int* BIT = new int[n+1]();

    for(int i=1;i<=n;i++){
        cin >> arr[i];
        update(i,arr[i],BIT,n);
    }

    cout << "Sum of first 5 elements " << query(5,BIT) <<endl;
    cout << "Sum of elements from 2 index to 6 index " << query(6,BIT) -
query(1,BIT) <<endl;
    return 0;
}
```

Coder Rating

Problem Statement: We have different coders C1, C2, C3 and so on. Their ratings depend on two factors **x** and **y**. For every coder we have to find out how many coders he/she is better from.

Explanation:

Example:

	x	y	Rating
C 1	1798	1832	6
C 2	862	700	0
C 3	1075	1089	2
C 4	1568	1557	4
C 5	2575	1984	7
C 6	1033	950	1
C 7	1656	1649	5
C 8	1014	1473	1

* C1 has 6 coders that he is above from
(C2, C2, C4, C6, C7, C8)

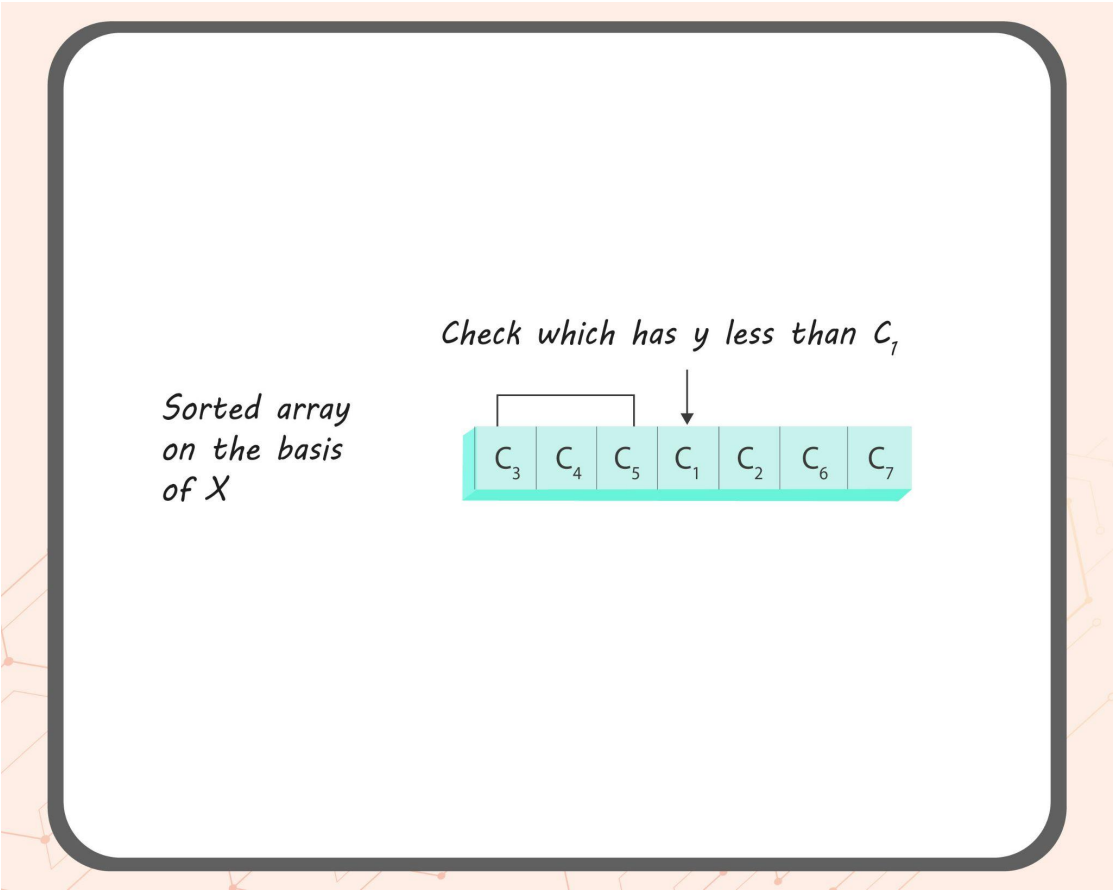
For a coder C1 to be better than C2, there can be three possibilities:

1. **C1.x > C2.x**
   C1.y > C2.y

2. **C1.x = C2.x**
   C1.y > C2.y

3. **C1.x > C2.x**
   C1.y = C2.y

If we think about sorting the array then we can do it only on the basis of one parameter. So let's say if we sort the array on the basis of **x** then we can see for every coder Ci that before him/her how many coders have **y** lesser than than Ci. For equality cases we will have to add an additional check.



Code:

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

struct coder
{
    int x,y,index;
};
int bit[100001];

void update(int y){

    for(;y<=100000;y+= y&(-y)){
        bit[y]++;
    }
}

int query(int y){
    int value = 0;
    for(;y>0;y-= y&(-y)){
        value += bit[y];
    }
    return value;
}
bool operator < (coder A, coder B){
    if(A.x == B.x){
        return A.y < B.y;
    }
    return A.x < B.x;
}
int main(){

    int n;
    cin >> n;
    coder arr[n];

    for(int i=0;i<n;i++){
        cin>>arr[i].x >> arr[i].y;
        arr[i].index = i;
```

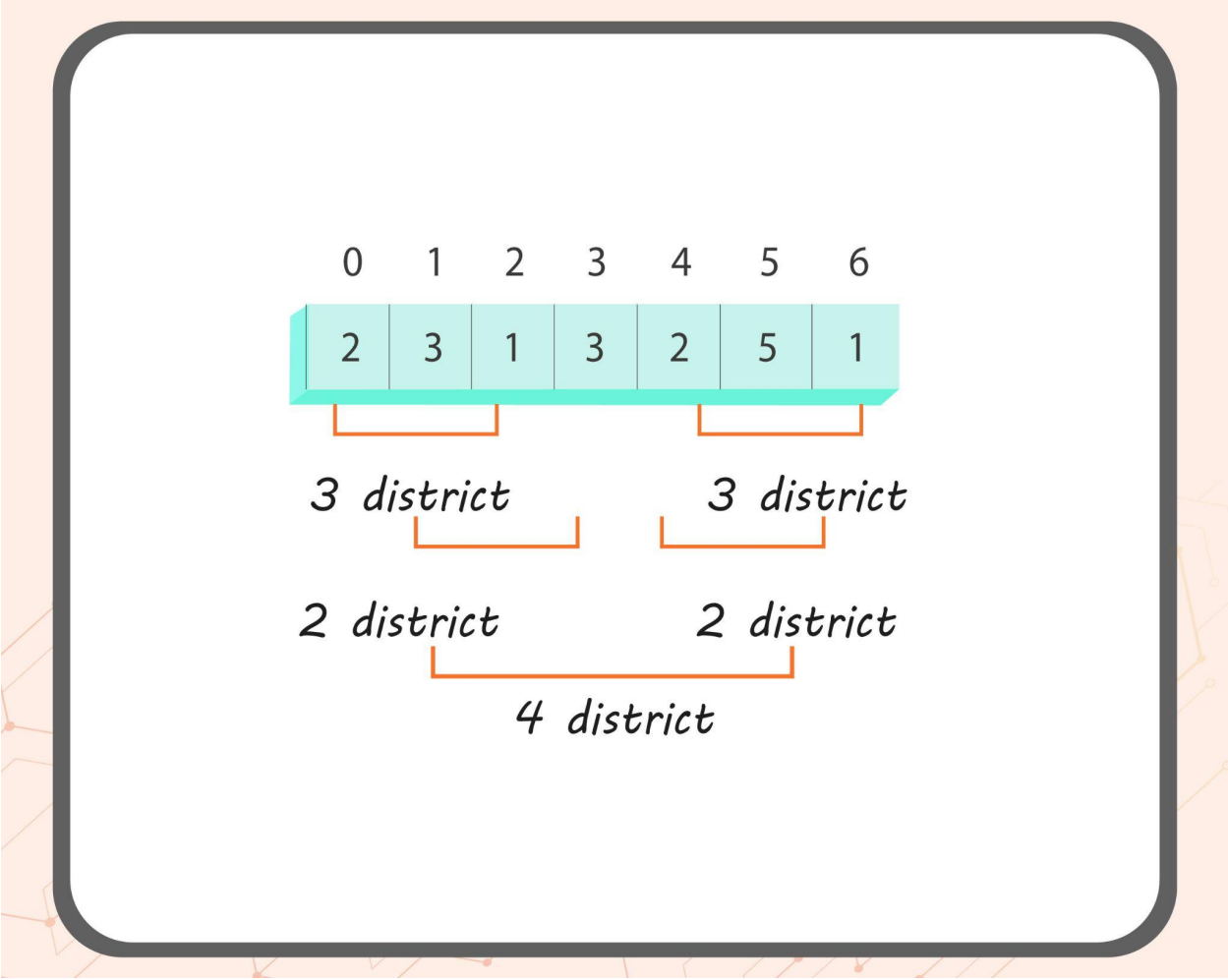
```
    }

    sort(arr,arr+n);
    int ans[n];
    for(int i=0;i<n;){
        int endIndex = i;

        while(endIndex <n && arr[endIndex].x == arr[i].x && arr[endIndex].y
== arr[i].y){
            endIndex++;
        }
        // query

        for(int j=i;j<endIndex;j++){
            ans[arr[j].index] = query(arr[j].y);
        }

        for(int j=i;j<endIndex;j++){
            update(arr[j].y);
        }

        i = endIndex;

        // update
    }

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


    return 0;
}
```

Distinct Query Problem

Problem Statement: Given a sequence of n numbers a1, a2, ..., an and a number of d-queries. A d-query is a pair (i, j) (1 ≤ i ≤ j ≤ n). For each d-query (i, j), you have to return the number of distinct elements in the subsequence ai, ai+1, ..., aj.

Explanation:

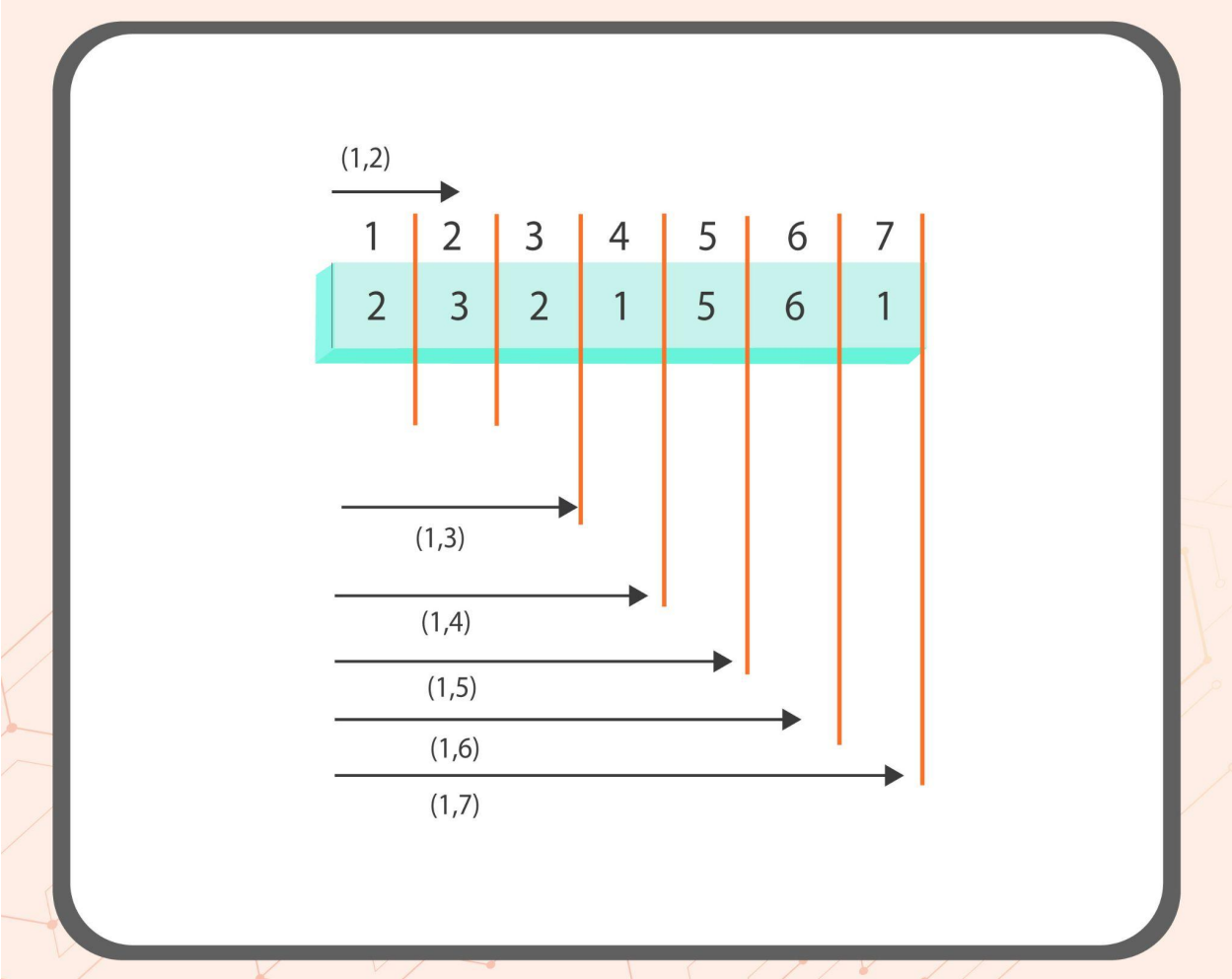
Example:



To solve this problem we will use the concept of **Offline Programming.**

Offline Programming VS Online Programming

In Online programming we calculate the result as soon as the query arrives whereas in Offline programming we store the results of the different possible queries beforehand.



We will use a Binary Indexed Tree to store the results of queries.

We will store the result of the coming queries on the basis of the end index of the query. That is, first of all we will store the result of the query (1, 1) then for (1, 2), then for (1, 3) and so on until the query (1, 7). For this we will maintain a query array and sort this on the basis of the end index of the queries.

Code:

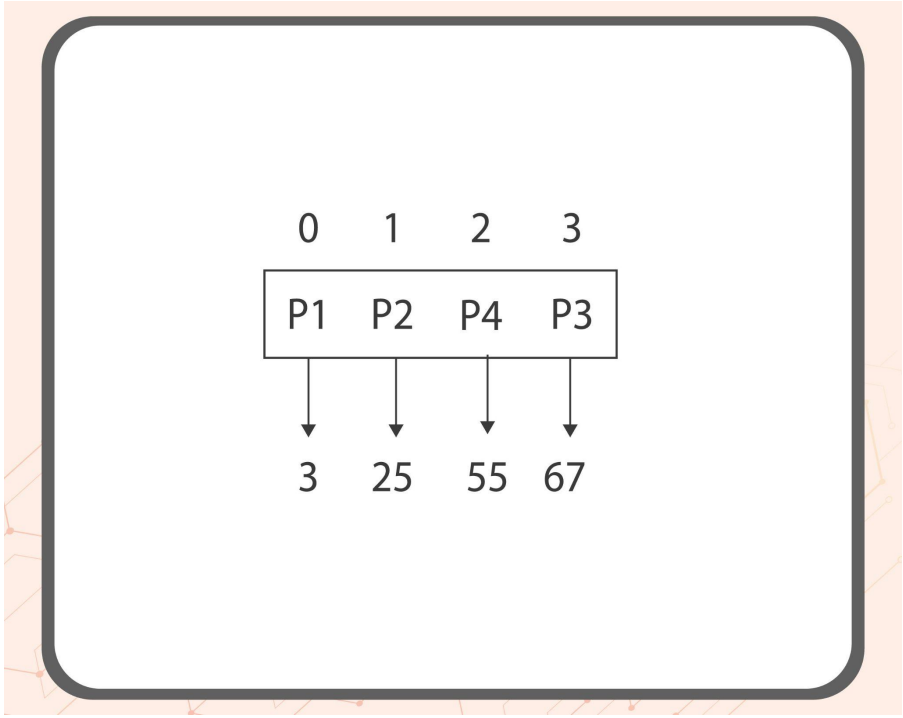
```
#include<bits/stdc++.h>
using namespace std;

struct coder
{
    int first,second,index;
}query[200000];

bool operator < (coder A, coder B){

    return A.second < B.second;
}

int bit[30001];
void update(int index,int value,int n){

    for(;index<=n;index += index &(-index)){
        bit[index] += value;
    }
}
int last[1000001];
int arr[30001];
int value(int index){

    int res= 0;
    for(;index >0 ;index -= index & (-index)){
        res += bit[index];
    }
    return res;
}

int main(){
    int n;
    cin >> n;


    for(int i=1;i<=n;i++){
        cin >> arr[i];
```

```
    }

    int q;
    cin>>q;

    for(int i=0;i<q;i++){
        cin>>query[i].first>>query[i].second;
        query[i].index = i;
    }

    sort(query,query+q);



    memset(last,-1,sizeof(last));



    int k=0;
    int ans[q];


    int total = 0;
    for(int i=1;i<=n;i++){


        if(last[arr[i]] != -1){
            update(last[arr[i]],-1,n);
        }else{
            total++;
        }

        update(i,1,n);
        last[arr[i]] = i;


        while(k<q && query[k].second == i){

            ans[query[k].index] = total - value(query[k].first-1);
            k++;
```

```
        }

    }


    for(int i=0;i<q;i++){
        cout << ans[i] <<endl;
    }
    return 0;
}
```

Coordinate Compression

Suppose we are given some points **P1 (3), P2 (25), P3 (67), P4 (55)** along with their coordinates. Now, the ordering of P1, P2, P3, P4 will be 0, 1, 3, 2 respectively.



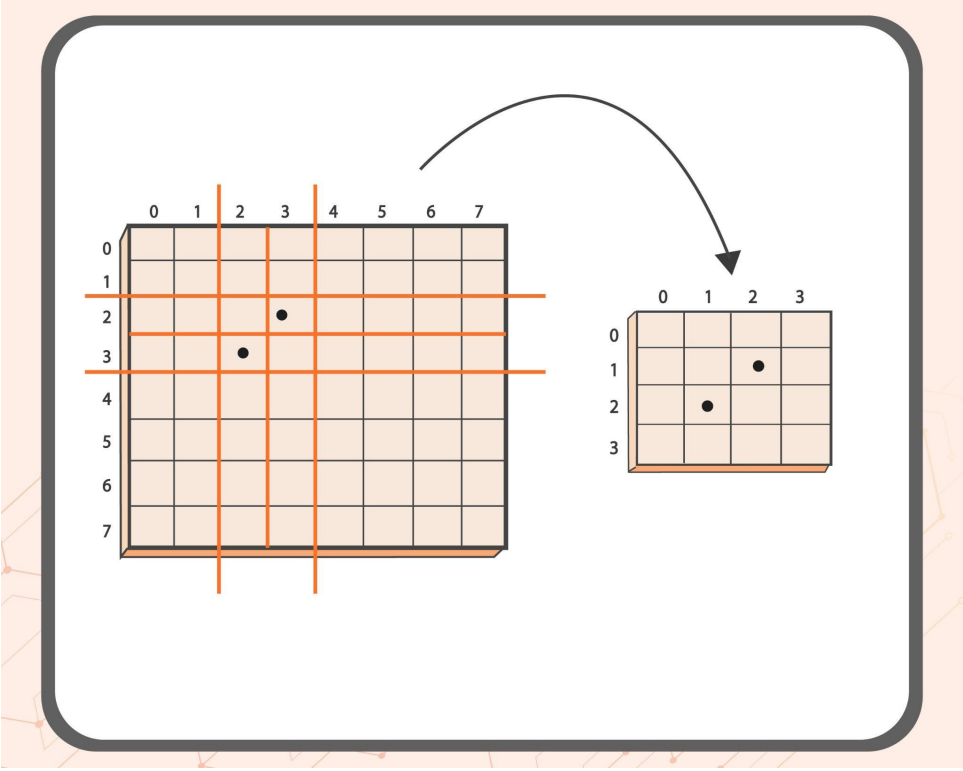
We will take input of the coordinates and then sort them and give the respective ranking to them.
We can also store the ranks in a map or an array.
Time Complexity of this will be the same as that of sorting that is **O(Nlog(N)**.

Let us see how the above concept can be used in coordinate compression.

Consider a problem in which we are given a grid with two obstacles in it and we have to somehow deal with the obstacles.

Note: This isn't an actual problem so how to deal with the obstacles are not our concern. We just have to find a way to reduce the overall effort and complexity of the solution.

The empty rectangles are kind of redundant so the above grid can be effectively reduced to



This can be achieved using the concept of mapping which we discussed above in the beginning of this section.

Mapping rows from initial grid to new grid :

- **Rows**
  - **0, 1 → 0**
  - **2 → 1**
  - **3 → 2**
  - **4, 5, 6, 7 → 3**

- **Columns**
  - **0, 1 → 0**
  - **2 → 1**
  - **3 → 2**
  - **4, 5, 6, 7 → 3**

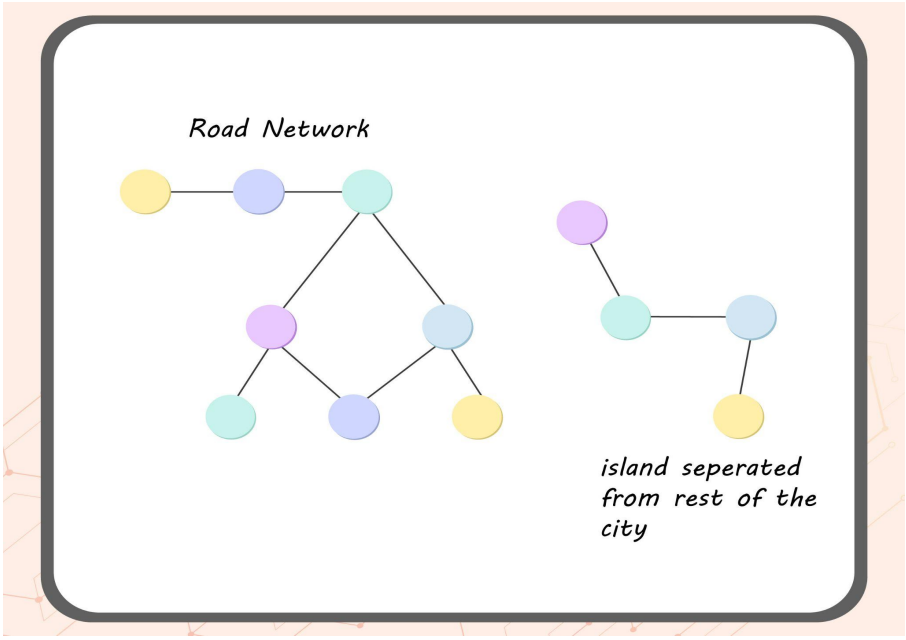
Graphs 1

Introduction

A **graph** is a pair $G = (V, E)$, where V is a set whose elements are called *vertices*, and E is a set of two-sets of vertices, whose elements are called *edges*.

The vertices x and y of an edge $\{x, y\}$ are called the *endpoints* of the edge. The edge is said to *join* x and y and to be *incident* on x and y. A vertex may not belong to any edge.

For example: Suppose there is a road network in a country, where we have many cities, and roads are connecting these cities. There could be some cities that are not connected by some other cities like an island. This structure seems to be non-uniform, and hence, we can't use trees to store it. In such cases, we will be using graphs. Refer to the figure for better understanding.

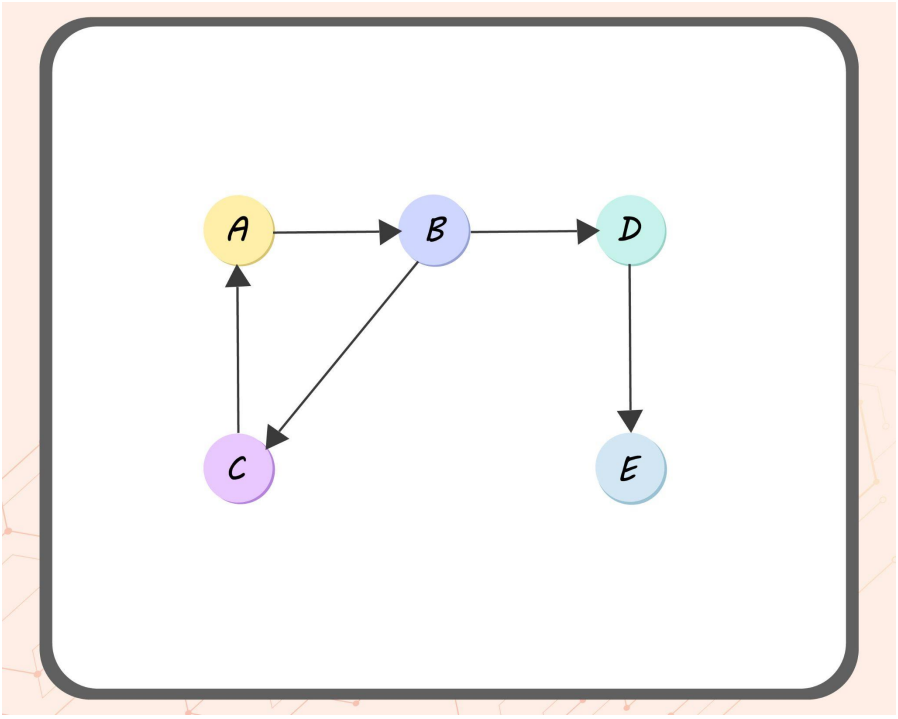


Graphs Terminology

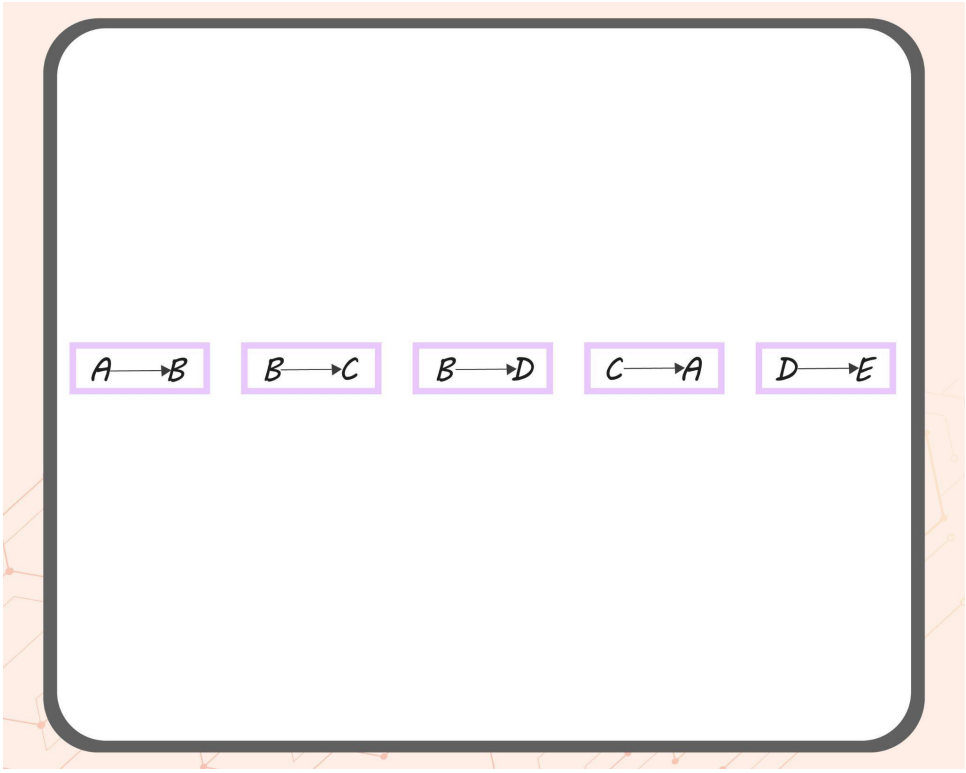
- Nodes are named as **vertices,** and the connections between them are called **edges**.
- Two vertices are said to be **adjacent** if there exists a direct edge connecting them.
- The **degree** of a node is defined as the number of edges that are incident to it.
- A **path** is a collection of edges through which we can reach from one node to the other node in a graph.
- A graph is said to be **connected** if there is a path between every pair of vertices.
- If the graph is not connected, then all the connected subsets of the graphs are called **connected components**. Each component is connected within the self, but two different components of a graph are never connected.
- The minimum number of edges in a graph can be zero, which means a graph could have no edges as well.
- The minimum number of edges in a connected graph will be (n-1), where n is the number of nodes.
- In a complete graph (where each node is connected to every other node by a direct edge), there are $^{n}C_{2}$ number of edges means (n * (n-1)) / 2 edges, where n is the number of nodes. It is the maximum number of edges that a graph can have. Hence, if an algorithm works on the terms of edges, let's say O(E), where E is the number of edges, then in the worst case, the algorithm will take $O(n^2)$ time, where n is the number of nodes.

Graphs Implementation

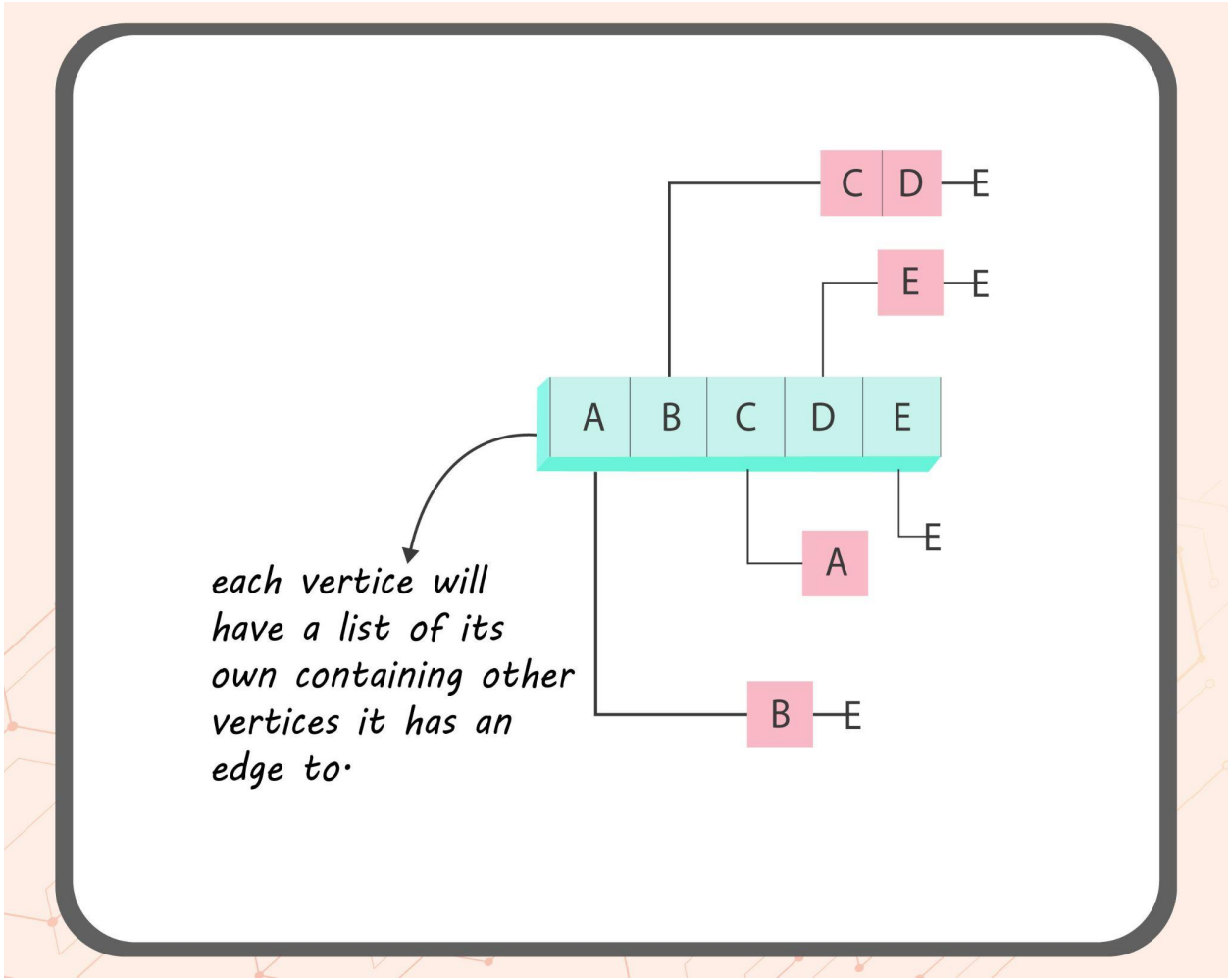
There are the following ways to implement a graph:



1. **Using edge list:** We can create a class that could store an array of edges. The array of edges will contain all the pairs that are connected, all put together in one place. It is not preferred to check for a particular edge connecting two nodes; we have to traverse the complete array leading to $O(n^2)$ time complexity in the worst case. Pictorial representation for the above graph using edge list is given below:



2. **Adjacency list:** We will create an array of vertices, but this time, each vertex will have its list of edges connecting this vertex to another vertex. Now to check for a particular edge, we can take any one of the nodes and then check in its list if the target node is present or not. This will take O(n) work to figure out a particular edge. Visually, it looks as follows:



3. **Adjacency matrix:** Here, we will create a 2D array where the cell (i, j) will denote an edge between node i and node j. It is the most reliable method to implement a graph in terms of ease of implementation. We will be using the same throughout the session. The major disadvantage of using the adjacency matrix is vast space consumption compared to the adjacency list, where each node stores only those nodes that are directly connected to them. For the above graph, adjacency matrix looks as follows:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

Note: In case the graph is not directed then we have to mark two points corresponding for every edge. For example if the given graph was not directed then the adjacency matrix would have been as followed:

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	0	0
D	0	1	0	0	1
E	0	0	0	1	0

This is because for every edge X to Y, we also have to mark [Y][X] index along with [X][Y] index in case of a non-directed graph.

DFS - Adjacency matrix

There are the following ways to implement a graph:

Here, if we have n vertices(labelled: 0 to n-1). Then we will be asking the user for the number of edges. We will run the loop from 0 to the number of edges, and at each

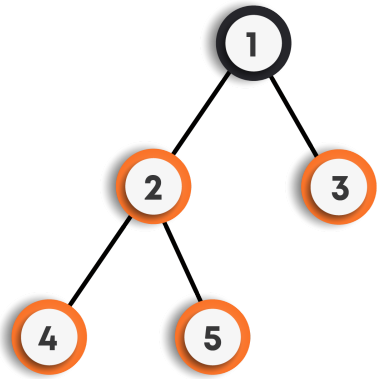
iteration, we will take input for the two connected nodes and correspondingly update the adjacency matrix. Let's look at the code for better understanding.

Code:

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

void print(int** edges, int n, int sv, bool* visited){
  cout << sv << endl;
  visited[sv] = true;           // marked the starting vertex true
  for(int i=0; i<n; i++){       // Running the loop over all n nodes and
                                // checking if there is an edge between
                                // sv and i
    if(i==sv){
      continue;
    }
    if(edges[sv][i]==1){        // As the edge is found, we then checked if
                                // the node i was visited or not
      if(visited[i]){
        continue;
      }
      print(edges, n, i, visited);  // Otherwise, recursively called over
                                    // node i taking it as starting vertex
    }
  }
}

int main(){
  int n;                // Number of nodes
  int e;                // Number of edges
  cin >> n >> e;

  int** edges = new int*[n];    // adjacency matrix of size n*n
  for(int i=0; i<n; i++){
    edges[i]=new int[n];
    for(int j=0; j<n; j++){
      edges[i][j]=0;            // 0 indicates that there is no edge
```

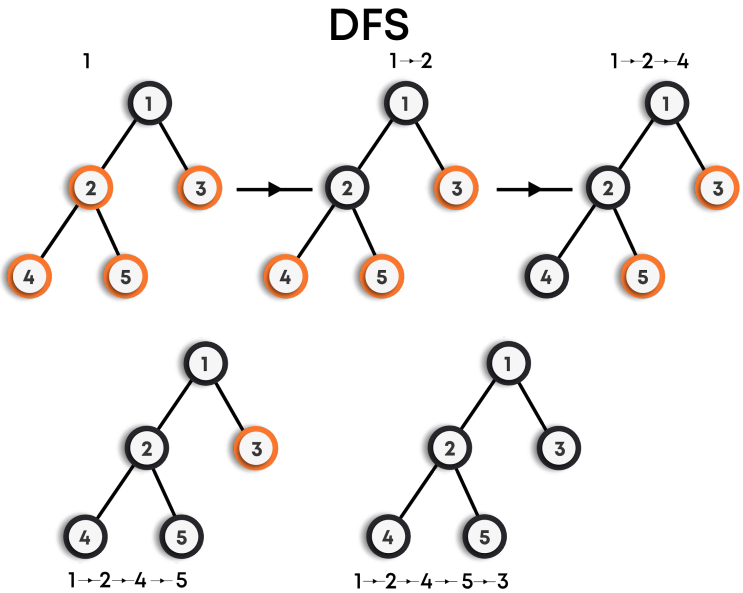
```
                                    // between i and j
    }
  }

  for(int i=0; i<e; i++){
    int f,s;
    cin >> f >> s;                  // Nodes having edges between them
    edges[f][s]=1;                  // marking f to s as 1
    edges[s][f]=1;                  // also, marking s to f as 1
  }

  bool* visited = new bool[n];        // this is used to keep the track of
                                      //nodes if we have visited them or not.
  for(int i=0; i<n; i++){
    visited[i]=false;               // Marking all the nodes as false which
                                    //means not visited
  }

  print(edges, n, 0, visited);   // starting vertex is taken as 0
  // Delete all the memory: Do it yourselves
  return 0;
}
```

Let’s take an example graph:



On dry running the above code, the output will be 1 2 4 5 3.

Here, we are starting from a node, going in one direction as far as we can, and then we return and do the same on the previous nodes. This method of graph traversal is known as the **depth-first search (DFS)**. As the name suggests, this algorithm goes into the depth first and then recursively does the same in other directions. Follow the figure below, for step-by-step traversal using DFS.

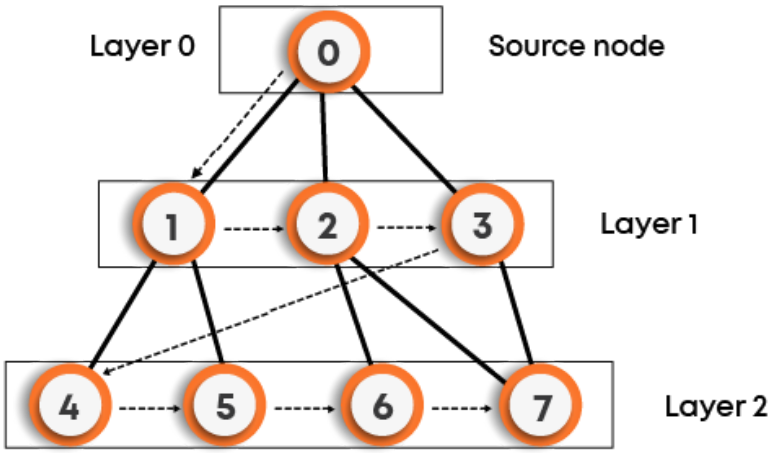


BFS Traversal

Breadth-first search(BFS) is an algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

As the name suggests:

- We first move horizontally and visit all the nodes of the current layer.
- Then move to the next layer.



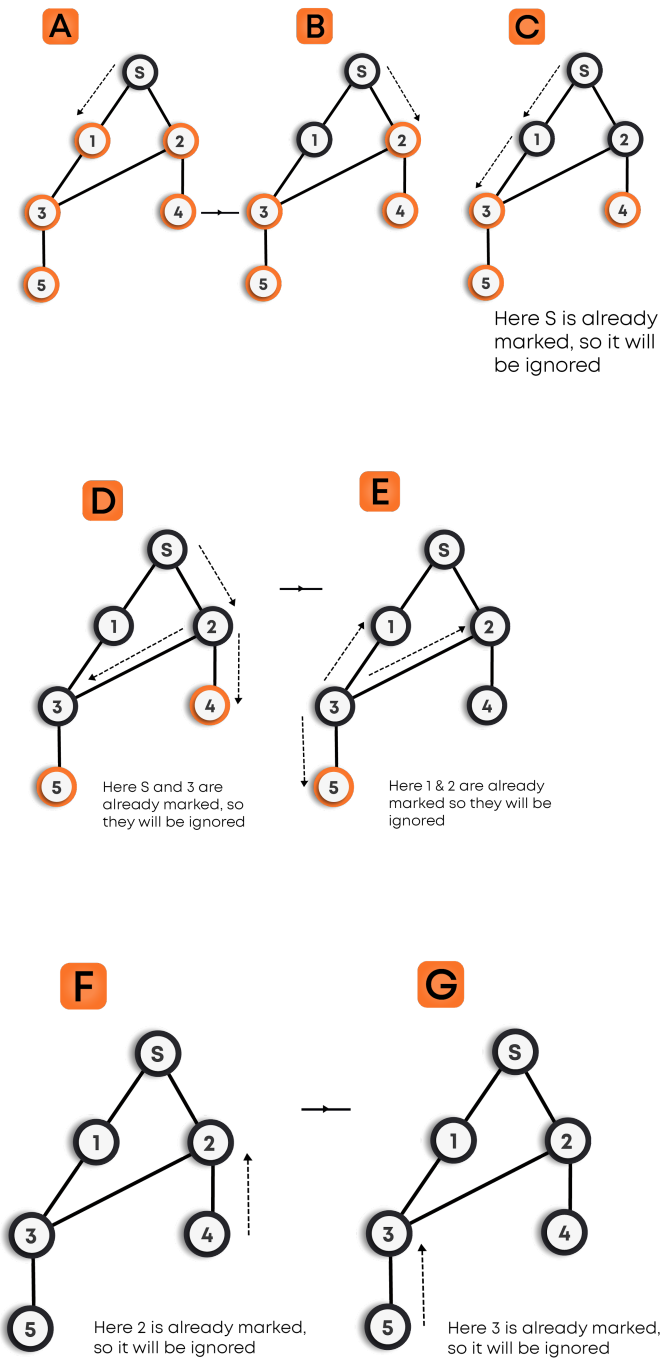
This is an iterative approach. We will use the queue data structure to store the child nodes of the current node and then pop out the current node. This process will continue until we have covered all the nodes. Remember to put only those nodes in the queue which have not been visited.

Code:

```
#include <queue>
#include <iostream>
#include <unordered_map>

void printBFS(int **edges, int n, int sv, bool *visited) {
    queue<int> pendingVertices;        // queue
    pendingVertices.push(sv);          // starting vertex directly pushed
    visited[sv] = true;
    while (!pendingVertices.empty()) { // until the size of queue is not 0
        // stored the top of queue
        int currentVertex = pendingVertices.top();
        pendingVertices.pop(); // deleted that top element
        cout << pendingVertices << " ";
        for (int i = 0; i < n; i++) {    // now checked for its vertices
            if (i == currentVertex) {
                continue;
            }
            if (edges[currentVertex][i] == 1 && !visited[i]) {
                pendingVertices.push(i); // if found, then inserted into
                                         // queue
                visited[i] = true;
            }
        }
    }
}

void BFS(int **edges, int n) {
    bool *visited = new bool[n];// visited array to keep the track of nodes
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    printBFS(edges, n, 0, visited);      // starting vertex  = 0
    delete [ ] visited;                  // deleted the visited array
}

int main() {
    int n;          // Number of nodes
```

```
    int e;          // Number of edges
    cin >> n >> e;
    int **edges = new int*[n];            // Adjacency matrix
    for (int i = 0; i < n; i++) {
        edges[i] = new int[n];
        for (int j = 0; j < n; j++) {
            edges[i][j] = 0;
        }
    }

    for (int i = 0; i < e; i++) {
        int f, s;
        cin >> f >> s;
        edges[f][s] = 1;
        edges[s][f] = 1;
    }

    BFS(edges, n);
    // delete the memory
    for (int i = 0; i < n; i++) {
        delete [ ] edges[i];
    }
    delete [ ] edges;
    return 0;
}
```

Consider the dry run over the example graph below for a better understanding of the same:



BFS & DFS for disconnected graph

Till now, we have assumed that the graph is connected. For the disconnected graph, there will be a minor change in the above codes. Just before calling out the print functions, we will run a loop over each node and check if that node is visited or not. If not visited, then we will call a print function over that node, considering it as the starting vertex. In this way, we will be able to cover up all the nodes of the graph.

Consider the same for the BFS function. Just replace this function in the above code to make it work for the disconnected graph too.

Code:

```
void BFS(int **edges, int n) {
    bool *visited = new bool[n];  //visited array to keep track of nodes
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    for (int i = 0; i < n; i++) {      // run a loop over each node
        if (!visited[i]) { // if a node is not visited, then called print()
           printBFS(edges, n, i, visited);  //on it taking it as starting
vertex
        }
    }
    delete [ ] visited;     // deleted the visited array
}
```

Has Path

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), check if there exists any path between them or not. Print true or false. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

Approach: This can be simply solved by considering the vertex v1 as starting vertex and then run either BFS or DFS as per your choice, and while traversing if we reach the vertex v2, then we will simply return true, otherwise return false.

This problem has been left for you to try yourself. For code, refer to the solution tab of the same.

Get Path - DFS

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), find and print the path from v1 to v2 (if exists). Print nothing if there is no path between v1 and v2.
Find the path using DFS and print the first path that you encountered irrespective of the length of the path.
V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1.
E is the number of edges present in graph G.
Print the path in reverse order. That is, print v2 first, then intermediate vertices and v1 at last.
Example: Suppose the given input is:

4 4
0 1
0 3
1 2

2 3
1 3

The output should be:

3 0 1

Explanation: Here, v1 = 1 and v2 = 3. The connected vertex pairs are (0, 1), (0, 3), (1, 2) and (2, 3). So, according to the question, we have to print the path from vertex v1 to v2 in reverse order using DFS only; hence the path comes out to be {3, 0, 1}.
Approach: We have to solve this problem by using DFS. Suppose, if the start and end vertex are the same, then we simply need to put the start in the solution array and return the solution array. If this is not the case, then from the start vertex, we will call DFS on the direct connections of the same. If none of the paths leads to the end vertex, then we do not need to push the start vertex as it is neither directly nor indirectly connected to the end vertex, hence we will simply return NULL. In case any of the neighbors return a non-null entry, it means that we have a path from that neighbor to the end vertex, hence we can now insert the start vertex into the solution array.
Try to code it yourself, and for the answer, refer to the solution tab of the same.

Get Path - BFS

Approach: It is the same problem as the above, just we have to code the same using BFS.

Approach: Using BFS will provide us with the shortest path between the two vertices. We will use the queue over here and do the same until the end vertex gets inserted into the queue. Here, the problem is how to figure out the node, which led us to the end vertex. To overcome this, we will be using a map. In the map, we will

store the resultant node as the index, and its key will be the node that led it into the queue. For example: If the graph was such that 0 was connected to 1 and 0 was connected to 2, and currently, we are on node 0 such that node 1 and node 2 are not visited. So our map will look like as follows:

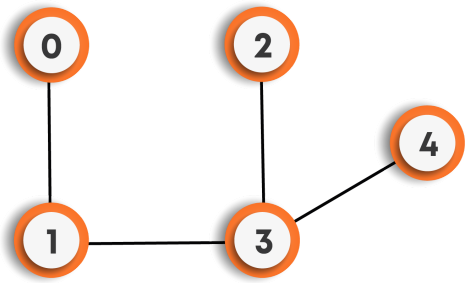
1	0
2	0

This way, as soon as we reach the end vertex, we can figure out the nodes by running the loop until we reach the start vertex as the key value of any node.

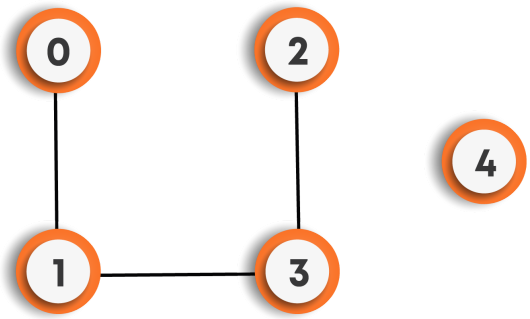
Try to code it yourselves, and for the solution, refer to the specific tab of the same.

Is connected?

Problem statement: Given an undirected graph G(V, E), check if the graph G is a connected graph or not. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.



Connected graph



Disconnected graph

Example 1: Suppose the given input is:

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

The output should be: **true**

Explanation: As the graph is connected, so according to the question, the answer will be true.

Example 2: Suppose the given input is:

4 3
0 1
1 3
0 3

The output should be: **false**

Explanation: The graph is not connected, even though vertices 0,1 and 3 are connected to each other, but there isn't any path from vertices 0,1,3 to vertex 2. Hence, according to the question, the answer will be false.

Approach: This is very start-forward. Take any vertex as the starting vertex and traverse the graph using either DFS or BFS. In the end, check if all the vertices are visited or not. If not, it means that the node was not connected to the starting vertex, which means it is a disconnected graph. Otherwise, it is a connected graph. Try to code it yourselves, and for the code, refer to the solution tab of the same.

Return all connected components

Problem statement: Given an undirected graph G(V, E), find and print all the connected components of the given graph G. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

You need to take input in the main and create a function that should return all the connected components. And then print them in the main, not inside a function. Print different components in a new line. And each component should be printed in increasing order (separated by space). The order of different components doesn't matter.

Example: Suppose the given input is:

4 3
0 1
1 3
0 3

The output should be:

0 1 3
2

Explanation: As we can see that {0, 1, 3} is one connected component, and {2} is the other one. So, according to the question, we just have to print the same.

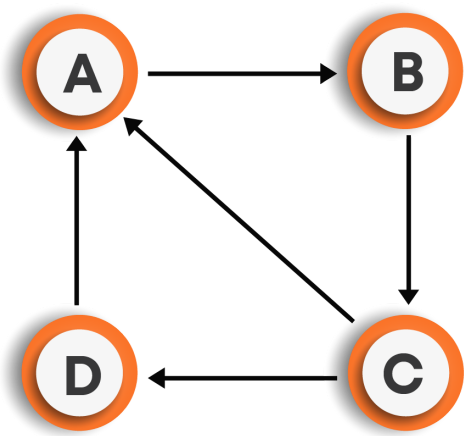
Approach: For this problem, start from vertex 0 and traverse until vertex n-1. If the vertex is not visited, then run DFS/BFS on it and keep track of all the connected vertices through that node. This way, we will get all the distinct connected components, and we can print them at last.

This problem is left for you to solve. For the code, refer to the solution tab of the same.

Weighted and directed graphs

There are two more variations of the graphs:

- **Directed graphs:** These are generally required when we have one-way routes. Suppose you can go from node A to node B, but you cannot go from node B to node A. Another example could be of social media(like Twitter) if you are following someone, it does not mean that they are following you too.



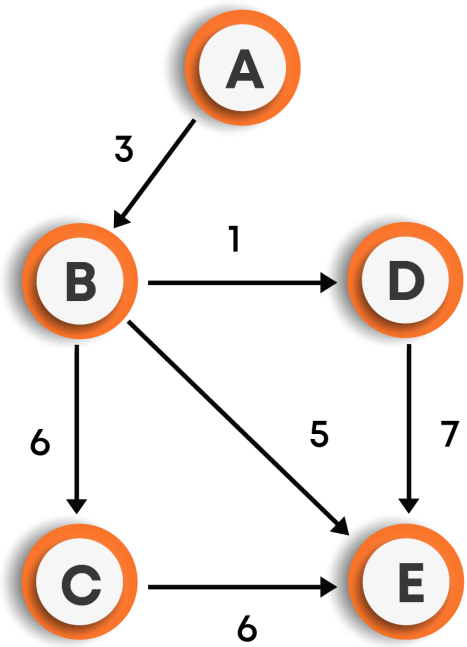
To implement these, there is a small change in the implementation of indirect graphs. In indirect graphs, if there was an edge between node i and j, then we did:

```
edges[i][j] = 1;
edges[j][i] = 1;
```

But, in the case of a directed graph, we will just do the following:

```
edges[i][j] = 1;
```

- **Weighted graphs:** These generally mean that all the edges are not equal, meaning somehow, each edge has some weight assigned to it. This weight can be the length of the road connecting the cities or many more.



To implement this, in the edges matrix, we will assign a weight to connected nodes instead of putting it 1 at that position. For example: If node i and j are connected, and the weight of the edge connecting them is 5, then edges[i][j] = 5.

Graphs 2

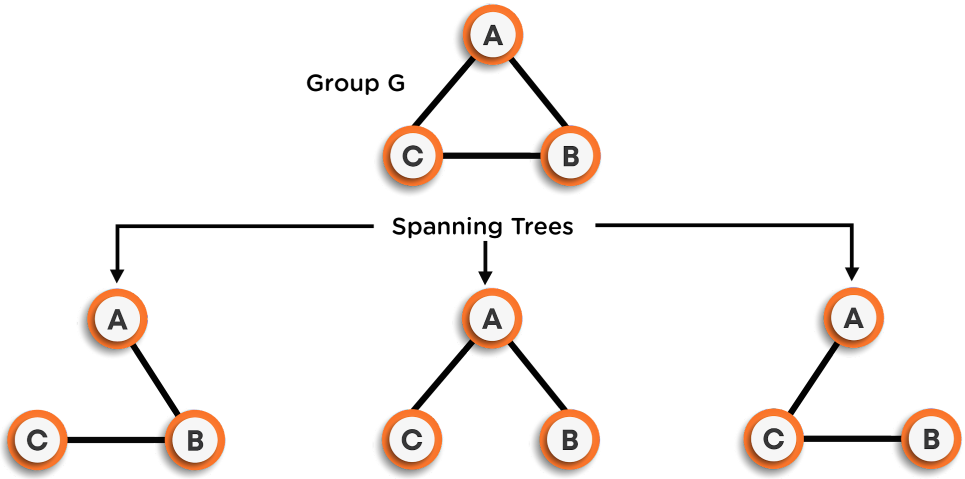
MST & Kruskal's introduction

As discussed earlier, a tree is a graph, which:

- Is always connected.
- Contains no cycle.

If we are given an undirected and connected graph, a **spanning tree** means a tree that contains all the vertices of the same. For a given graph, we can have multiple spanning trees.

Refer to the example below for a better understanding of spanning trees.



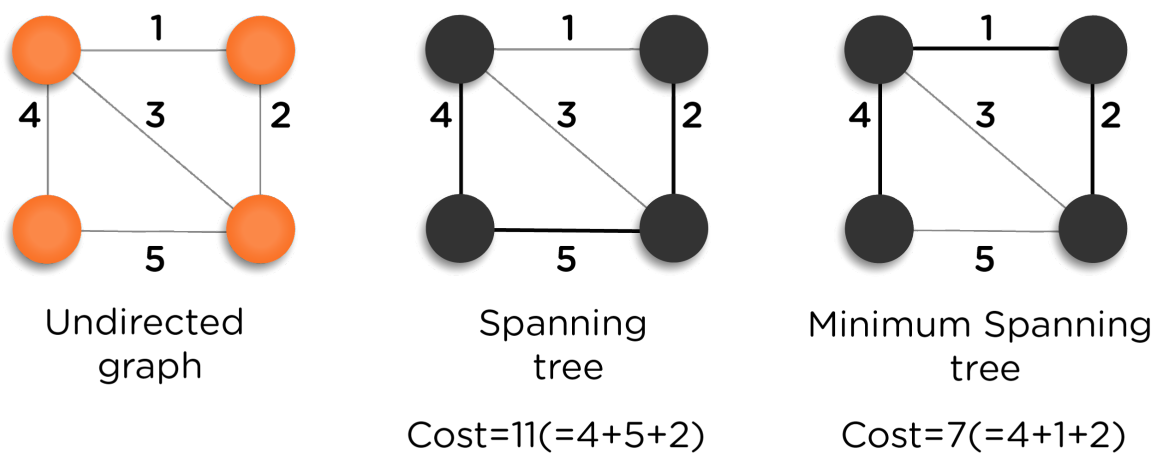
If there are *n* vertices and *e* edges in the graph, then any spanning tree corresponding to that graph contains *n* vertices and *n-1* edges.

Properties of spanning trees:

- A connected and undirected graph can have more than one spanning tree.
- Spanning tree is free of loops, i.e., it is acyclic.
- Removing any one of the edges will make the graph disconnected.
- Adding an extra edge to the spanning tree will create a loop in the graph.

Minimum Spanning Tree(MST) is a spanning tree with weighted edges.

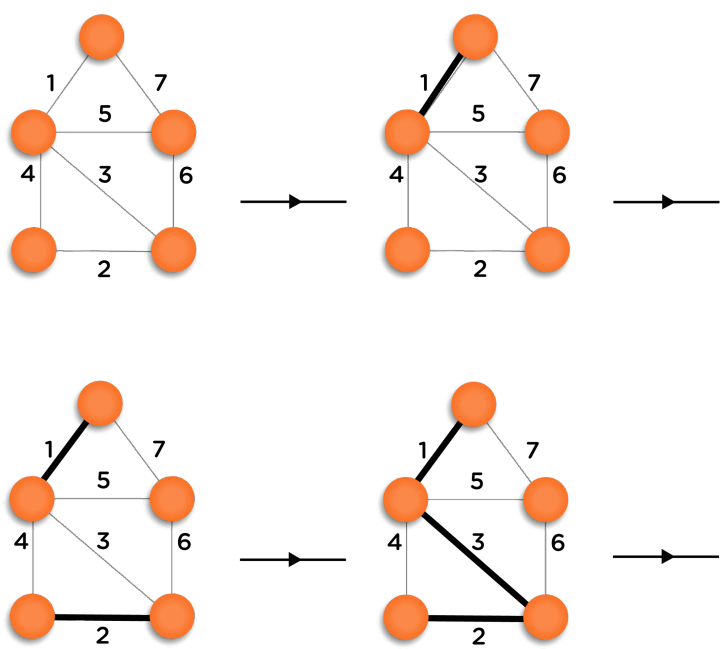
In a weighted graph, the MST is a spanning tree with minimum weight than all other spanning trees of that graph. Refer to the image below for better understanding.

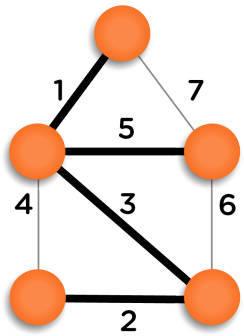


Kruskal's Algorithm:

This algorithm is used to find MST for the given graph. It builds the spanning tree by adding edges one at a time. We start by picking the edge with minimum weight, adding that edge into the MST, and increasing the count of edges in the spanning tree by one. Now, we will be picking the minimum weighted edge by excluding the already chosen ones and correspondingly increasing the count. While choosing the edge, we will also make sure that the graph remains acyclic after including the same. This process will continue until the count of edges in the MST reaches *n-1*. Ultimately, the graph obtained will be MST.

Refer to the example below for a better understanding of the same.





This is the final MST obtained using Kruskal's algorithm. It can be checked manually that the final graph is the MST for the given graph.

Cycle Detection

While inserting a new edge in the MST, we have to check if introducing that edge makes the MST cyclic or not. If not, then we can include that edge, otherwise not.

Now, let's figure out a way to detect the cycle in a graph. The following are the possible cases:

- By including an edge between the nodes A and B, if both the nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.
- Out of two vertices, if any of them has not been visited (or not present in the MST), then that vertex can also be included in the MST.
- If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can't use this method to detect the presence of the cycle.

Let's think of a better approach. We have already solved the **hasPath** question in the previous module, which returns true if there is a path present between two vertices v1 and v2, otherwise false.

Now, before adding an edge to the MST, we will check if a path between two vertices of that edge already exists in the MST or not. If not, then it is safe to add that edge to the MST.

As discussed in previous lectures, the time complexity of the **hasPath** function is O(E+V), where E is the number of edges in the graph and, V is the number of vertices. So, for (n-1) edges, this function will run (n-1) times, leading to bad time complexity, as in the worst case, $E = V^2$.

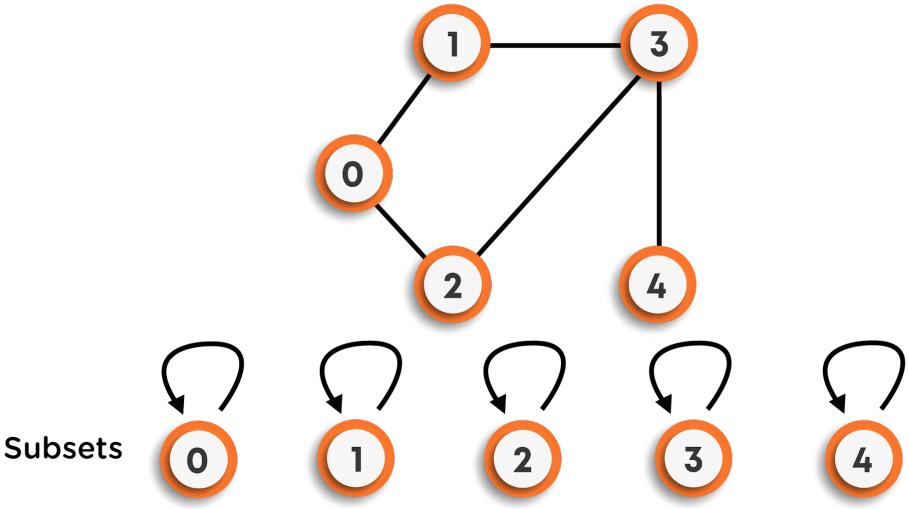
Now, moving on to a better approach for cycle detection in the graph.

Union-find algorithm:

Before adding any edge to the graph, we will check if the two vertices of the edge lie in the same component of MST or not. If not, then it is safe to add that edge to the MST.

Following the steps of the algorithm:

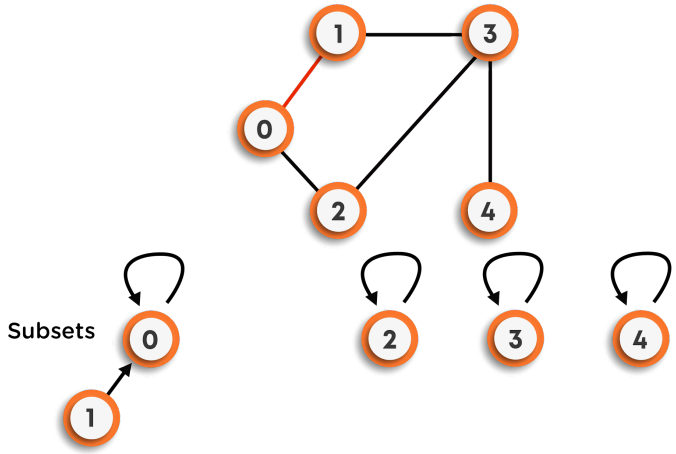
- We will assume that initially, the total number of disjoint sets is equal to the number of vertices in the graph starting from 0 to n-1.
- We will maintain a parent array specifying the parent vertex of each of the vertices of the graph. Initially, as each vertex belongs to the different disjoint set (connected component), hence each vertex will be its own parent.



Initially all parent pointers are pointing to self
means only one element in each subset

- Now, before inserting any edge into the MST, we will check the parent of the vertices. If their parent vertices are equal, they belong to the same connected component; hence it is unsafe to add that edge. Otherwise, we can add that edge into the MST, and simultaneously update the parent array so that they belong to the same component(Refer to the code on how to do so).

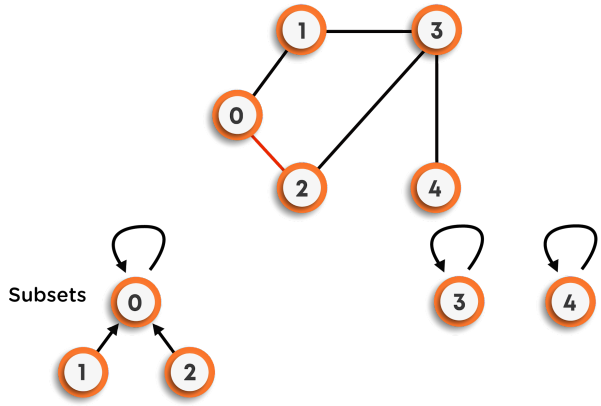
  Look at the following example, for better understanding:



Edge 0-1

Find: 0 belongs to subset 0 and 1 belongs to subset 1 so they are in different subsets.

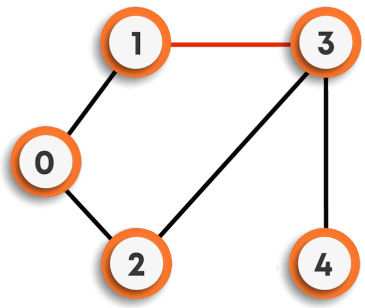
Union: Make 0 as the parent of 1, Updated set is {0,1}. 0 is the set representative since 0 is parent for itself.

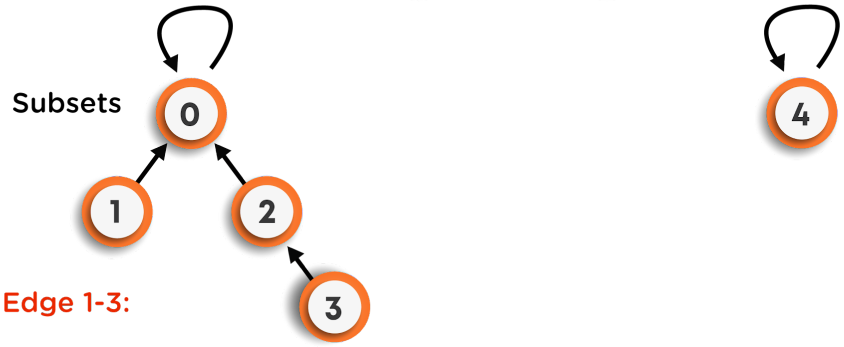


Edge 0-2:

Find: 0 belongs to subset 0 and 2 belongs to subset 2 so they are in different subsets.

Union: Make 0 as the parent of 2, Updated set is {0,1,2}. 0 is the set representative since 0 is parent for itself.

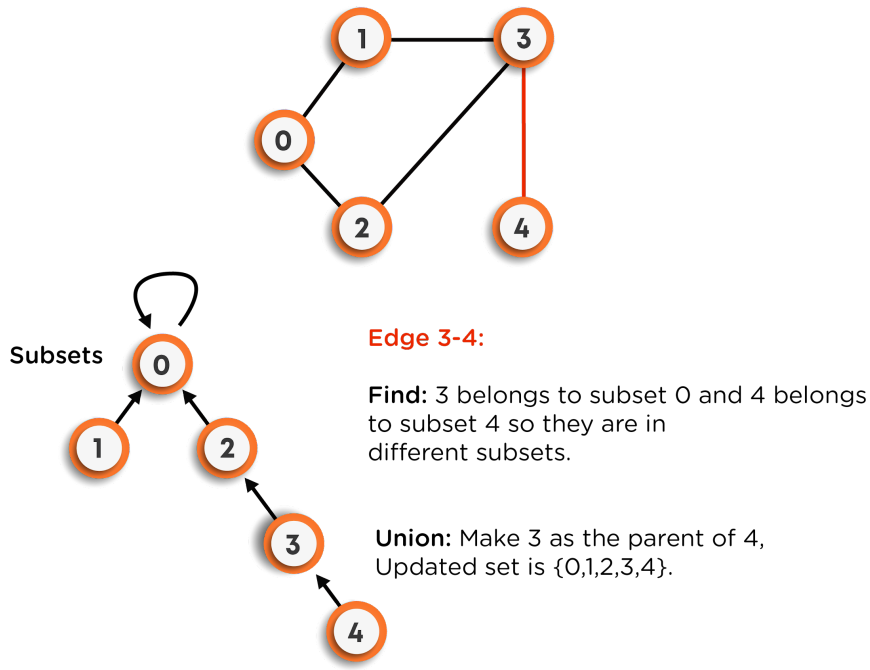




Edge 1-3:

Find: 1 belongs to subset 0 and 3 belongs to subset 3 so they are in different subsets.

Union: Make 1 as the parent of 3, Updated set is {0,1,2,3}. 0 is the set representative since 0 is parent for itself.



Note: While finding the parent of the vertex, we will be finding the topmost parent(Oldest Ancestor). For example: suppose, the vertex 0 and the vertex 1 were connected, where the parent of 0 is 1, and the parent of 1 is 1. Now, while determining the parent of the vertex 0, we will visit the parent array and check the vertex at index 0. In our case, it is 1. Now we will go to index 1 and check the parent of index 1, which is also 1. Hence, we can't go any further as the index is the parent of itself. This way, we will be determining the parent of any vertex.

The time complexity of the union-find algorithm becomes O(V) for each vertex in the worst case due to skewed-tree formation, where V is the number of vertices in the graph. Here, we can see that time complexity for cycle detection has significantly improved compared to the previous approach.

Kruskal's algorithm: Implementation

Till now, we have studied the logic behind Kruskal's algorithm for finding MST. Now, let's discuss how to implement it in code.

Consider the code below and follow the comments for a better understanding.

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

// Class that store values for each vertex
class Edge {
    public:
        int source;
        int dest;
        int weight;
};
// Comparator function used to sort edges
bool compare(Edge e1, Edge e2) {
    // Edges will sorted in order of their weights
    return e1.weight < e2.weight;
}
// Function to find the parent of a vertex
int findParent(int v, int *parent) {
    // Base case, when a vertex is parent of itself
    if (parent[v] == v) {
        return v;
    }
    // Recursively called to find the topmost parent of the vertex.
    return findParent(parent[v], parent);
}

void kruskals(Edge *input, int n, int E) {
    // In-built sort function: Sorts the edges in
    // increasing order of their weights
```

```
    sort(input, input + E, compare);

    // Array to store final edges of MST
    Edge *output = new Edge[n-1];
    // Parent array initialized with their indexes
    int *parent = new int[n];

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

    int count = 0;       //  To maintain the count of number of edges in
the MST
    int i = 0;                  // Index to traverse over the input array
    while (count != n - 1) {     // As the MST contains n-1 edges.
        Edge currentEdge = input[i];
        // Figuring out the parent of each edge's vertices
        int sourceParent = findParent(currentEdge.source, parent);
        int destParent = findParent(currentEdge.dest, parent);
        // If their parents are not equal, then we added that edge to
output
        if(sourceParent != destParent) {
            output[count] = currentEdge;
            count++;       // Increased the count
            parent[sourceParent] = destParent; // Updated the parent array
        }
        i++;
    }
    // Finally, printing the MST obtained.
    for (int i = 0; i < n-1; i++) {
        if(output[i].source < output[i].dest) {
            cout << output[i].source << " " << output[i].dest << " " <<
output[i].weight << endl;
        }
        else {
            cout<< output[i].dest << " " <<output[i].source << " " <<
output[i].weight << endl;
        }

    }
```

```
}

int main() {
    int n, E;
    cin >> n;
    cin >> E;

    Edge *input = new Edge[E];

    for (int i = 0; i < E; i++) {
        int s, d, w;
        cin >> s >> d >> w;
        input[i].source = s;
        input[i].dest = d;
        input[i].weight = w;
    }

    kruskals(input, n, E);
            return 0;
}
```

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n, and the total number of edges is E)

- Take input in the array of size E.
- Sort the input array on the basis of edge-weight. This step has the time complexity of O(E log(E)).
- Pick (n-1) edges and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle

detection, in the worst-case time complexity of E edges will be O(E.n), as discussed earlier.

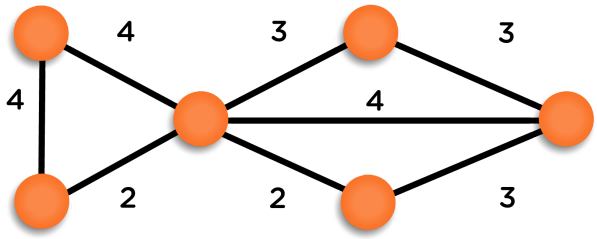
Hence, the total time complexity of Kruskal's algorithm becomes O(E log(E) + n.E). This time complexity is bad and needs to be improved. We can't reduce the time taken for sorting, but the time taken for cycle detection can be improved using another algorithm named **Union by Rank and Path Compression**. You need to explore this on yourselves. The basic idea in these algorithms is that we will be avoiding the formation of skewed-tree structure, which reduces the time complexity for each vertex to O(log(E)).

Prim's Algorithm

This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

In this algorithm, the MST is built by adding one edge at a time. In the beginning, the spanning tree consists of only one vertex, which is chosen arbitrarily from the set of all vertices of the graph. Then the minimum weighted edge, outgoing from this vertex, is selected and simultaneously inserted into the MST. Now, the tree contains two edges. Further, we will be selecting the edge with the minimum weight such that one end is already present there in the MST and the other one from the unselected set of vertices. This process is repeated until we have inserted a total of (n-1) edges in the MST.

Consider the following example for a better understanding.

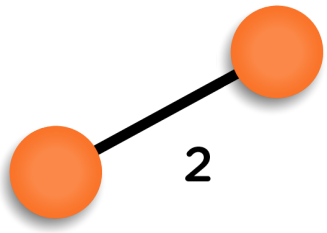


Step: 1

Start with a weighted graph

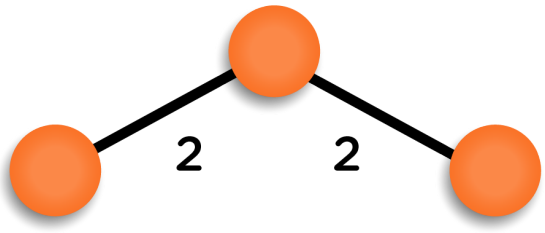
Step: 2

Choose a vertex



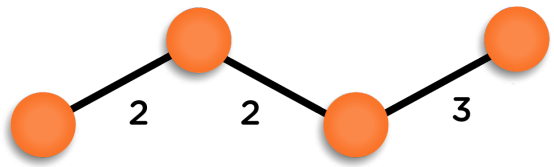
Step: 3

Choose the shortest edge from this vertex add it



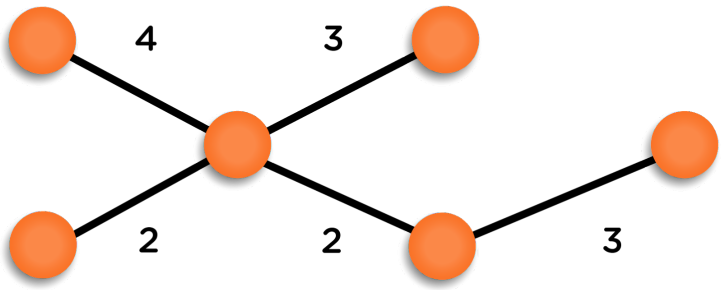
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Implementation:

- We are considering the starting vertex to be 0 with a parent equal to -1, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).
- The parent of all other vertices is assumed to be NIL, and the weight will be equal to infinity, which means that the vertex has not been visited yet.

- We will mark the vertex 0 as visited and rest as unvisited. If we add any vertex to the MST, then that vertex will be shifted from the unvisited section to the visited section.
- Now, we will update the weights of direct neighbors of vertex 0 with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from vertex 0.
- This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.

Let's look at the code now:

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

int findMinVertex(int *weights, bool *visited, int n) {
    // Initialized to -1 means there is no vertex till now
    int minVertex = -1;
    for (int  i = 0; i < n; i++) {
//Conditions: the vertex must be unvisited and either minVertex value is -1
//  or if minVertex has some vertex to it, then weight of currentvertex
// should be less than the weight of the minVertex.
        if (!visited[i] && (minVertex == -1 || weights[i] <
weights[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}

void prims(int **edges, int n) {

    int *parent = new int[n];
    int *weights = new int[n];
    bool *visited = new bool[n];
    // Initially, the visited array is assigned to false and weights array
//to infinity.
```

```
    for(int i = 0; i < n; i++) {
        visited[i] = false;
        weights[i] = INT_MAX;
    }
    // Values assigned to vertex 0.(the selected starting vertex to begin
with)
    parent[0] = -1;
    weights[0] = 0;

    for (int i = 0; i < n-1; i++) {
        // Find min vertex
        int minVertex = findMinVertex(weights, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++) {
            if(edges[minVertex][j] != 0 && !visited[j]) {
                if(edges[minVertex][j] < weights[j]) {
                    // updating weight array and parent array
                    weights[j] = edges[minVertex][j];
                    parent[j] = minVertex;
                }
            }
        }
    }
    // Final MST printed
    for (int i = 0; i < n; i++) {
        if (parent[i] < i) {
            cout << parent[i] << " " << i << " " <<weights[i] << endl;
        }
        else {
            cout << i << " " << parent[i] << " " << weights[i] << endl;
        }
    }
}

int main() {

    int n;
    int e;
    cin >> n >> e;
```

```
    int **edges = new int*[n]; // Adjacency matrix used to store the graph
    for (int i = 0; i < n; i++) {
        edges[i] = new int[n];
        for (int j = 0; j < n; j++) {
            // Initially all pairs are assigned 0 weight which
            // means that there is no edge between them
            edges[i][j] = 0;
        }
    }

    for (int i = 0; i < e; i++) {
        int f, s, weight;
        cin >> f >> s >> weight;
        edges[f][s] = weight;
        edges[s][f] = weight;
    }

    prims(edges, n);

    for(int i = 0; i < n; i++) {
        delete [] edges[i];
    }
    delete [] edges;
    return 0;
}
```

Time Complexity of Prim's Algorithm:

Here, n is the number of vertices, and E is the number of edges.

- The time complexity for finding the minimum weighted vertex is O(n) for each iteration. So for (n-1) edges, it becomes $O(n^2)$.
- Similarly, for exploring the neighbor vertices, the time taken is $O(n^2)$.

It means the time complexity of Prim's algorithm is $O(n^2)$. We can improve this in the following ways:

- For exploring neighbors, we are required to visit each and every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.
- Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of $O(n^2)$. Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a **priority queue** where the priority will be taken as weights of the vertices. This will take O(log(n)) time complexity to remove a vertex from the priority queue.

These optimizations can lead us to the time complexity of O((n+E)log(n)), which is much better than the earlier one. Try to write the optimized code by yourself.

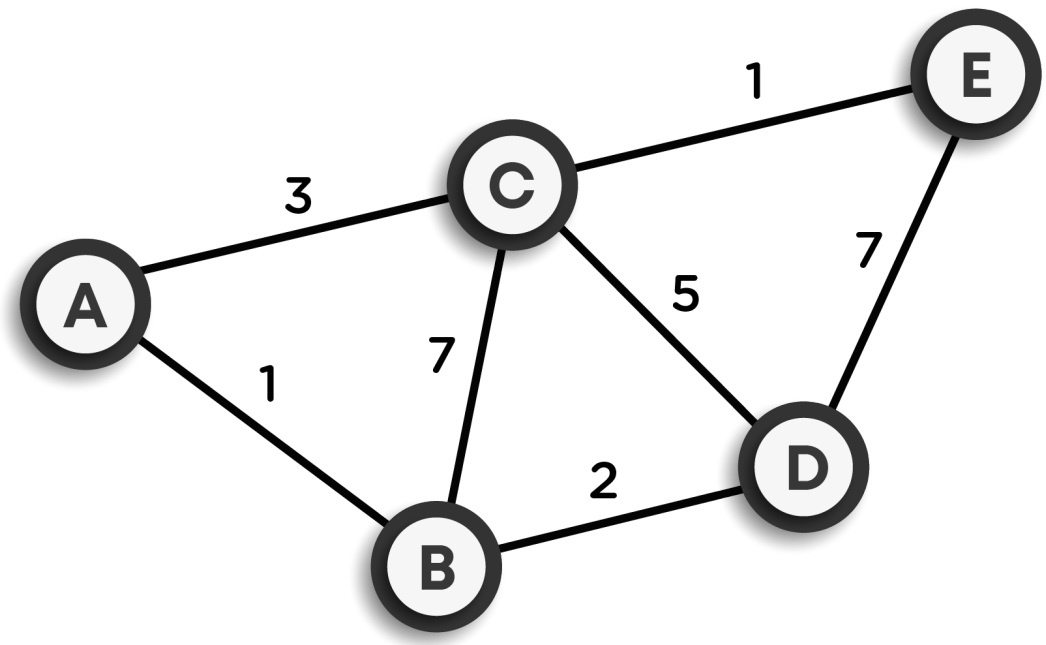
Dijkstra's Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular source vertex.

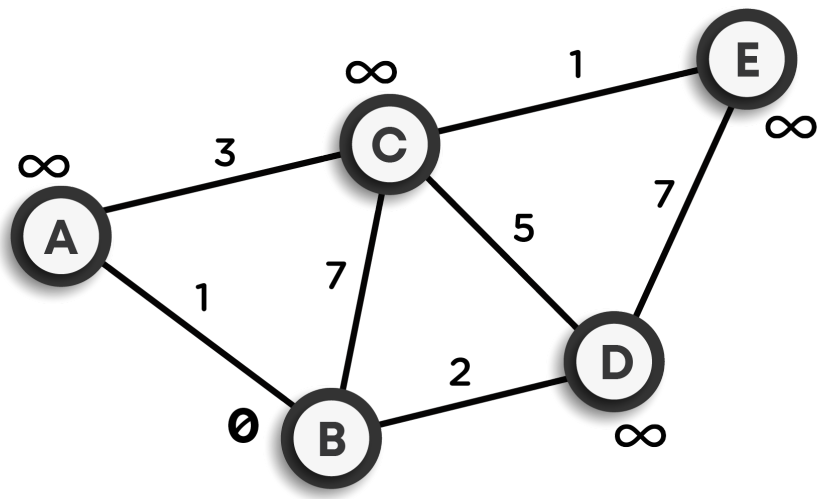
Let's consider the algorithm with an example:

1. We want to calculate the shortest path between the source vertex C and all other vertices in the following graph.

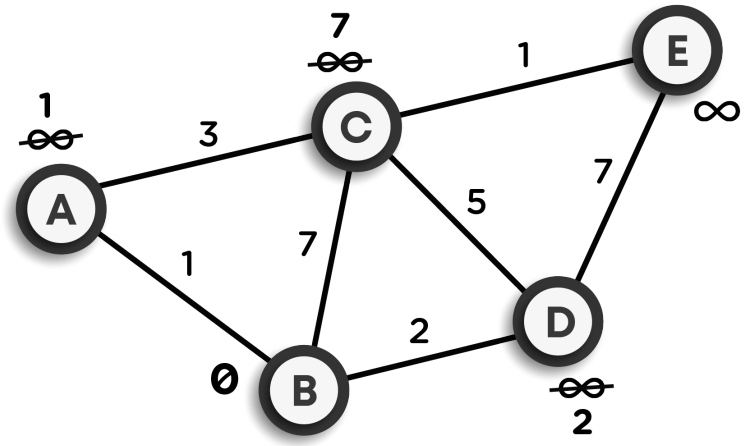


2. While executing the algorithm, we will mark every node with its **minimum distance** to the selected node, which is C in our case. Obviously, for node C

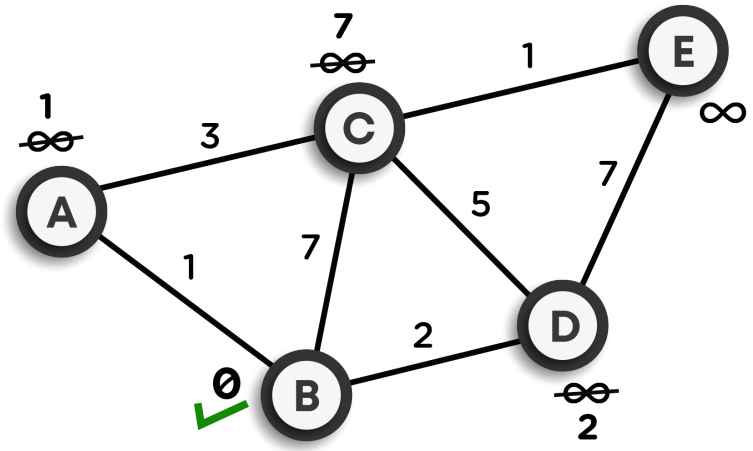
itself, this distance will be 0, and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.



3. Now, we will check for the neighbors of the current node, which in our case is A, B, and D. Now, we will add the minimum cost of the current node to the weight of the edge connecting the current node and the particular neighbor node. For example, for node B, it's weight will become minimum(infinity, 0+7) = 7. This same process is repeated for other neighbor nodes.

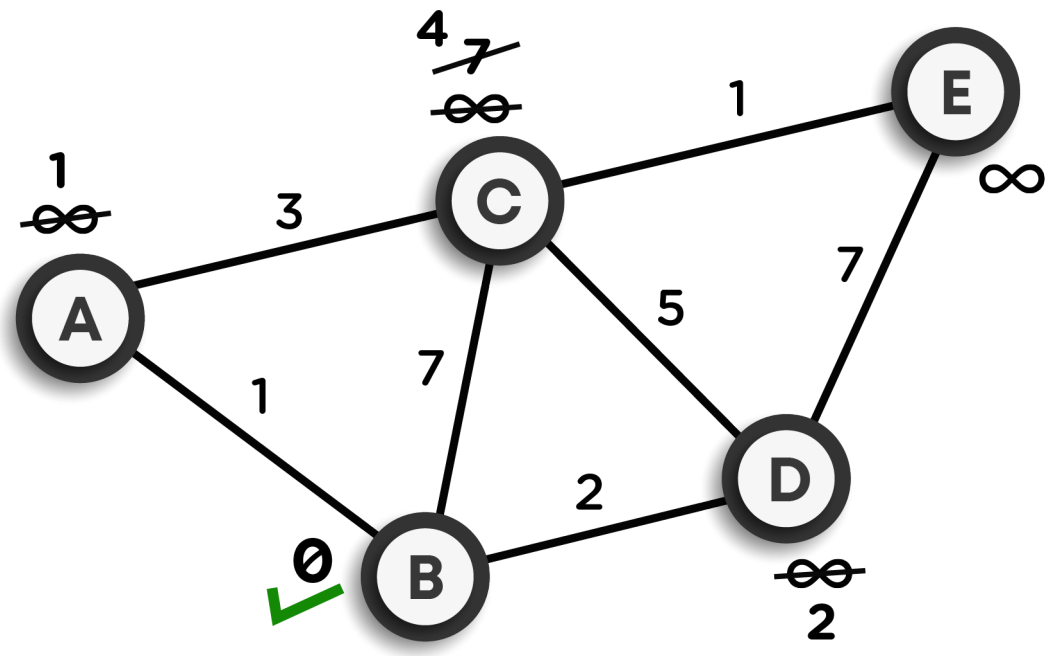


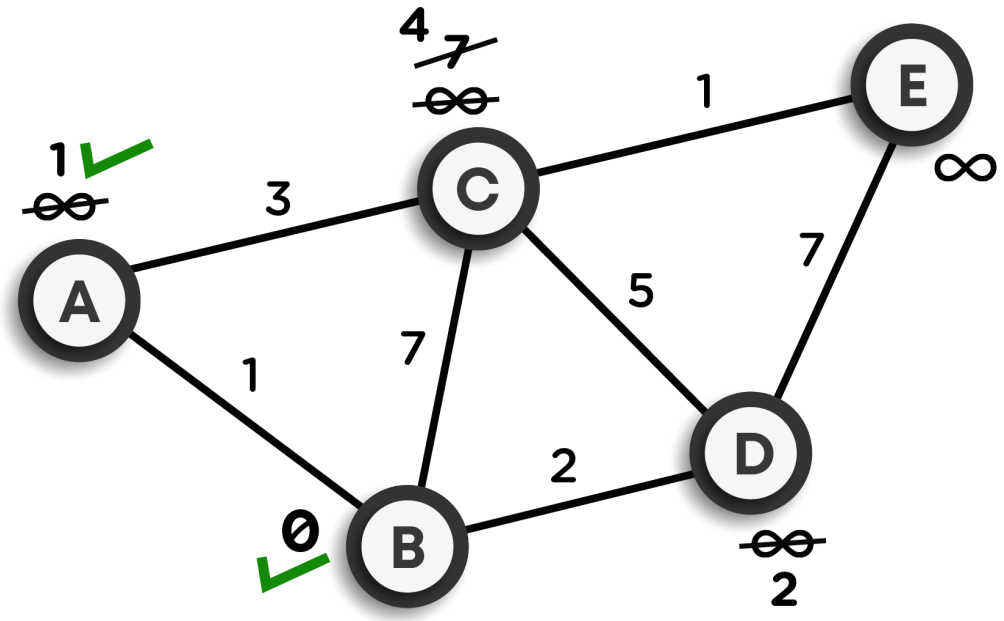
4. Now, as we have updated the distance of all the neighbor nodes of the current node, we will mark the current node as visited.

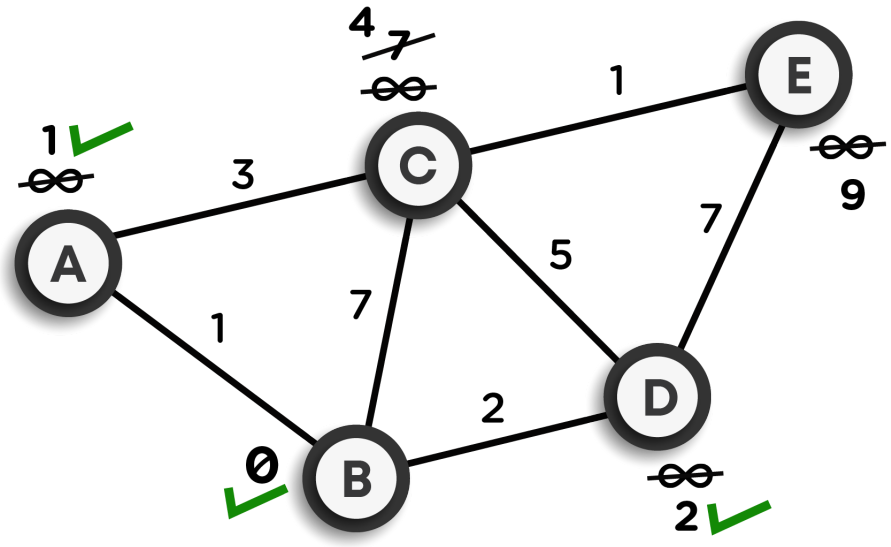


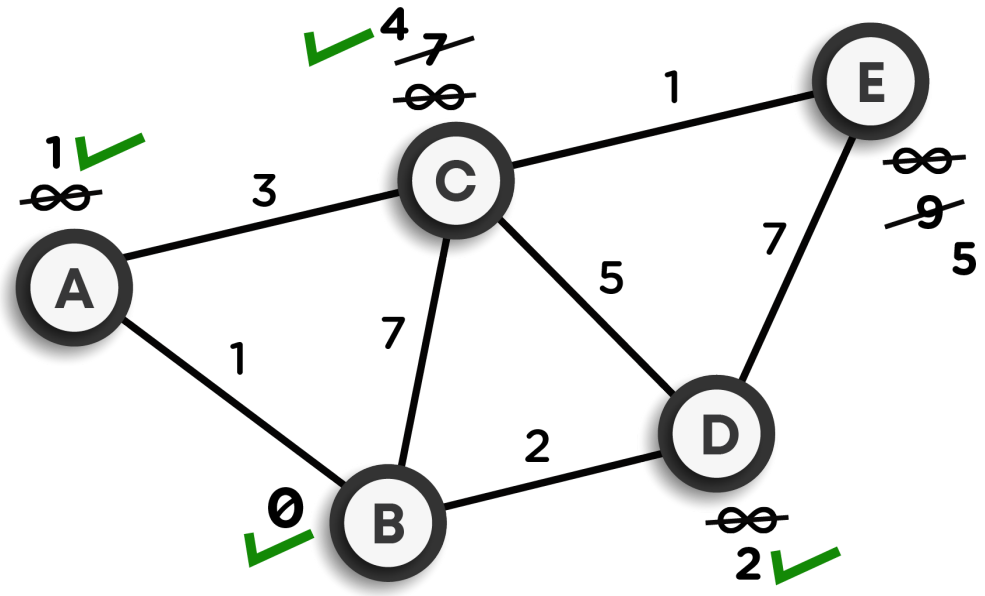
5. After this, we will be selecting the minimum weighted node among the remaining vertices. In this case, it is node A. Take this node as the current node.

6. Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:

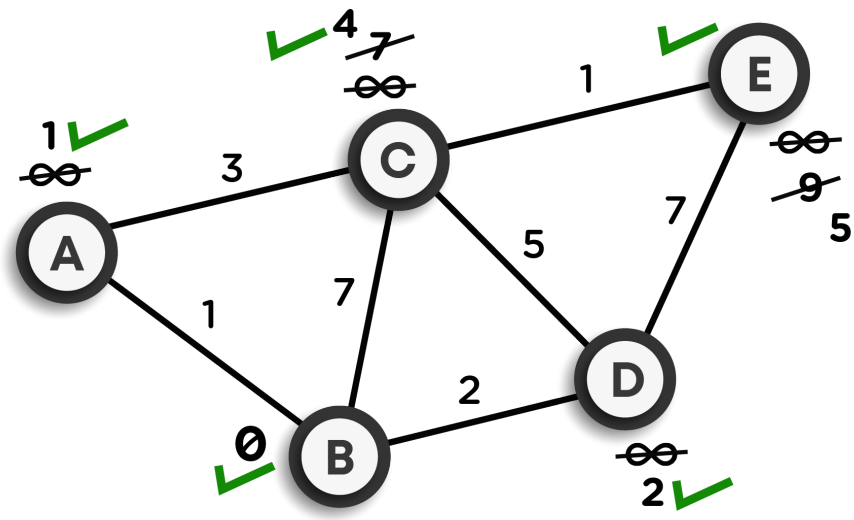








7. Finally, we will get the graph as follows:



The distances finally marked at each node are minimum from node C.

Implementation:

Let's look at the code below for a better explanation:(Code is nearly same as that of Prim's algorithm, just a change while updating the distance)

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

int findMinVertex(int *distance, bool *visited, int n) {

    int minVertex = -1;
    for (int  i = 0; i < n; i++) {
        if (!visited[i] && (minVertex == -1 || distance[i] <
distance[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}
```

```
void dijkstra(int **edges, int n) {

    int *distance = new int[n];
    bool *visited = new bool[n];

    for(int i = 0; i < n; i++) {
        visited[i] = false;
        distance[i] = INT_MAX;
    }

    distance[0] = 0;  // 0 is considered as the starting node.

    for (int i = 0; i < n-1; i++) {
        // Find min vertex
        int minVertex = findMinVertex(distance, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++) {
            if(edges[minVertex][j] != 0 && !visited[j]) {
                    // distance of any node will be the current node's
distance + the weight
                    // of the edge between them
                int dist = distance[minVertex] + edges[minVertex][j];
                if(dist < distance[j]) {     // If required, then updated.
                    distance[j] = dist;
                }
            }
        }
    }
    // Final output of distance of each node with respect to 0
    for (int i = 0; i < n; i++) {
        cout << i << " " << distance[i] << endl;
    }
}

int main() {

    int n;
    int e;
```

```
    cin >> n >> e;
    int **edges = new int*[n];
    for (int i = 0; i < n; i++) {
        edges[i] = new int[n];
        for (int j = 0; j < n; j++) {
            edges[i][j] = 0;
        }
    }

    for (int i = 0; i < e; i++) {
        int f, s, weight;
        cin >> f >> s >> weight;
        edges[f][s] = weight;
        edges[s][f] = weight;
    }

    dijkstra(edges, n);

    for(int i = 0; i < n; i++) {
        delete [] edges[i];
    }
    delete [] edges;
    return 0;
}
```

Time Complexity of Dijkstra's algorithm:

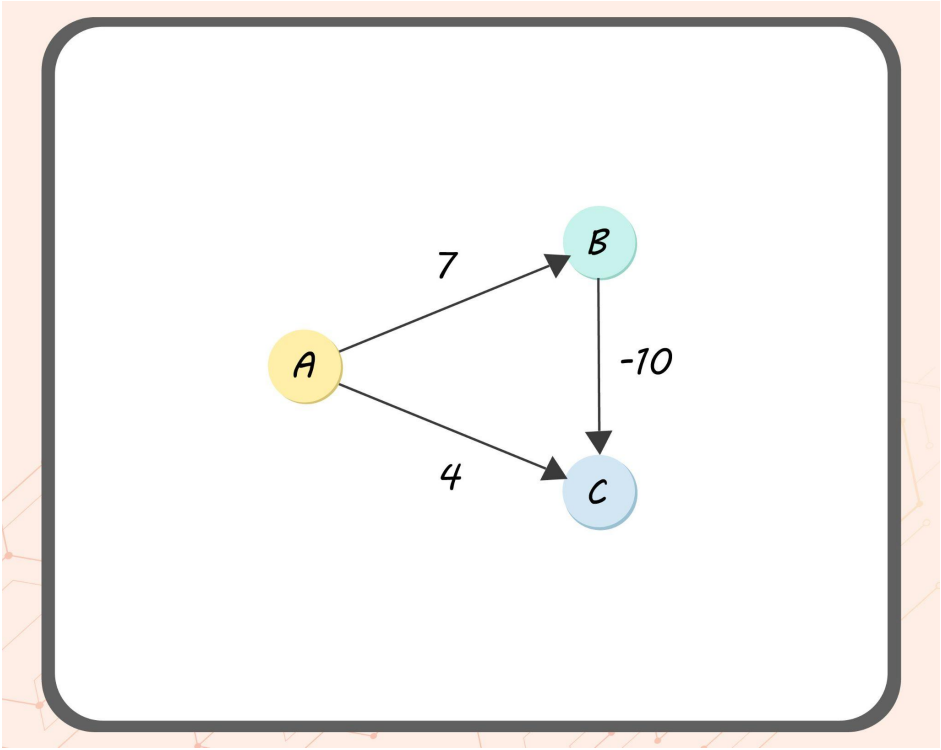
The time complexity is also the same as that of Prim's algorithm, i.e., $O(n^2)$. This can be reduced by using the same approaches as discussed in Prim's algorithm's content.

Bellman Ford Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

It overcomes the shortcomings of Dijkstra's algorithm, which fails for a graph with negative weight.

Considering an example :



If we apply Dijkstra's algorithm in the above example it gives us the path A → C.

However in reality the path A → B → C is the correct answer with the distance of 7 - 10 = -3.

So Bellman Ford suggests that in order to take into account the negative edges, we must calculate the shortest path between any two vertices through different numbers of edges so as to find out the correct answer amongst them.

In the above example, if we find the shortest path with at most one edge then our distance is 4. But if we find the shortest path once more and this time we could take at most 2 edges to travel from our source vertex to the destination vertex then our total distance is -3. So amongst 4 and -3, -3 is shorter and hence our final path will be A→ B→ C.

Algorithm:

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

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

a) Do following for each edge u-v

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

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

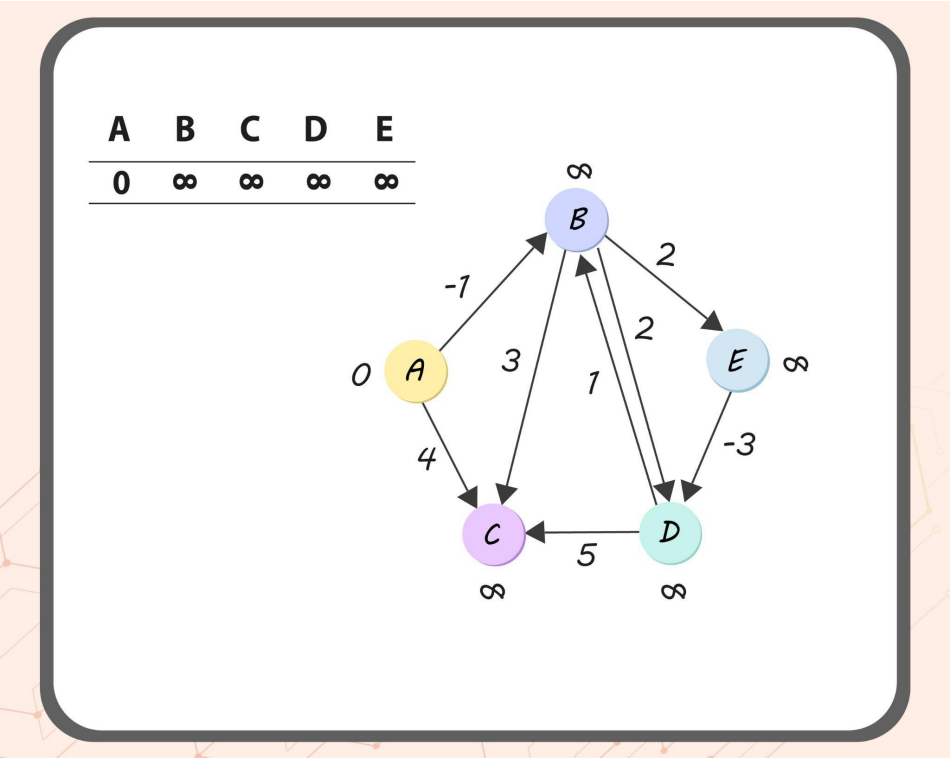
3) This step reports if there is a negative weight cycle in the graph. Do following for each edge u-v

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

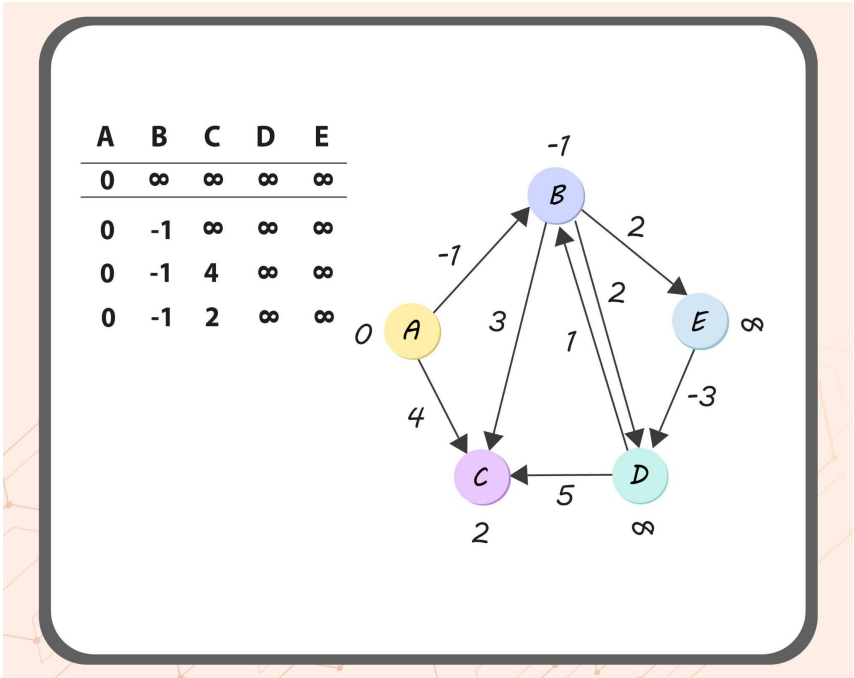
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one

more time and get a shorter path for any vertex, then there is a negative weight cycle.

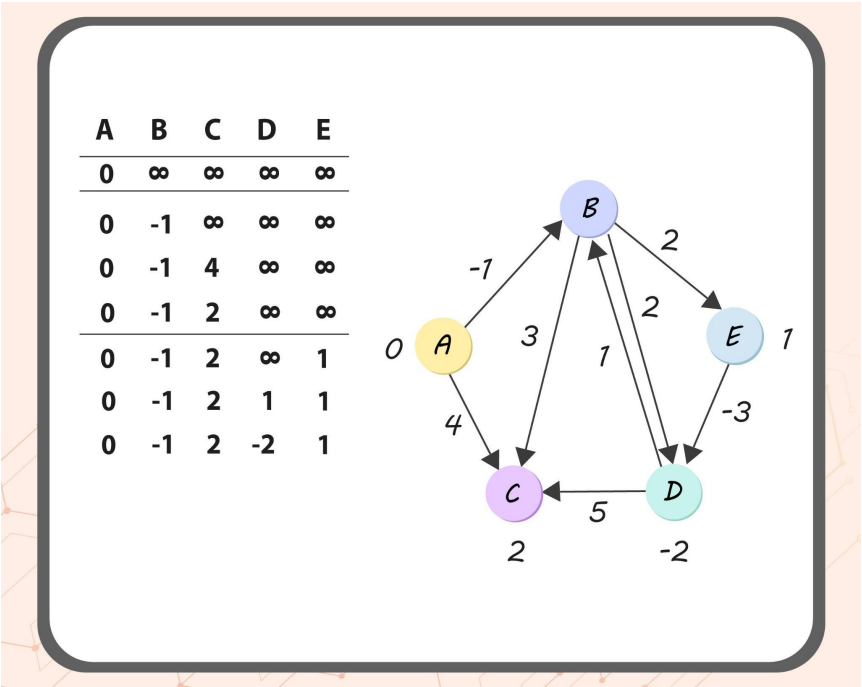
Let's take a look at another example and see how this works step by step.



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).



The first iteration guarantees to give all shortest paths which are at most 1 edge long.



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

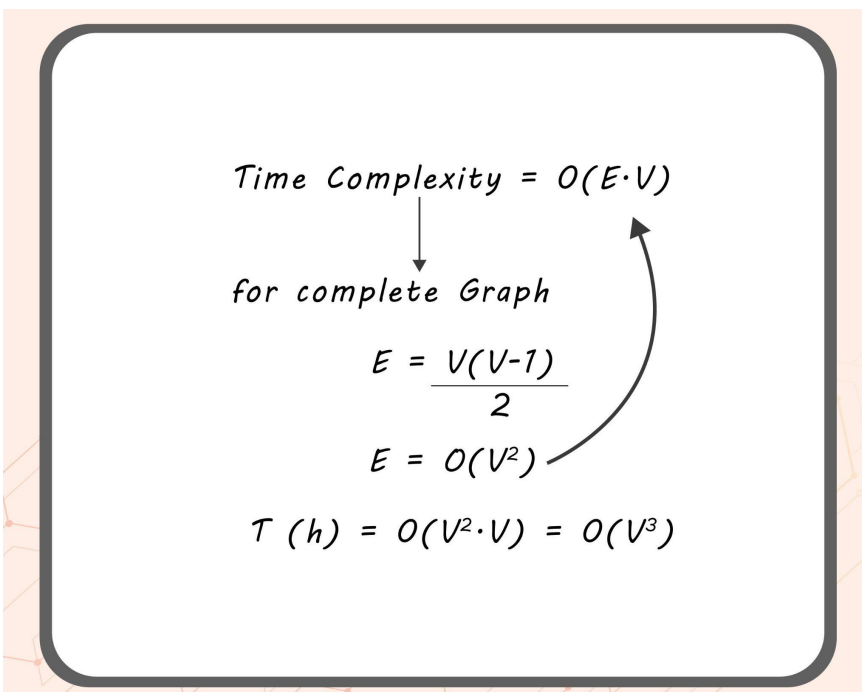
Implementation:

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

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void printAns(int dist[], int n)
{
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
```

```
    int E = graph->E;
    int dist[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return; // If negative cycle is detected, simply return
        }
    }

    printAns(dist, V);
    return;
}

int main()
{
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

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

```
    for (int i = 0; i < E; i++)
    {
        int u, v, w;
        cout << "Enter the u, v and w(weight) for " << i + 1 << "th edge:
";
        cin >> u >> v >> w;
        graph->edge[i].src = u;
        graph->edge[i].dest = v;
        graph->edge[i].weight = w;
    }
    BellmanFord(graph, 0);
    return 0;
}
```

Time Complexity of Bellman Ford algorithm:

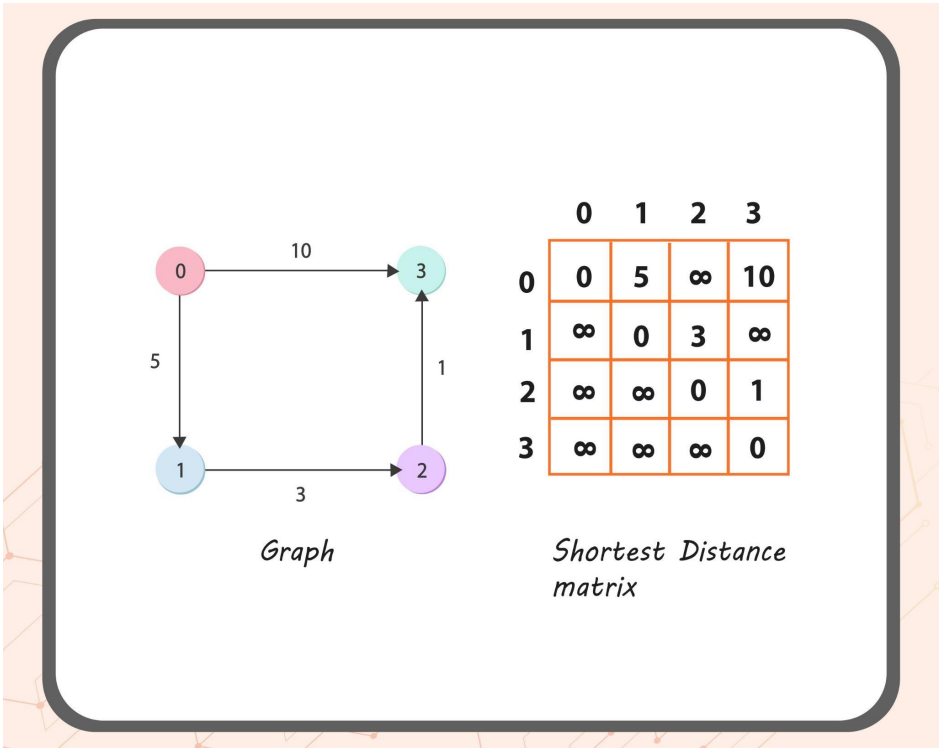


Floyd Warshall Algorithm

This algorithm is used to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

We store the distances in a matrix of N X N in which if the value at any cell [i][j] is infinity, then that means there exists no path between vertex i and vertex j.

Let's look at an example for better understanding:



	0	1	2	3
0	0	5	∞	10
1	∞	0	3	∞
2	∞	∞	0	1
3	∞	∞	∞	0

The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered

vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

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

Code :

```
/*
Time complexity: O(N^3)
Space complexity: O(N^2)
where N is the number of vertex in the graph
*/
#include <bits/stdc++.h>
using namespace std;
int INF = 1e9;
int main(){
    int n, m;
    cout << "Enter the number of vertices: ";
    cin >> n;
    cout << "Enter the number of edges: ";
    cin >> m;
    int mat[n][n];
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            mat[i][j] = INF;
            if (i == j){
                mat[i][j] = 0;
            }
        }
    }
    for (int i = 0; i < m; i++){
        int a, b, c;cout << "Enter the u, v and w(weight) for " << i + 1 <<
"th edge: ";
        cin >> a >> b >> c;
        mat[a][b] = c;
```

```
    }
    int i, j, k;
    for (k = 0; k < n; k++){
        for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                if (mat[i][j] > (mat[i][k] + mat[k][j]) && (mat[k][j] !=
INF && mat[i][k] != INF))
                mat[i][j] = mat[i][k] + mat[k][j];
            }
        }
    }
    cout << "\nFinal Matrix A" << n << ": " << endl;
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            cout << mat[i][j] << " ";
    }
    cout << endl;
    }
}
```

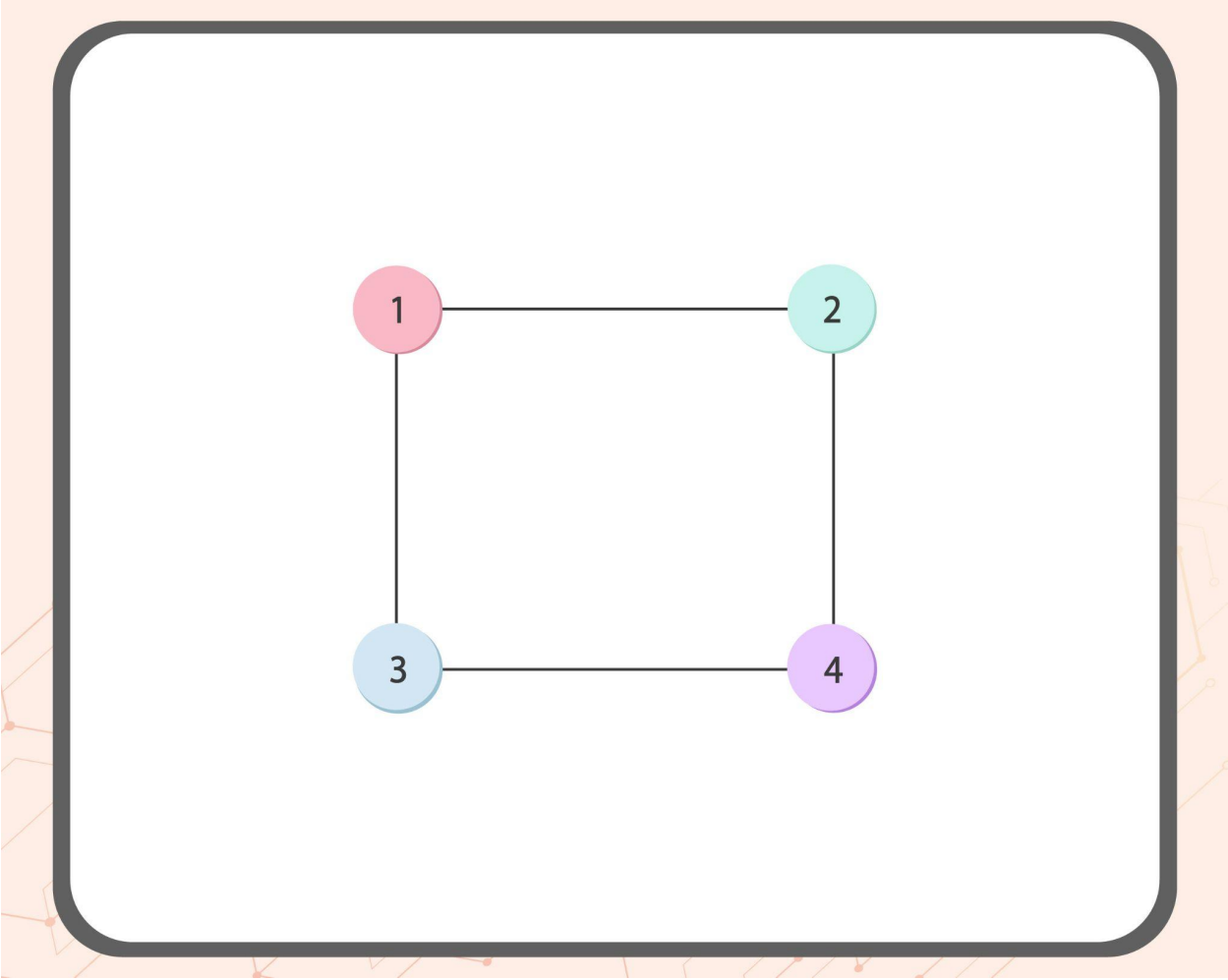
Advance Graphs

Connected Components

Problem Statement: We have to find out the number of connected components in an undirected graph.

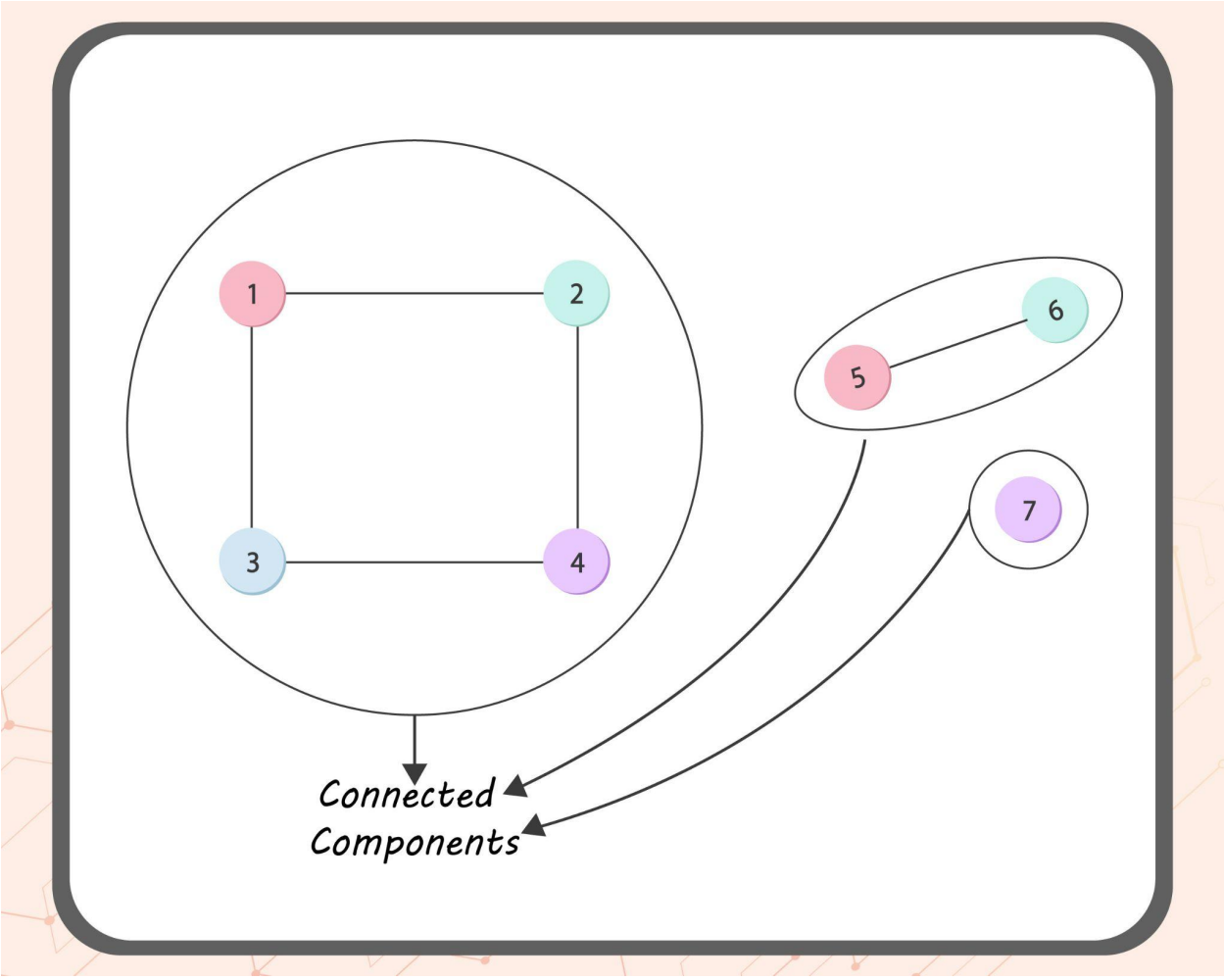
Explanation:

Let us consider an undirected graph



In the above graph there is a path from every vertex to every other vertex, therefore this is a connected graph. In this case we say that the graph has only one connected component.

There may be graphs which do not satisfy the above conditions and are called disconnected graphs. However in such cases, there may still be several components which are connected within themselves. For example,



In the above figure vertices 1, 2, 3, 4 form a connected component. Another connected component is formed by vertices 5, 6 and vertex 7 itself is a connected component.

To find the connected components in an undirected graph, we can use either DFS or BFS for this task.

Let's take a look at the algorithm that uses DFS to solve this problem:

```
Data: Given an undirected graph G (V, E)
Result: Number of Connected Components
Component_Count = 0;
for each vertex k ϵ V do
   if Visited[k] == False;
end
for each vertex k ϵ V do
   if Visited[k] == False  then
      DFS (V, k);
      Component_Count = Component_Count + 1;
   end
end
Print Component_Count;
Procedure DFS (V, k)
Visited[k] = True;
for each Vertex P ϵ V.Adj[k] do
   if Visited[p] == False  then
      DFS (V, p);
   end
end
```

We have already taken a look at how DFS and BFS work in the previous lectures along with their codes.

Permutations Swap

Problem Statement:

Kevin has a permutation P of N integers 1, 2, ..., N, but he doesn't like it. Kevin wants to get a permutation Q.

He also believes that there are M good pairs of integers (ai, bi). Kevin can perform following operation with his permutation:

Swap Px and Py only if (x, y) is a good pair.

Help him and tell if Kevin can obtain permutation Q using such operations.

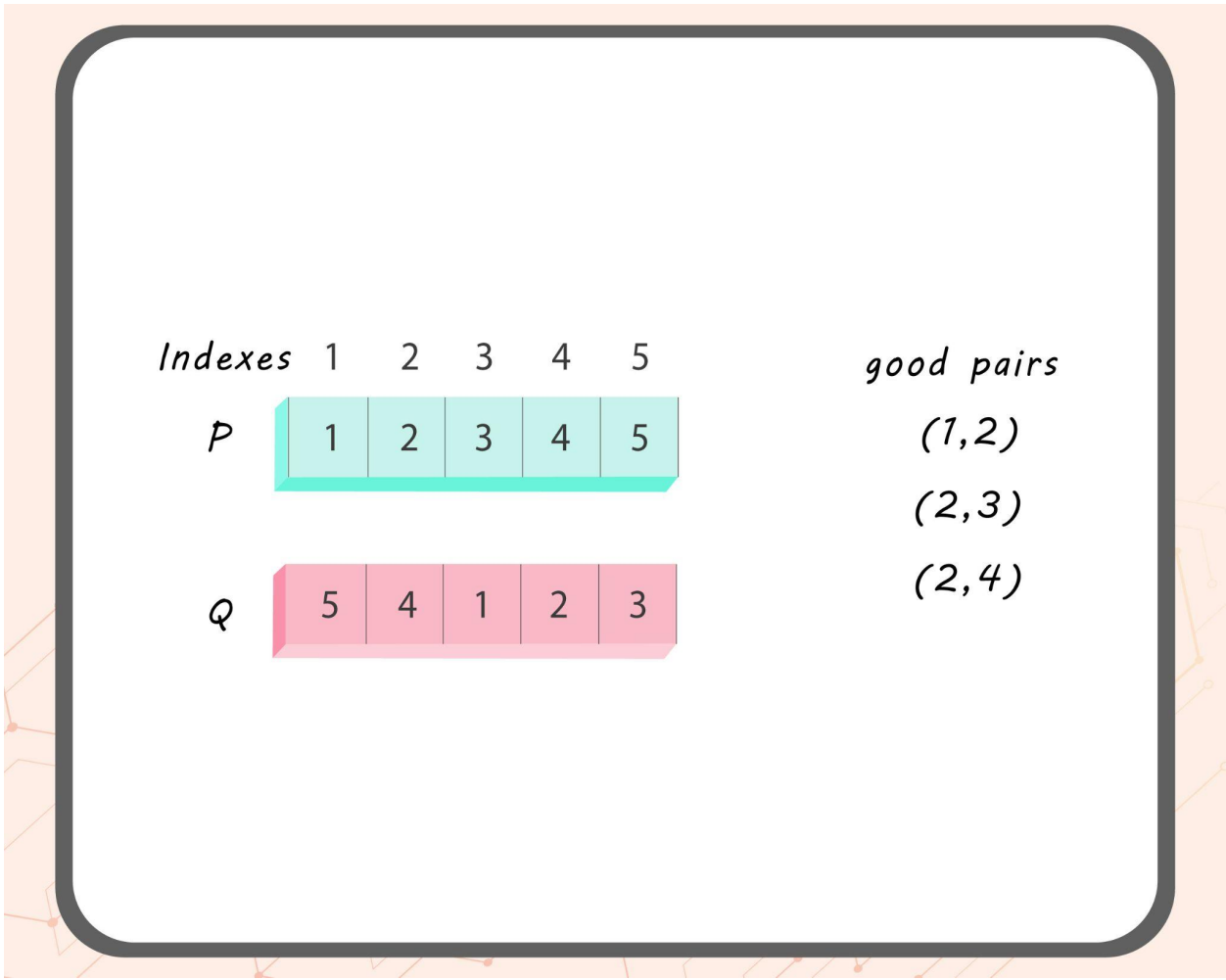
Explanation:

Let's take an example to better understand the problem statement.

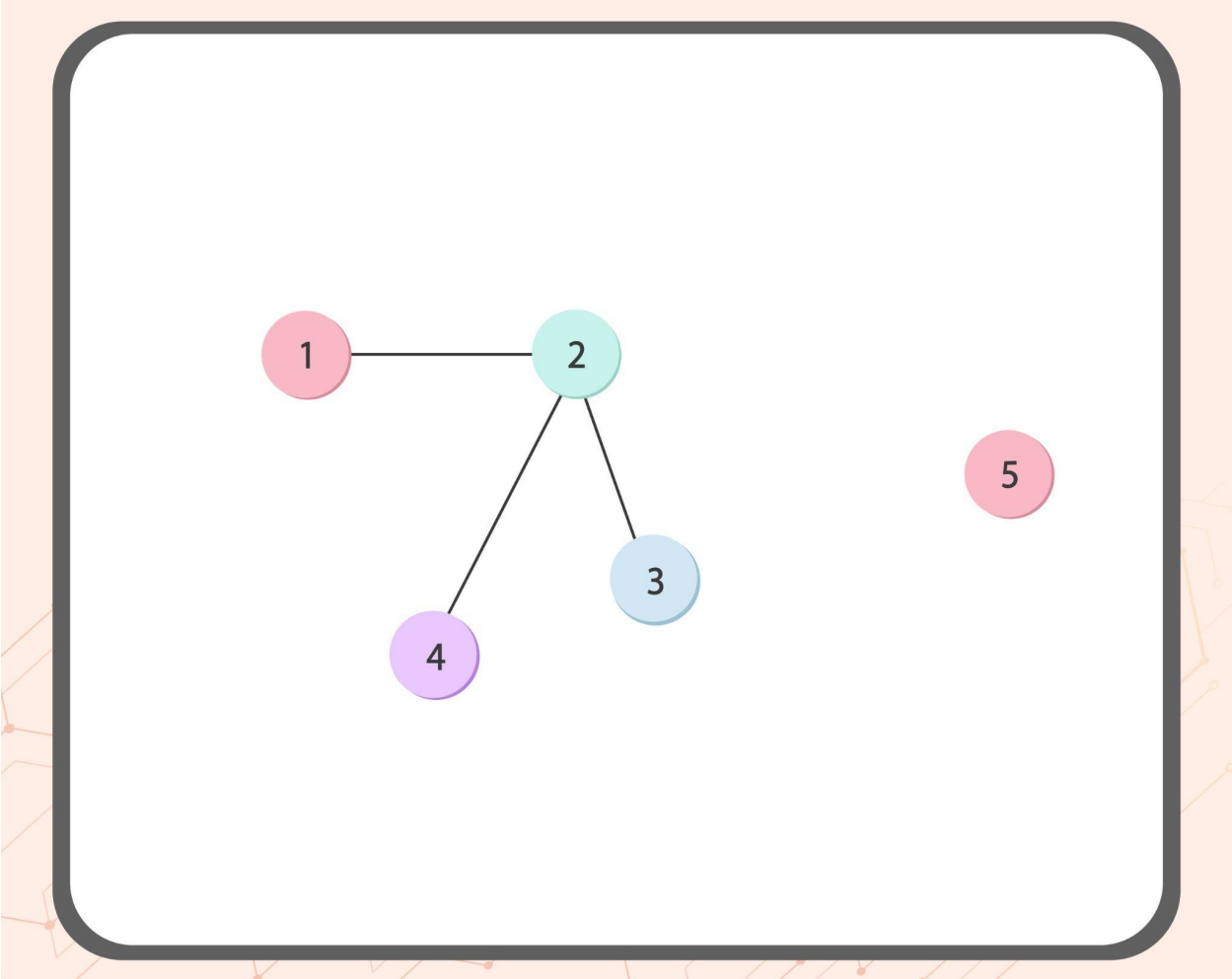
We are given N= 5 numbers **{1, 2, 3, 4, 5}** and we also have permutation **P = {1, 2, 3, 4, 5}** and permutation **Q = {5, 4, 1, 2, 3}** of these N numbers.

We are also given some pairs of numbers **{(1, 2), (2, 3), (2, 4)}**. Each number in a pair represents that the two indexes or positions can be switched with each other.

We need to tell whether permutation P can be converted to permutation Q.



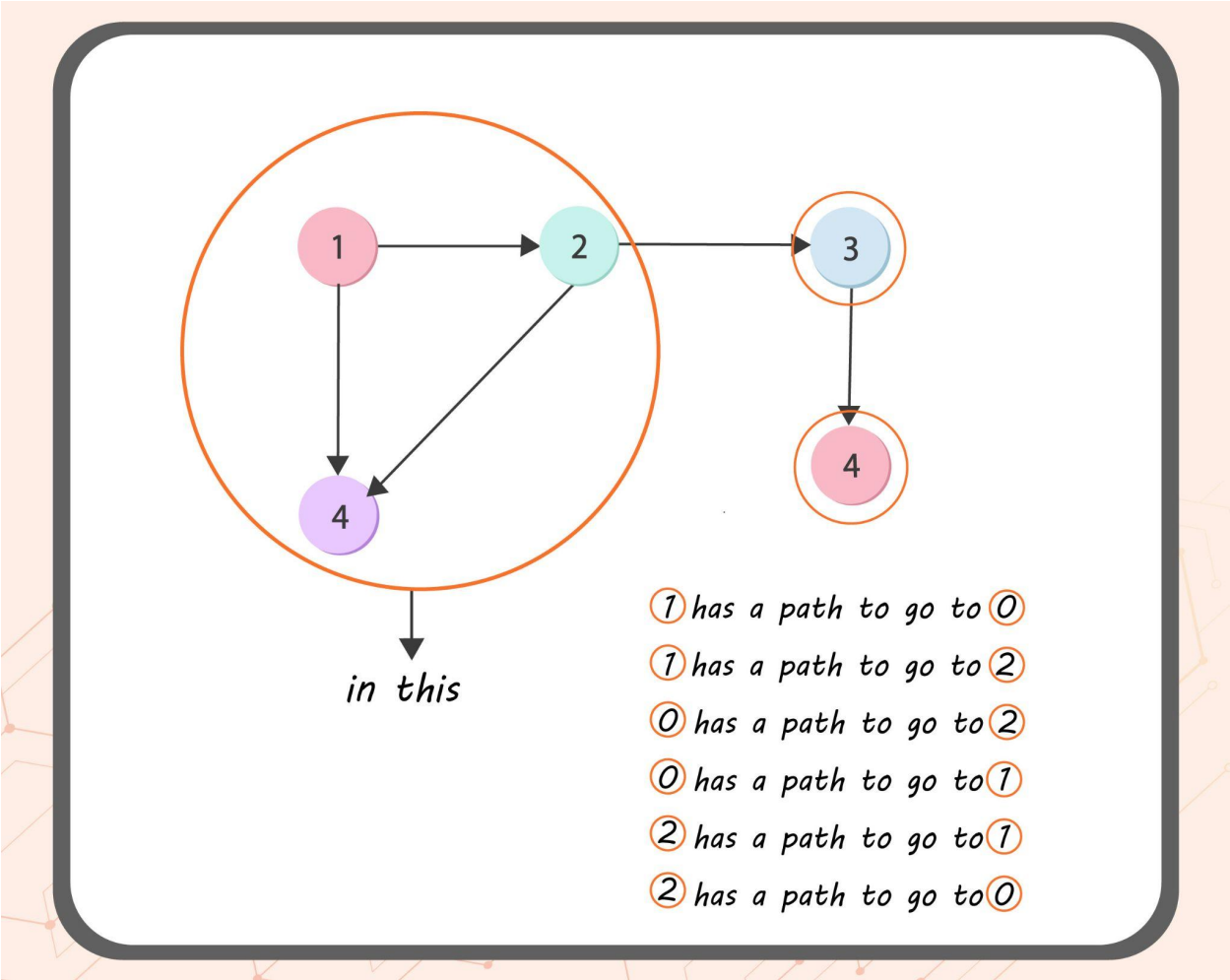
This problem can be approached using the concept of hasPath. This basically means that if we are given a pair (1, 2), then these can be considered as the vertices in a graph and since they are a pair that means there exists an edge between them. SImilarly there exists an edge between (2, 3) and (2, 4). So the corresponding graph for the above problem can be represented as the following figure:



Now, even if we can't directly perform some operations we can indirectly do them. For example, if we have to swap indexes 1 and 3 then they are not in our given pairs but what we could do is first perform swap on index 1 and 2 and then perform swap on index 2 and 3. So overall there needs to exist a path between the indexes (vertices) for swap operation.

Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a **directed graph** is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.

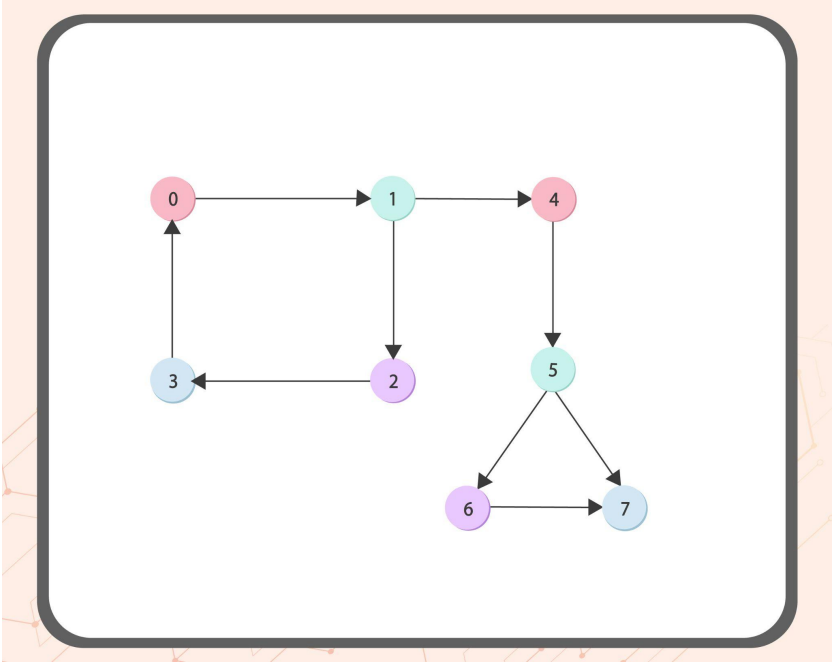


Kosaraju's Algorithm

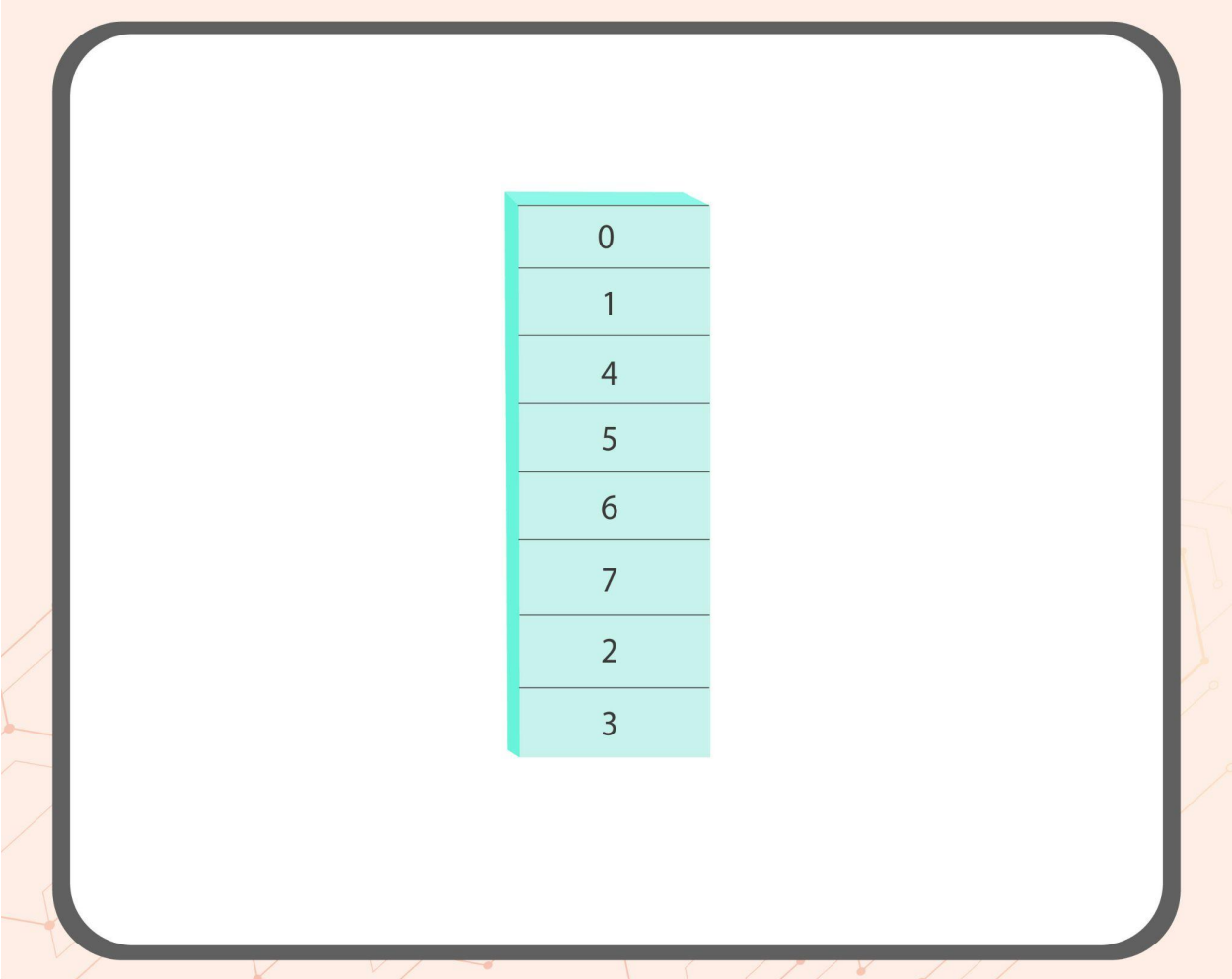
Kosaraju's algorithm is used to find strongly connected components in a graph. The algorithm consists of three steps:

1. **Do a DFS on the original graph**: Do a DFS on the original graph, keeping track of the finish times of each node. This can be done using a stack, when a DFS finishes put the source vertex on the stack. This way the node with the highest finishing time will be on top of the stack.
2. **Reverse the original graph**: Reverse the graph using an adjacency list.
3. **Do DFS again**: Do DFS on the reversed graph, with the source vertex as the vertex on top of the stack. When DFS finishes, all nodes visited will form one Strongly Connected Component. If any more nodes remain unvisited, this means there are more Strongly Connected Components, so pop vertices from top of the stack until a valid unvisited node is found. This will have the highest finishing time of all currently unvisited nodes. This step is repeated until all nodes are visited.

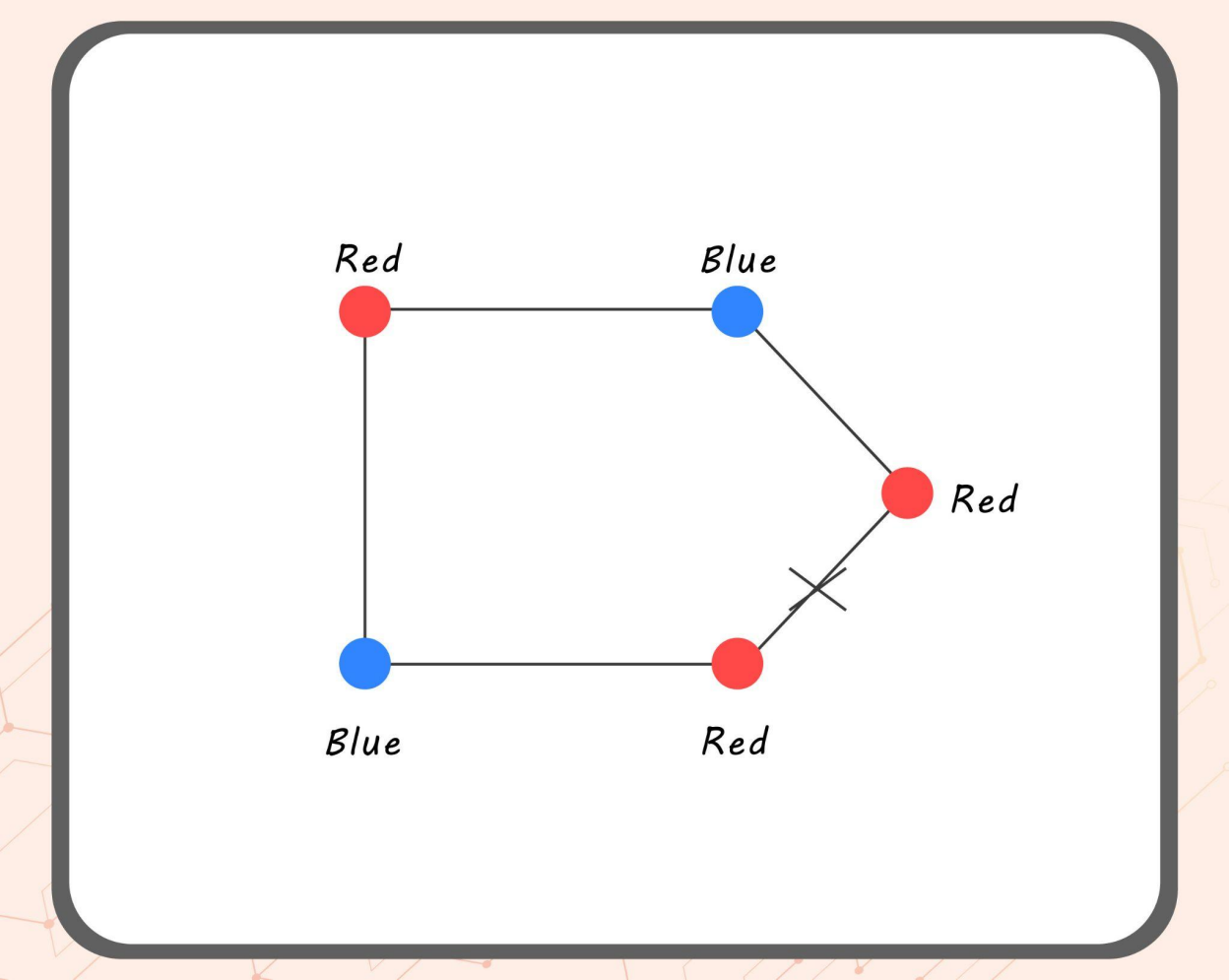
Let us take a look at an example to see the working of the above algorithm:



1. After applying DFS on this graph starting from node 0, and keeping track of the finishing time of each node with the help of the stack, our stack becomes



2. Now reversing the original graph gives us



3. Pop every element out of the stack. If it's not visited, apply DFS on it and that will give us the SCC.

   Step 1. We pop an element at the top that is node 0. We apply DFS(0) , hence it will visit node 1,2,3. Now we mark them as visited and this forms our first group.

   Step 2. We pop the second element which is 1 but since it is visited we skip this.

   Step 3. We pop the third element which is 4. Since it is not visited we apply DFS(4). Now it will visit only 4 as there are no adjacent nodes. Hence 4 is a separate group.

Step 4. We pop the fourth element which is 5. Since it is not visited we apply DFS(5). It will visit 5,6,7. Hence 5,6,7 will form a separate group.

Step 5. We pop the fifth element which is 6 , since it is already visited we skip. Now we pop the sixth element which is 7 , since it is already visited we skip it. Similarly we skip seventh and eight elements which are 2 and 3 as they are already visited.

Now , the stack is empty and our algorithm stops.

The nodes printed are:

0 3 2 1

4

5 7 6

Code:

```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_set>
using namespace std;

void dfs(vector<int>* edges, int start, unordered_set<int> &visited,
stack<int> &finishStack) {
      visited.insert(start);
      for (int i = 0; i < edges[start].size(); i++) {
            int adjacent = edges[start][i];
            if (visited.count(adjacent) == 0) {
                  dfs(edges, adjacent, visited, finishStack);
            }
      }
      finishStack.push(start);
}

void dfs2(vector<int>* edges, int start, unordered_set<int>* component,
unordered_set<int> & visited) {
```

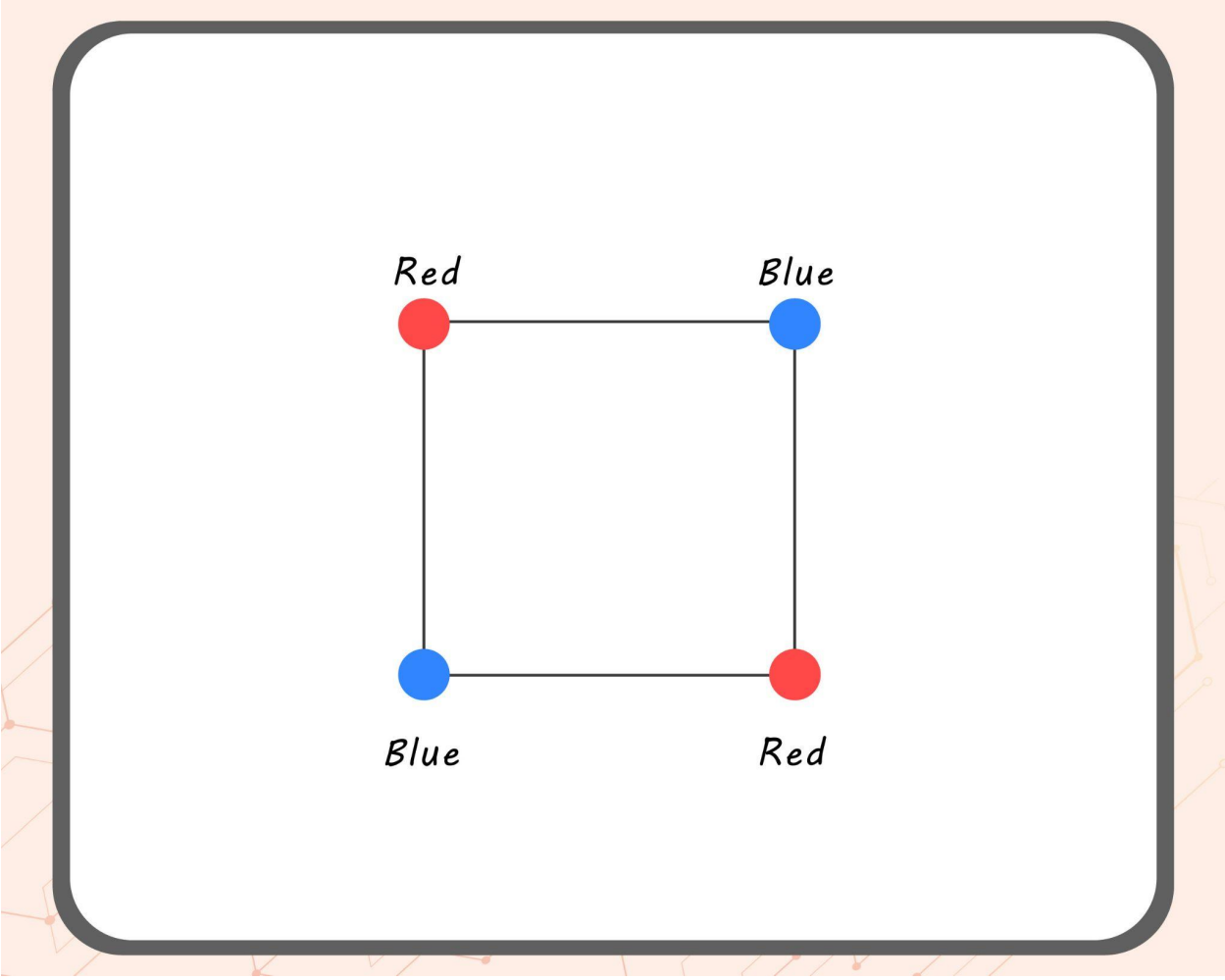
```
	visited.insert(start);
	component->insert(start);
	for (int i = 0; i < edges[start].size(); i++) {
		int adjacent = edges[start][i];
		if (visited.count(adjacent) == 0) {
			dfs2(edges, adjacent, component, visited);
		}
	}
}

unordered_set<unordered_set<int>*>* getSCC(vector<int>* edges, vector<int>*
edgesT, int n) {
	unordered_set<int> visited;
	stack<int> finishedVertices;
	for (int i = 0; i < n; i++) {
		if (visited.count(i) == 0) {
			dfs(edges, i, visited, finishedVertices);
		}
	}
	unordered_set<unordered_set<int>*>* output = new
unordered_set<unordered_set<int>*>();
	visited.clear();
	while (finishedVertices.size() != 0) {
		int element = finishedVertices.top();
		finishedVertices.pop();
		if (visited.count(element) != 0) {
			continue;
		}
		unordered_set<int>* component = new unordered_set<int>();
		dfs2(edgesT, element, component, visited);
		output->insert(component);
	}
	return output;
}

int main() {
	int n;
	cin >> n;
	vector<int>* edges = new vector<int>[n];
	vector<int>* edgesT = new vector<int>[n];
```

```
	int m;
	cin >> m;
	for (int i = 0; i < m; i++) {
		int j, k;
		cin >> j >> k;
		edges[j - 1].push_back(k - 1);
		edgesT[k - 1].push_back(j - 1);
	}
	unordered_set<unordered_set<int>*>* components = getSCC(edges,
edgesT, n);
	unordered_set<unordered_set<int>*>::iterator it =
components->begin();
	while (it != components->end()) {
		unordered_set<int>* component = *it;
		unordered_set<int>::iterator it2 = component->begin();
		while (it2 != component->end()) {
			cout << *it2  + 1<< " ";
			it2++;
		}
		cout << endl;
		delete component;
		it++;
	}
	delete components;
	delete [] edges;
	delete [] edgesT;
}
```

Bipartite Graphs

A bipartite graph is a graph whose vertices can be divided into two disjoint sets so that every edge connects two vertices from different sets (i.e. there are no edges which connect vertices from the same set). These sets are usually called sides.

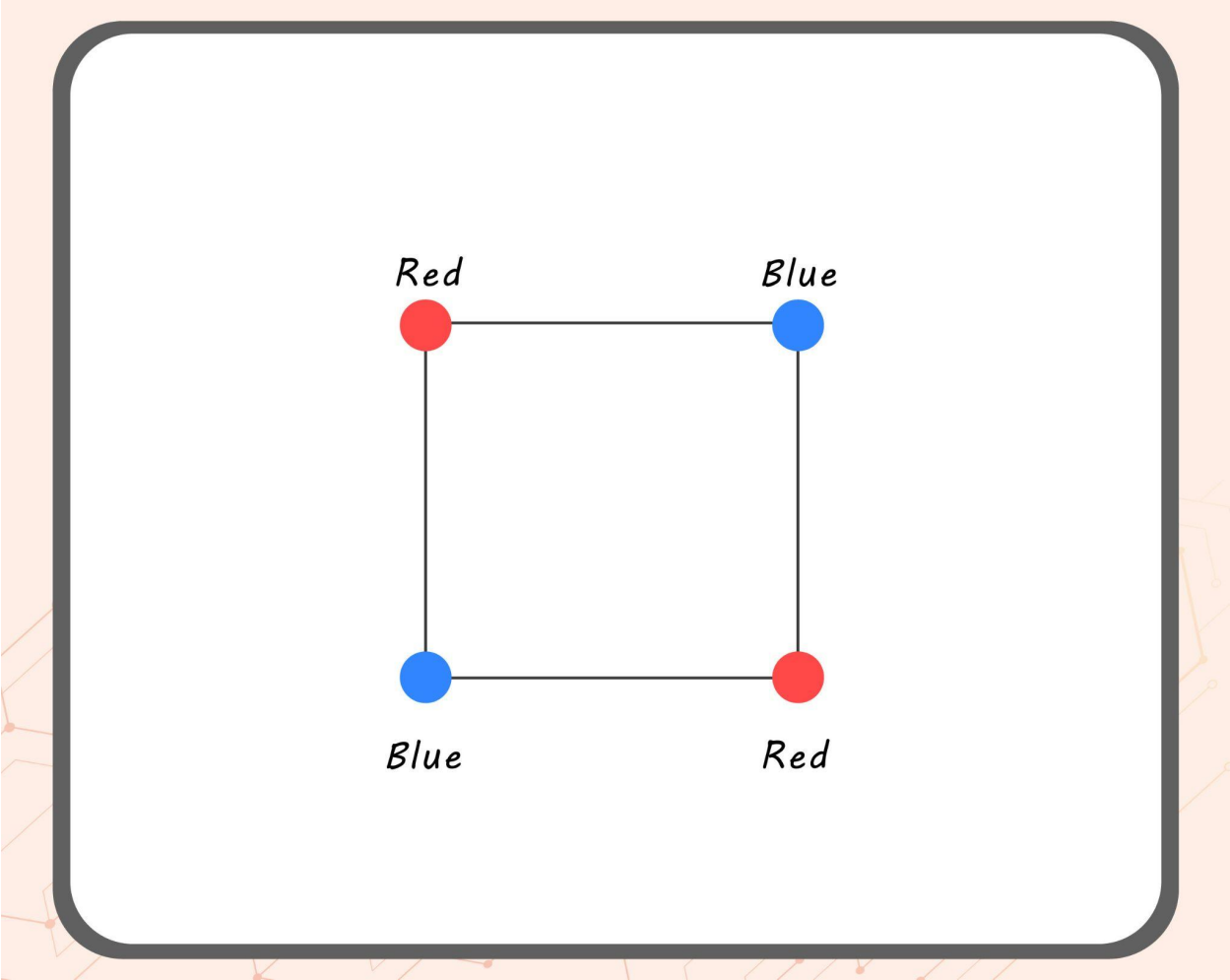
Let's take an example to understand the concept better



In the above graph, the nodes are marked with either Red Color or Blue Color. Now, these colors can be represented as Sets. So according to the definition of Bipartite

graphs there should be no edge between two red vertices or two blue vertices. So the above graph is clearly not Bipartite.

Now, let's take a look at another example



The above graph has no edge between the vertices of the same color (set), therefore, it is a bipartite graph.

To find whether a graph is bipartite or not, we follow the following steps:

1. We are going to maintain two sets and a queue.

2. Now, we are going to pick a starting vertex and place it in any of the sets (let's say set1) and we are going to push it in the queue.
3. Now, pop the vertex from the queue and place its neighbours in set2 and the queue.
4. On popping the next vertex from the queue, we have to check whether any of its neighbours is present in set2 or not.
5. If the 4th statement returns true, then the graph is not bipartite.

Let's take a look at the code for this algorithm:

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

bool bipartite(vector<int>* edges, int n) {
      if (n == 0) {
            return true;
      }
      unordered_set<int> sets[2];
      vector<int> pending;
      sets[0].insert(0);
      pending.push_back(0);
      while (pending.size() > 0) {
            int current = pending.back();
            pending.pop_back();
            int currentSet = sets[0].count(current) > 0 ? 0 : 1;
            for (int i = 0; i < edges[current].size(); i++) {
                  int neighbor = edges[current][i];
                  if (sets[0].count(neighbor) == 0 &&
sets[1].count(neighbor) == 0) {
                        sets[1 - currentSet].insert(neighbor);
                        pending.push_back(neighbor);
                  } else if (sets[currentSet].count(neighbor) > 0) {
                        return false;
```

```
                }
            }
        }
        return true;
}

int main() {
        while (true) {
            int n;
            cin >> n;
            if (n == 0)
                break;
            vector<int>* edges = new vector<int>[n];
            int m;
            cin >> m;
            for (int i = 0; i < m; i++) {
                int j, k;
                cin >> j >> k;
                edges[j].push_back(k);
                edges[k].push_back(j);
            }
            bool ans = bipartite(edges, n);
            delete [] edges;
            if (ans) {
                cout << "BICOLORABLE." << endl;
            } else {
                cout << "NOT BICOLORABLE." << endl;
            }
        }
}
```

Game Theory

Introduction

Game theory is the science that involves mathematical strategies to determine the winner at the end of a game. When we talk about game theory, we are generally referring to a **Combinatorial Game** (i.e. a game that has an outcome and doesn't get stuck in an infinite loop of moves or turns).

For example, in chess there may come a stage where the two players are left with no choice other than repeating their moves infinitely.

In combinatorial game theory there are two types of games:

1. **Impartial Games:** In an impartial game, the two players can play the same kind of moves.
2. **Partial Games:** In a partial game there is a restriction as to what moves the two players can play. For example, in the game of chess one player can only move pieces of a single color. That is, if player 1 is playing for the black side then he/she can't move the white pieces.

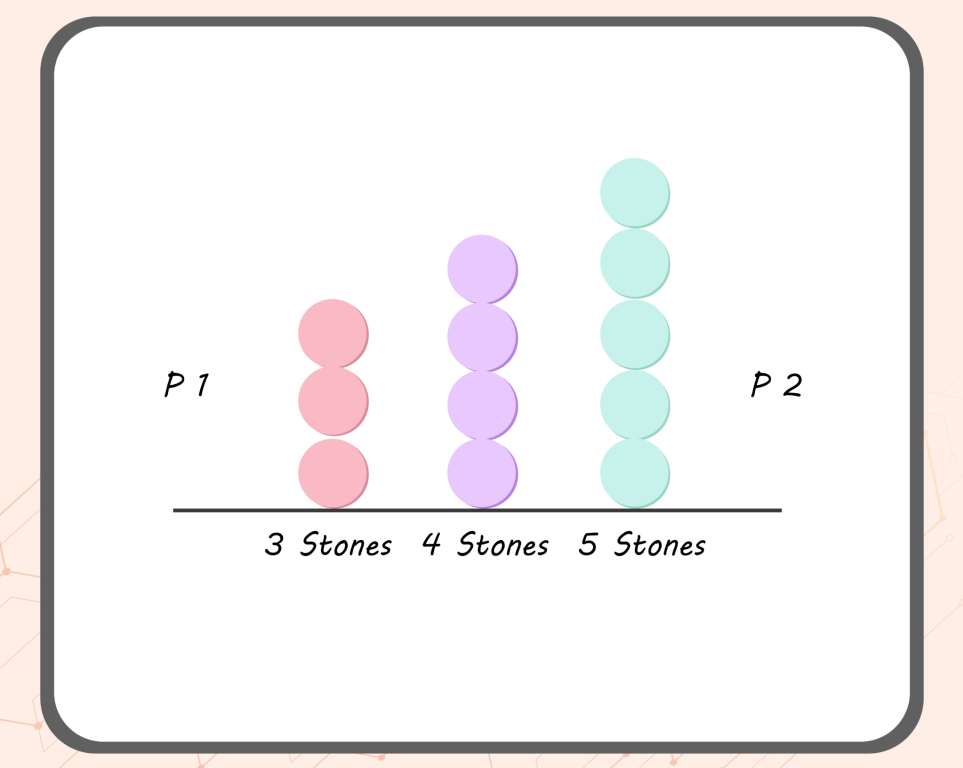
We have taken the example of chess above and seen how it is an impartial game and also that there may be a situation in the game where it is also not a combinatorial one.

Now, let's take a look at The game of Nims which is both a combinatorial game and also an impartial one.

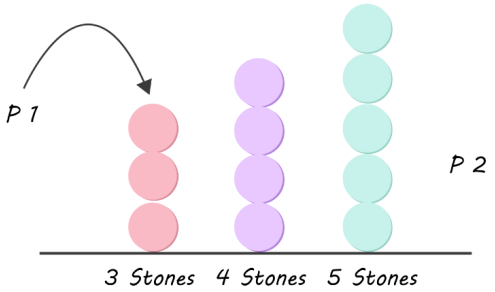
Game of Nims

In this game we are given numerous piles of stones and there are two players. The objective of the game is to remove a certain number of stones from one pile at a time in one move. The player may choose how many stones he/she wants to remove at their own convenience. The player who removes the last stone wins the game.

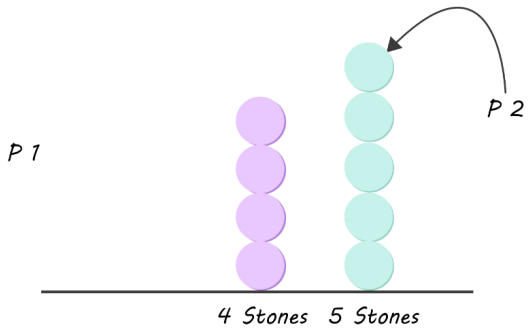
Let's take a look at the image given below for an example:



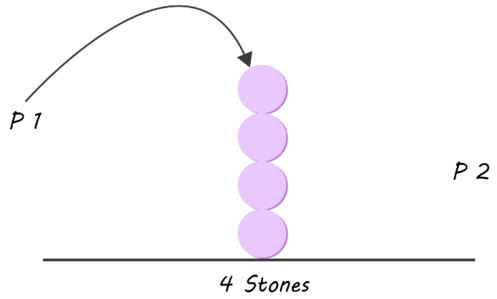
Now let's discuss one of the many possibilities in which **player 1 wins** the game.



* Player 1 removes all stones from pile 1

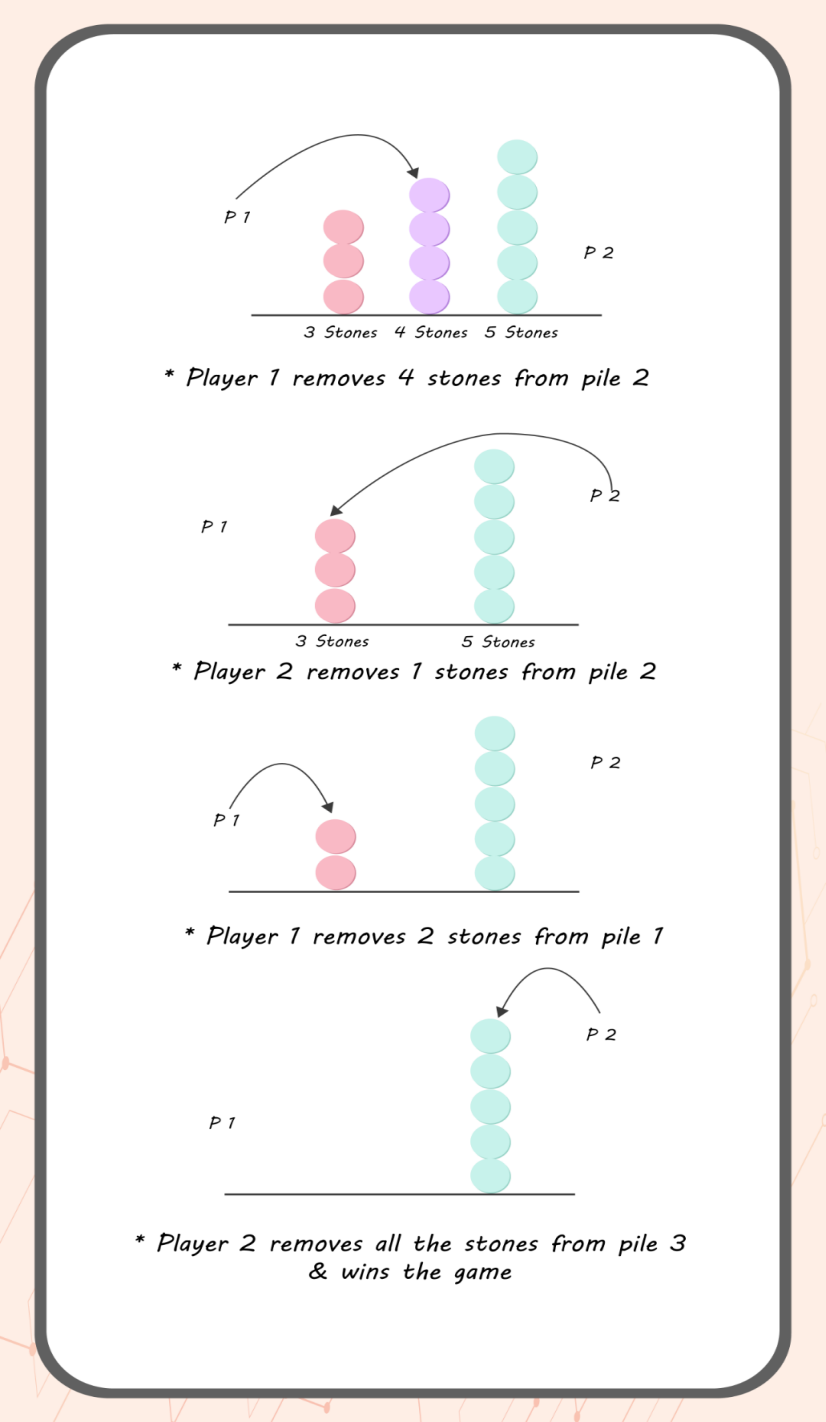


* Player 2 removes all stones from pile 3



* Player 1 removes all stones from pile 2
& wins the game

Now let's discuss one of the many possibilities in which **player 2 wins** the game.



Nim Formula:

Game theory says that we can predict the outcome of the game even before it is played.

We have to calculate a **Nim Sum** which is equal to the **cumulative XOR of the stones in each pile at the initial stage.** If the value of Nim Sum is **zero** and player 1 and player 2 are both playing optimally, then player 1 always loses. If the Nim Sum is **non zero** and player 1 and player 2 are both playing optimally then player 1 always wins.

In the above case Nim Sum = 3 ^ 4 ^ 5.

The proof of this formula is taught in the videos but the reader might not need to know the proof of concept to solve problems. So it is left for the reader to explore on their own.

mex

Before jumping on to what grundy numbers are, let's discuss a small mathematical concept called **mex.**

The mex or Minimum Excludent Set of a subset of a well ordered set is the smallest non-negative value from the whole set that does not belong to the subset.

For example, in a set of Non-Negative Integers, mex { 0, 1, 3 } = 2 because 2 is the smallest non-negative integer that is not present in the subset {0, 1, 3}.

Similarly, mex {1, 2, 3} = 0 because 0 is the smallest non-negative integer that is not present in the subset {1, 2, 3}.

Grundy Numbers

Let us consider a state **N** of a two player game and let **N-1** and **N-2** be the states reachable from N. We calculate the grundy value of N by

Grundy (N) = mex {Grundy(N-1), Grundy(N-2)}

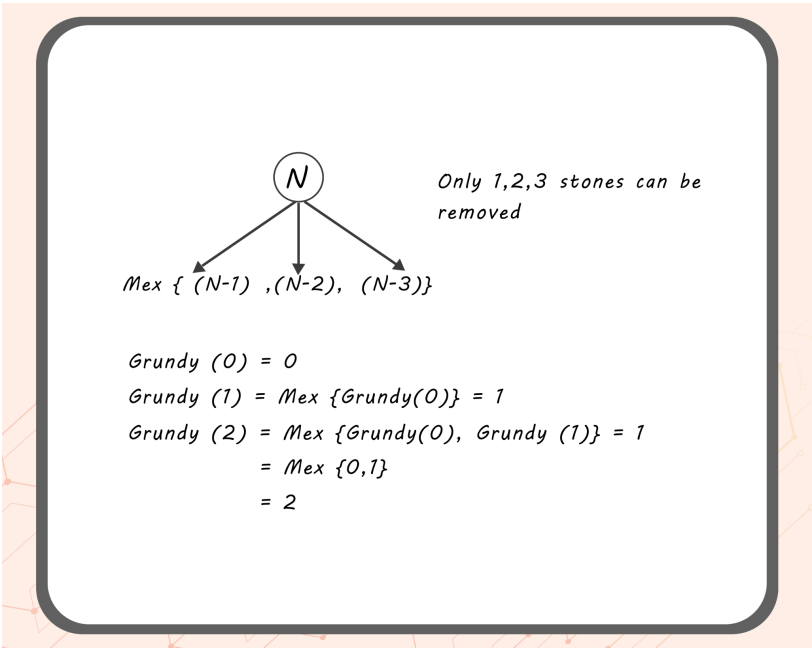
Also, Grundy (0) is always 0.

Let us take an example of the game of nims in which we have N number of stones initially and we can reach states from 0 to N-1.

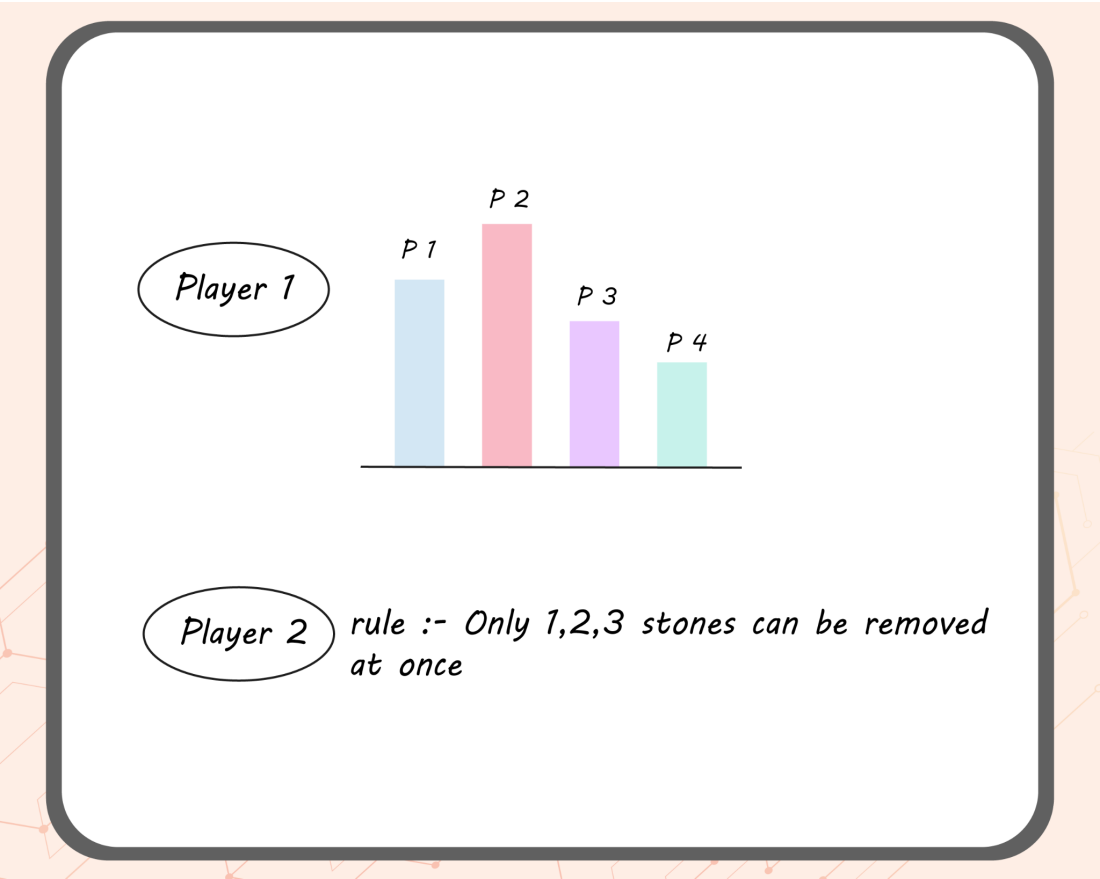
Then, **Grundy (N) = mex { Grundy (0), Grundy (1), Grundy (2) ..... Grundy (N-1) }**

Sprague Grundy Numbers

We have already studied the game of nims and how the value of Grundy(N) is calculated but what if the rules of the games are modified such that in one turn a player can only remove 1, 2 or 3 stones.



In such a game it would be very difficult to calculate the outcome of this game. So Sprague- Grundy Theorem states that instead of taking the cumulative XOR of the piles of stones initially present, we take the cumulative XOR of the grundy value of the piles.



If **Cumulative XOR (Grundy (p1), Grundy (p2), Grundy (p3), Grundy (p4))** is Non-Zero, then Player 1 always wins. Else, Player 2 always wins.

The proof of this concept is left for the reader to explore.

MiniMax Algorithm

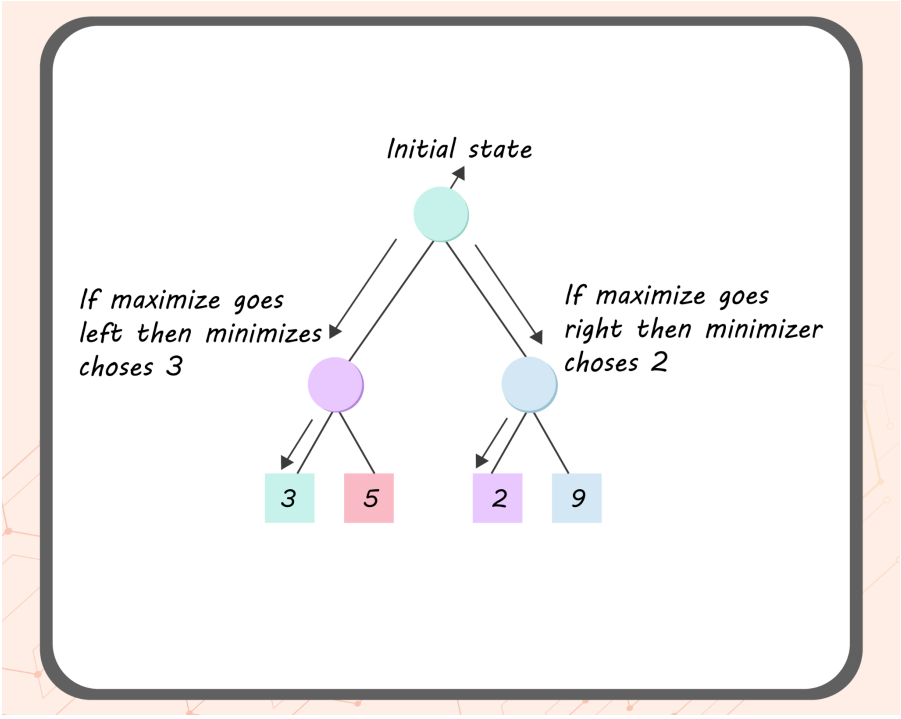
MiniMax is a kind of backtracking algorithm that is used in decision making to find the optimal move for a player, assuming that your opponent also plays optimally.

In MiniMax there are two players, Maximizer (who wants to maximize his score in the current state) and **Minimizer** (who wants to maximize his score in the current state).

Example:

Let's consider a game which has 4 final states and you are the maximizer and you have the first chance.

Since this is a backtracking algorithm, we have to explore all the paths and determine which would work the best.



Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go Left and the optimal value is 3.

Othello Evaluation Function

Rules of Othello Game:

- It is a two player game with a board size of 8X8 squares.
- At each round, the player has to play a disc on the board. One player plays white disc while the other plays black disc.
- The columns of the board are labeled A through H; the rows are labeled 1 through 8. A1 is the upper left square on the board; H8 is the lower right square.
- The initial configuration of the game is:
    - Two white discs in squares D4 and E5,
    - Two black discs in squares E4 and D5.
- If a player places a disc in a square (call it square X), and there is another square (call it square Y) on a row, column, or diagonal going through X, and all squares on the line between X and Y are filled with discs of the opponent's color, then the player is committing a legal move. After making a legal move, all discs between squares X and Y flip their color, i.e. the player making the legal move acquires the discs of the opponent. It is important to mention that a legal move can constitute disc flipping on more than one row, column, or diagonal.
- The game ends when all squares are filled, or when both players pass in two consecutive rounds.
- The winner is the player who has more discs on the board; the game is drawn when the disc count is even.
- It is a convention that all empty squares left on the board when the game ends are counted towards the winner.

String Algorithms

Introduction

In this section, we will primarily walk through two string searching algorithms—**Knuth Morris Pratt algorithm** and **Z-Algorithm**.

Suppose you are given a string **text** of length **n** and a string **pattern** of length **m**. You need to find all the occurrences of the **pattern** in the **text** or report that no such instance exists.

In the above example, the **pattern** string to be searched, that is, **"abaa"** appears at position 3 (0-indexed) in the **text** string **"abcabaabcabac".**

Naive Algorithm

A naive way to implement this pattern searching algorithm is to move from each position in the **text** string and start matching the **pattern** from that position until we encounter a mismatch between the characters of the two strings or say that the current position is a valid one.

In the given picture, The length of the **text** string is 5, and the length of the **pattern** string is 3. For each position from 0 to 3 in the **text** string, we choose it as the starting position and then try to match the next 3 positions with the **pattern**.

Naive pattern matching algorithm

- Run a loop for i from 0 to **n – m**, where **n** and **m** are the lengths of the **text** and the **pattern** string, respectively.

- Run a loop for j from 0 to **m– 1**, try to match the jth character of the **pattern** with (i + j)th character of the **text** string.
- If a mismatch occurs, skip this instance and continue to the next iteration.
- Else output this position as a matching position.

```
function NaivePatternSearch(text, pattern)
    //  iterate for each candidate position
    for i from 0 to text.length – pattern.length

        // boolean variable to check if any mismatch occurs
            match  = True

          for j from 0 to pattern.length – 1
                //  if mismatch make match = False
                if text[i  + j] not equals pattern[j]
                    match = False

        //  if no mismatch print this position
            if match == True
                print the occurrence  i
    return
```

Knuth Morris Pratt Algorithm

We first define a prefix function of the string - The prefix function of a string **s** is an array **lps** of length same as that of string **s** such that **lps[i]** stores the length of the maximum prefix that is also the suffix of the substring s[0..i].

For example:

For the pattern "AAABAAA",

lps[] is [0, 1, 2, 0, 1, 2, 3]

lps[0] is 0 by definition.

The longest prefix that is also the suffix of string s[0..1] which is AA is 1. (Note that we are only considering the proper prefix and suffix).

Similarly, For the whole string AAABAAA it is 3; hence the lps[6] is 3.

Algorithm for computing the LPS array

- We compute the prefix values **lps[i]** in a loop by iterating from 1 to **n – 1.**

- To calculate the current value **lps[i]** we set a variable j denoting the length of best suffix possible till index i – 1. So **j = lps[i – 1].**

- Test if the suffix of length j + 1 is also a prefix by comparing s[j] with s[i]. If they are equal then we assign lps[i] = j + 1 else reduce j = lps[j – 1].

- If we reach j = 0 , we assign lps[i] = 0 and continue to the next iteration .

```
function PrefixArray(s)
        n = s.length;
        //  initialize to all zeroes
        lps = array[n];

    for i from 1 to n – 1
            j = lps[i –1];
            //  update j untill s[i] becomes equal to s[j] or j becomes zero
            while j greater than 0 && s[i] is not equal to s[j]
                j = lps[j – 1];

            //  if extra character matches increase j
            if s[i] equal to s[j]
                j += 1;

            //  update lps[i]
            lps[i] = j;

        //  return the array
        return lps
```

Algorithm for searching the pattern using KMP Algorithm

Consider a new string **S' = pattern + '#' + 'text'** where + denotes the concatenation operator.

Now, the condition for the **pattern** string to appear at a position [i – M + 1… i] in the string text, the **lps[i]** should be equal to **M** for the corresponding position of i in S'.

Note that lps[i] cannot be larger than M because of the '#' character.

- Create S' = pattern + '#' + 'text'
- Compute the lps array of S.'
- For each i from **2*M** to **M + N** check the value of **lps[i].**
- If it is equal to **M,** then we have found an occurrence at the position **i – 2*M** in the string text else,; else continue to the next iteration.

```
function StringSearchKMP(text, pattern)
        //  construct the new string
        S' = pattern + '#' + text

    //  compute its prefix array
        lps = PrefixArray(S')
        N = text.length
        M = pattern.length
```

```
for i from 2*M to M + N
        // longest prefix match is equal to the length of pattern
        if lps[i] == M
            // print the corresponding position
            print the occurrence as i - 2*M
    return
```

Let us understand the above algorithm with the help of the following example:

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

lps[] = {0, 1, 2, 3}

i = 0, j = 0

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 1, j = 1

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 2, j = 2

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

pattern[i] and pattern[j] match, do i++, j++

i = 3, j = 3

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 4, j = 4

Since j == M, print pattern found and reset j,

j = lps[j–1] = lps[3] = 3

Here, unlike the Naive algorithm, we do not match the first three characters of this window. Value of lps[j– 1] (in the above step) gives us an index of the next character to match.

i = 4, j = 3

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 5, j = 4

Since j == M, print pattern found and reset j,

j = lps[j– 1] = lps[3] = 3

Again unlike Naive algorithm, we do not match first three

characters of this window. Value of lps[j– 1] (in above

step) gave us an index of the next character to match.

i = 5, j = 3

text[] = "AAAAAZAAAZA"

pattern[] =   "AAAA"

text[i] and pattern[j] do NOT match and j > 0, change only j

j = lps[j– 1] = lps[2] = 2

i = 5, j = 2

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] do NOT match and j > 0, change only j

j = lps[j– 1] = lps[1] = 1

i = 5, j = 1

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] do NOT match and j > 0, change only j

j = lps[j– 1] = lps[0] = 0

i = 5, j = 0

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] do NOT match and j is 0, we do i++.

i = 6, j = 0

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++ and j++

i = 7, j = 1

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++ and j++// the algorithm continues in the same fashion

Z-Algorithm

Link : [https://www.codingninjas.com/codestudio/problems/z-algorithm_1112619]

Problem Statement: You are given a string S of length N and a string P of length M. Your task is to find the number of occurrences of string P in the string S.

Note:

The string only consists of lowercase English alphabets.

Example:

Sample Input:

1

5 2

ababa

ab

Sample Output:

2

Explanation:

String **ab** occurs two times in the string **"ababa"**.

The first occurrence is from position 1 to position 2, and the second occurrence is from position 4 to position 5.

Approach 1: Brute Force Approach

A basic approach is to check all substrings of string **S** and see if any of the substrings is equal to **P**.

Steps:

1. Initialize an integer variable **count** = 0.
2. Start by running an outer loop (loop variable *i*) from 0 to **N – M**.
3. Then run a nested loop(loop variable *j*) from 0 to **M**.
4. For index *i*, check if **S[*i* + *j*] != P[*j*]**; if this condition is true, we simply break the inner loop and move to the next *i*. We do this to check every substring of length **M** if it is equal to **P** or not.
5. Later on, we check if *j* is equal to **M** or not; if this condition is true, we increment our **count** by one because we have found a valid substring of length **M** which is equal to string **P**.
6. Finally, we return the **count,** which is our final answer.

Time Complexity: O(N * M), where **N** and **M** are the lengths of String **S** and **P,** respectively.

Since for every substring, we are checking if it is equal to the string **P** or not; therefore, the time complexity is of **O(N * M).**

Space Complexity: O(1), We are not using any auxiliary space.

Approach 2: Z-Algorithm Approach

In this approach, we maintain a **Z array**. The **Z array** for any string **S** of length **N** is an array of size **N** where the *i*th element is equal to the greatest number of characters starting from the position *i* that coincide with the first characters of **S.**

The idea here is to concatenate both **S** and **P** and make a string **P#S**, where **'#'** is a special character that we are using. Now, for this concatenated string, we make a **Z** array. In this **Z** array, if the value at any index **(z[i])** is equal to **M**, we increment our **answer** (initially = 0) by 1 because this indicates that **P** is present at that index.

Steps:

1. Start by making a concatenated string **C**, where **C = P + # + S** and initialize an integer variable **count = 0.**
2. Make an array (**Z** array) of size **K**, where **K** is the length of string **C**.
3. Now initialize two integer values **L = 0 and R = 0.** We will be using these variables to create an interval to check if the string inside this interval matches **P** or not.
4. Now run a loop (loop variable *i*) from 1 till **K** and check if **i > R** or not.
    1. If **i > R**. This means that there is no substring that is starting before *i* and ending after *i*. So we reset the values of **L** and **R** and calculate this new interval of **[L, R]** by using the Z array method where we use a while loop. We simply check if **C[R – L] = C[R]** and **R < K** (to check if we are still inside the string **C**), and till the time both of these conditions are met, we simply increment our **R** by 1 and finally obtain the Z array element**(z[i]**) by using **z[i] = R – L** and then we decrease **R** by 1.
    2. If **i <= R**. Start by making another integer variable **pos = i – L.**
        1. We first check if **z[pos]** is less than the remaining interval **(R – i + 1)** or not. If it is so then, we simply make **z[i] = z[pos]** because this means that there is no prefix substring that starts at **C[i]** because if it was, then z[pos] would have been larger than the remaining interval **(R – i + 1).**
        2. If **z[pos]** was greater than or equal to the remaining interval **(R – i + 1)**. This condition implies that we can still extend our **[L, R]** interval. To do so, we set **L = i** . Now we start with **R** and manually check as we did in the case of **i > R** where we calculated **z[i] = R – L,** and then we decrease **R** by 1.

5. Since our **Z** array is created, we just traverse our array once and check if the current value of the array is equal to **M** or not. If it is equal to **M**, we increment the **count** by one.
6. Finally, we return **count**, which is our final answer.

Let us understand the above algorithm with the help of the following example :

For example- String S = abba and String P = ab

Here **N = 4 and M = 2**

1. We start by making a string **C = ab + "#" + abba = ab#abba**
2. Make a Z array, whose size is **K**, where **K** is the length of the string **C** (here, **K** = 7) and initialise it with zero.
3. Make two integer variables **L** = 0 and **R** = 0.
4. Run a loop from 1 till **K** (loop variable *i*) and start by checking if *i* less **R** or not.
5. Since our **i > R** we reset **L = i** and **R = i**. While **R < K(7)** and **C[R – L]=C[R]** we keep incrementing **R** and after coming out of this while loop we calculate z[1] and since this while loop is never executed because of **C[0] != C[1]**. Hence, **z[1] = R – L = 1 – 1 = 0.**
6. Next for index 2 we see that **i > R** just like index 1. Here also since **C[0] != C[2]** we find **z[2] = R – L = 2 – 2 = 0.**
7. For index 3 still **i < R**. But here we see that **C[0] = C[3]**, hence we increment **R** by 1 and still **C[1] = C[4]**, hence we again increment by 1 and finally since **C[2] != C[5]** we get out of the while loop and calculate **z[3] = R – L= 5 – 3 = 2** and change **R** to 4.
8. For index 4 we see that still **i >= R**. We make another variable **pos = 4 – 3 = 1**. And since **z[1] < R – i + 1, z[4] = z[1] = 0.**
9. Similarly, for the next index, also **z[5]=0.**
10. For the final index **z[6]**. Since our **i>=R**. We make another variable **pos = 6 – 5 = 1**. And since **z[1] < R – i + 1, z[6] = z[1] = 0.**
11. Our final Z array is [0,0,0,2,0,0,1], and in this array, there is only one element equal to **M(2)**, which is our answer.

Time Complexity: O(N + M), where **N** and **M** are the lengths of String **S** and **P,** respectively.

As we are traversing the string **C** only once and the length of C is **N + M**.

Space Complexity: O(K), here **K = N+M** , where **N** and **M** are the lengths of the string **S** and **P** respectively, as we are making an array of length **K**.

Tries

Introduction

Suppose we want to implement a word-dictionary using a C++ program and perform the following functions:

- Addition of the word(s)
- Searching a word
- Removal of the word(s)

To do the same, hashmaps can be thought of as an efficient data structure as the **average case time complexity** of insertion, deletion, and retrieval is O(1) for integer, character, float, and decimal values.

Let us discuss the time complexity of the same in case of strings.

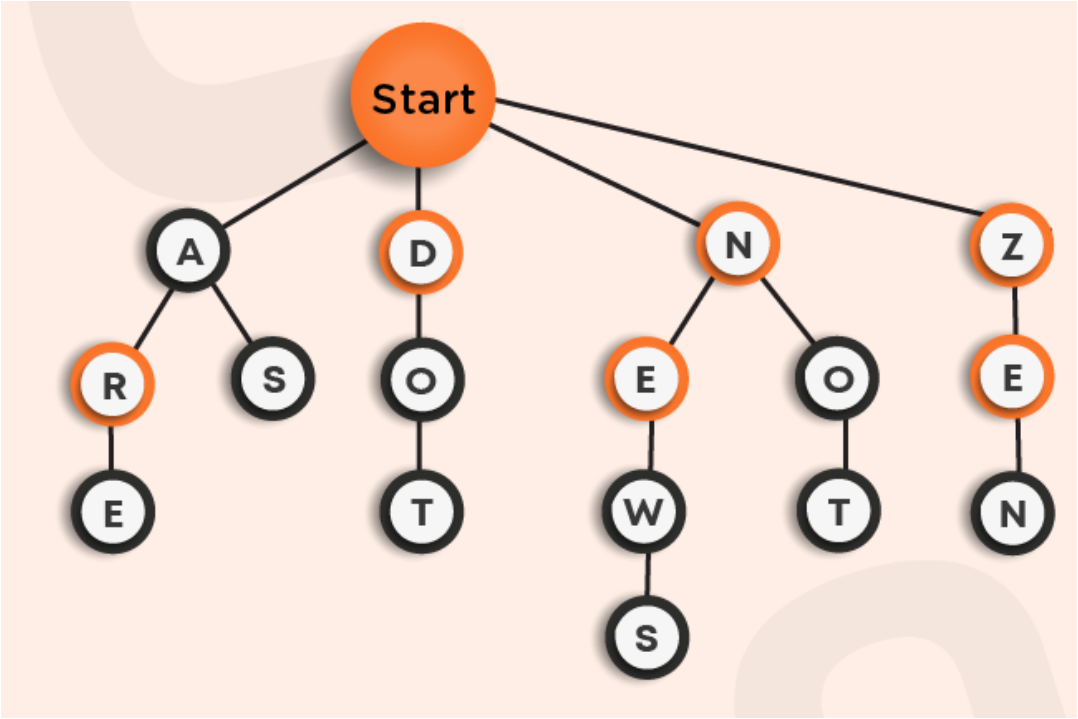
Suppose we want to insert string ***abc*** in our hashmap. To do so, first, we would need to calculate the hashcode for it, which would require the traversal of the whole string *abc*. Hence, the time taken will be the length of the entire string irrespective of the time taken to perform any of the above operations. Thus, we can conclude that the insertion of a string will be O(string_length).

To search a word in the hashmap, we again have to calculate the hashcode of the string to be searched, and for that also, it would require O(string_length) time.

Similarly, for removing a word from the hashmap, we would have to calculate the hashcode for that string. It would again require O(string_length) time.

For the same purpose, we can use another data structure known as **tries**.

Below is the visual representation of the trie:



Here, the node at the top named as the **start** is the root node of our **n-ary tree.**

Suppose we want to insert the word ***ARE*** in the trie. We will begin from the root, search for the first word ***A*** and then insert ***R*** as its child and similarly insert the letter ***E***. You can see it as the left-most path in the above trie.

Now, we want to insert the word ***AS*** in the trie. We will follow the same procedure as above. First-of-all, we find the letter ***A*** as the child of the root node, and then we will search for ***S***. If ***S*** was already present as the child of ***A***, then we will do nothing as the given word is already present otherwise, we will insert ***S***. You can see this in the above trie. Similarly, we added other words in the trie.

This approach also takes O(word_length) time for insertion.

Moving on to searching a word in the trie, for that also, we would have to traverse the complete length of the string, which would take O(word_length) time.

Let us take an example. Suppose we want to search for the word ***DOT*** in the above trie, we will first-of-all search for the letter ***D*** as the direct child of the start node and then search for ***O*** and ***T*** and, then we will return true as the word is present in the trie.

Consider another example ***AR***, which we want to search for in the given trie. Following the above approach, we would return true as the given word is present in it. However ideally, we should return false as the actual word was ***ARE*** and not ***AR***. To overcome this, it can be observed that in the above trie, some of the letters are marked with a bolder circle, and others are not. This boldness represents the termination of a word starting from the root node. Hence, while searching for a word, we will always be careful about the last letter that should be bolded to represent the termination.

Note: While insertion in a trie, the last letter of the word should be bolded as a mark of termination of a word.

In the above trie, the following are all possibilities that can occur:

- ARE, AS
- DO, DOT
- NEW, NEWS, NOT
- ZEN

Here, to remove the word, we will simply unbold the terminating letter as it will make that word invalid. For example: If you want to remove the string ***DO*** from the above trie by preserving the occurrence of string DOT, then we will reach ***O*** and then unbolden it. This way the word ***DO*** is removed but at the same time, another word ***DOT*** which was on the same path as that of word ***DO*** was still preserved in the trie structure.

For removal of a word from trie, the time complexity is still O(word_length) as we are traversing the complete length of the word to reach the last letter to unbold it.

When to use tries over hashmaps?

It can be observed that using tries, we can improve the space complexity.

For example: We have 1000 words starting from character ***A*** that we want to store. Now, if you try to hold these words using hashmap, then for each word, we have to store character ***A*** differently. But in case of tries, we only need to store the character ***A*** once. In this way, we can save a lot of space, and hence space optimization leads us to prefer tries over hashmaps in such scenarios.

In the beginning, we thought of implementing a dictionary. Let's recall a feature of a dictionary, namely **Auto-Search.** While browsing the dictionary, we start by typing a character. All the words beginning from that character appear in the search list. But this functionality can't be achieved using hashmaps as in the hashmap, the data stored is independent of each other, whereas, in case of tries, the data is stored in the form of a tree-like structure. Hence, here also, tries prove to be efficient over hashmaps.

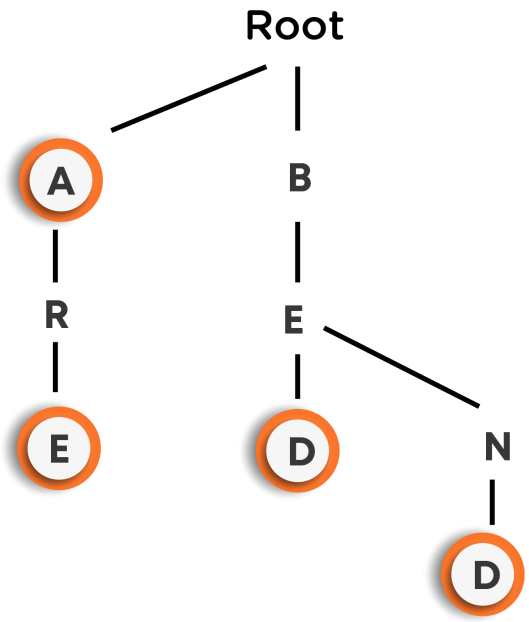
TrieNode class implementation

Follow the below-mentioned code (with comments)...

```
class TrieNode {
      public :
      char data;      // To store data (character value: 'A' - 'Z')
      TreeNode **children;   // To store the address of each child
      bool isTerminal;  // it will be TRUE if the word terminates at
this character
      TrieNode(char data) {  // Constructor for values initialization
            this -> data = data;
            children = new TrieNode*[26];
                        for(int i = 0; i < 26; i++) {
                                    children[i] = NULL;
                        }
            isTerminal = false;
      }
};
```

Insert function

To insert a word in a trie, we will use recursion. Suppose we have the following trie:



Now we want to insert the word ***BET*** in our trie.

Recursion says that we need to work on a smaller problem, and the rest will handle itself. So, we will do it on the root node.

Note: Practically, the functionality of bolding the terminal character is achieved using the boolean **isTerminal** variable for that particular node. If this value is true means that the node is the terminal value of the string, otherwise not.

Approach:

We will first search for letter ***B*** and check if it is present as the children of the root node or not and then call recursion on it. If ***B*** is present as a child of the root node as in our case it is, then we will simply recurse over it by shifting the length of the string by 1. In case, character ***B*** was not the direct child of the root node, then we have to create one and then call recursion on it. After the recursive call, we will see

that be is now a root node, and the character ***E*** is now the word we are searching for. We will follow the same procedure as done in searching for character ***B*** against the root node and then move forward to the next character of the string, i.e., ***T,*** which happens to be the last character of our string. Now we will check character ***T*** as the child of character ***E***. In our case, it is not a child of character ***E***, so we'll create it. As ***T*** is the last character of the string, so we will mark its **isTerminal** value to **True**.

Following will be the three steps of recursion:

- **Small Calculation:** We will check if the root node has the first character of the string as one of its children or not. If not, then we will create one.
- **Recursive call:** We will tell the recursion to insert the remaining string in the subtree of our trie.
- **Base Case:** As the length of the string becomes zero, or in other words, we reach the NULL character, then we need to mark the **isTerminal** for the last character as True.

Follow the code below, along with the comments...

```
class Trie {
      TrieNode *root;

      public :
      Trie() {
            root = new TrieNode('\0');
      }

      void insertWord(TrieNode *root, string word) {
            // Base case
            if(word.size() == 0) {
                  root -> isTerminal = true;
                  return;
```

```
            }
            // Small Calculation
            int index = word[0] - 'a';   // As for 'a' refers to index
0, 'b' refers to index 2 and so on, so to reach the correct index we
will do so
            TrieNode *child;
            if(root -> children[index] != NULL) {   // If the first
character of string is already present as the child node of the root
node
                  child = root -> children[index];
            }
            else {        // If not present as the child then creating
one.
                  child = new TrieNode(word[0]);
                  root -> children[index] = child;
            }

            // Recursive call
            insertWord(child, word.substr(1));
      }
      // For user
      void insertWord(string word) {
            insertWord(root, word);
      }
};
```

Search in tries

Objective: Create a search function which will get a string as its argument to be searched in the trie and returns a boolean value. **True** if the string is present in the trie and **False** if not.

Approach: We will be using the same process as that used during insertion. We will call recursion over the root node after searching for the first character as its child. If the first character is not present as one of the children of the root node, then we will simply return false; otherwise, we will send the remaining string to the recursion. When we reach the empty string, then check for the last character's **isTerminal** value; if it is **true,** it means that word exists in our trie, and we will return true from there otherwise, return false.

Try to code it yourselves, and for the code, refer to the solution tab of the same.

Tries implementation: Remove

Objective: To delete the given word from our trie.

Approach: First-of-all we need to search for the word in the trie, and if found, then we simply need to mark the **isTerminal** value of the last character of the word to **false** as that will simply denote that the word does not exist in our trie.

Check out the code below: (Nearly same as that of insertion, just a minor changes which are explained along side)

```
void removeWord(TrieNode *root, string word) {
            // Base case
            if(word.size() == 0) {
                  root -> isTerminal = false;
                  return;
            }

            // Small calculation
            TrieNode *child;
            int index =  word[0] - 'a';
            if(root -> children[index] != NULL) {
                  child = root -> children[index];
            }
            else {
                  // Word not found
                  return;
            }

            removeWord(child, word.substr(1));
// Suppose if the character of the string doesn't have any child and
is a part of the word to be deleted, then we can simply delete that
node also as it is not referencing to any other word in the trie

            // Removing child Node if it is useless
```

```
        if(child -> isTerminal == false) {
            for(int i = 0; i < 26; i++) {
                if(child -> children[i] != NULL) {
                    return;
                }
            }
            delete child;
            root -> children[index] = NULL;
        }
    }

        // For user
         void removeWord(string word) {
        removeWord(root, word);
    }
```

Types of tries

There are two types of tries:

- **Compressed tries:**
  - Majorly, used for space optimization.
  - We generally club the characters if they have at most one child.
  - **General rule:** Every node has at least two child nodes.

  Refer to the figure below:

  Suppose our regular trie looks like this-

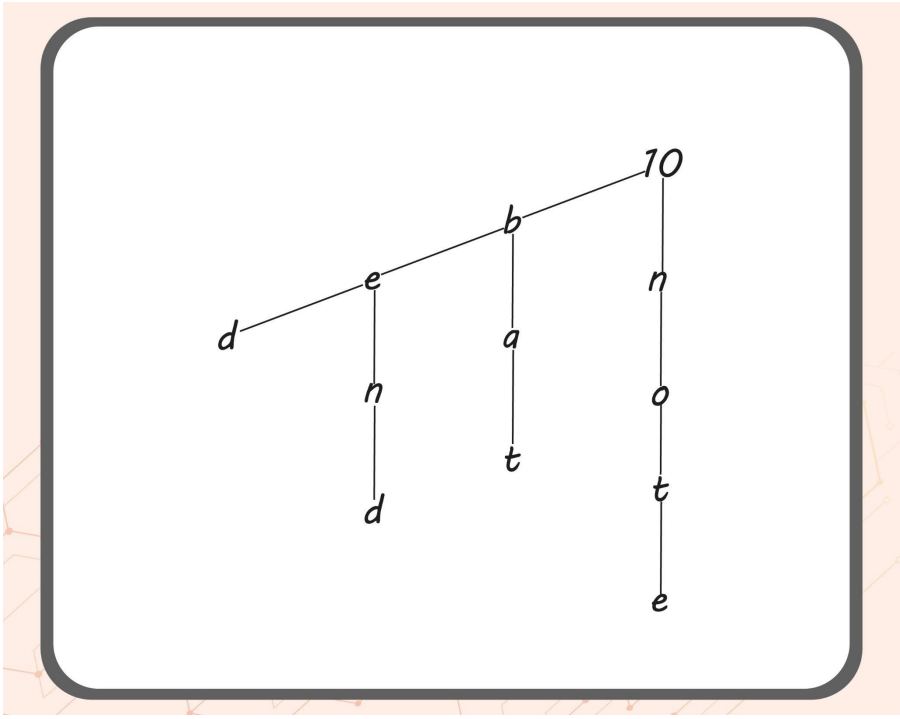
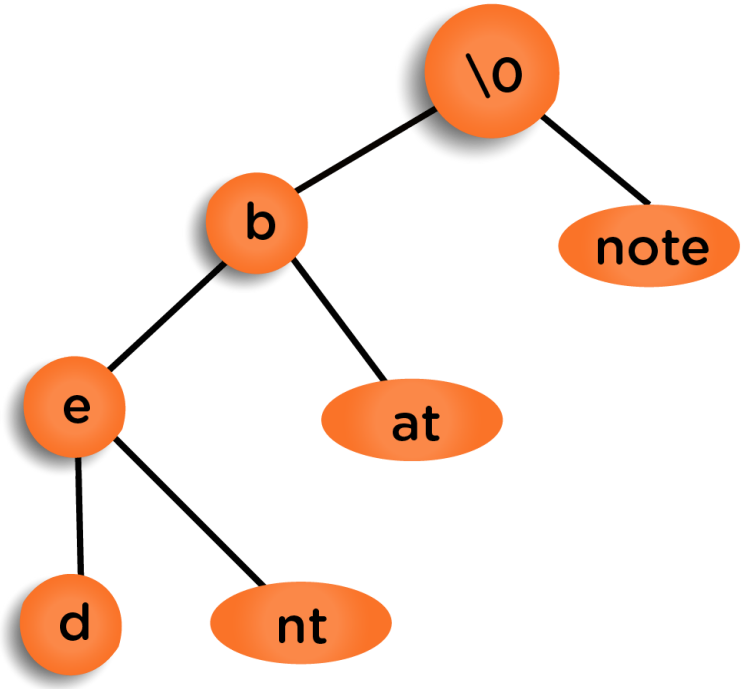


Figure - 1

  Its compressed trie version will look as follows:



- **Pattern matching:**
    - Used to match patterns in the trie.
        - Example: In the figure-1 (shown above), if we want to search for pattern **ben** in the trie, but the word **bend** was present instead, using the normal search function, we would return false, as the last character **n**'s isTerminal property was false, but in this trie, we would return true.
    - To overcome this problem of the last character's identification, just remove the **isTerminal** property of the node.
    - In the figure-1, instead of searching for the pattern **ben**, we now want to search for the pattern **en**. Our trie would return false if **en** is not directly connected to the root. But as the pattern is present in the word **ben**, it should return true. To overcome this problem, we need to

attach each of the prefix strings to the root node so that every pattern is encountered.

- **For example:** for the string **ben,** we would store **ben**, **en**, **n** in the trie as the direct children of the root node.

Maximum XOR Subarray

Problem Statement: We are given an array and we have to find a subarray that has the maximum XOR possible.

Explanation:

Example:

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

Output: 7

The subarray {3, 4} has maximum XOR value

Basic Approach: The basic approach to the solution of this problem includes calculating the XOR of every possible two elements. This would take $O(N^2)$ time.

Trie Approach:

Algorithm:

Maintain a trie data structure which can handle two types of queries:

1. Insert an element (bitwise representation) into trie.
2. For each element Y:
    a. Find the maximum XOR of Y with all the numbers that have been inserted till now.
    b. Insert integers as you go, and keep finding the maximum XOR and keep track of the overall maximum.

Let's say our number Y is b1,b2...bn, where b1,b2.. are binary bits. We start from b1. Now for the XOR to be maximum, we'll try to make the most significant bit 1 after taking XOR. So, if b1 is 0, we'll need a 1 and vice versa. In the trie, we go to the

required bit side. If a favorable option is not there, we'll go to the other side. Doing this all over for i=1 to n, we'll get the maximum XOR possible.

Sub - XOR

Problem Statement: Given an array of positive integers, you have to print the number of subarrays whose XOR is less than K.

Explanation:

Example:

Input: arr[] = {8, 9, 10, 11, 12}, k=3

Output: 4

Sub-arrays [1:3], [2:3], [2:5], [4:5] have xor

values 2, 1, 0, 1 respectively.

We will calculate all of the prefix XOR values i.e. arr[1:i] for all i. It can be verified that the xor of a subarray arr[l:r] can be written as (arr[1:l-1] xor arr[1:r]), where arr[i, j] is the xor of all the elements with index such that, i <= index <= j.

If xor[i, j] represents the xor of all elements in the subarray a[i, j], then at an index i what we have is, a trie which has xor[1:1], xor[1:2].....xor[1:i-1] already inserted. Now we somehow count how many of these (numbers in trie) are such that its xor with xor[1:i] is smaller than k. This will cover all the subarrays ending at the index i and having xor i.e. xor[j, i] <=k;

Now the problem remains, how to count the numbers with xor smaller than k. So, for example take the current bit of the ith index element is p, current bit of number k be q and the current node in trie be node.

Take the case when p=1, k=1. Then if we go to the right child the current xor would be 0 (as the right child means 1 and p=1, 1(xor)1=0).As k=1, all the numbers that are

to the right child of this node would give xor value smaller than k. So, we would count the numbers that are right to this node.

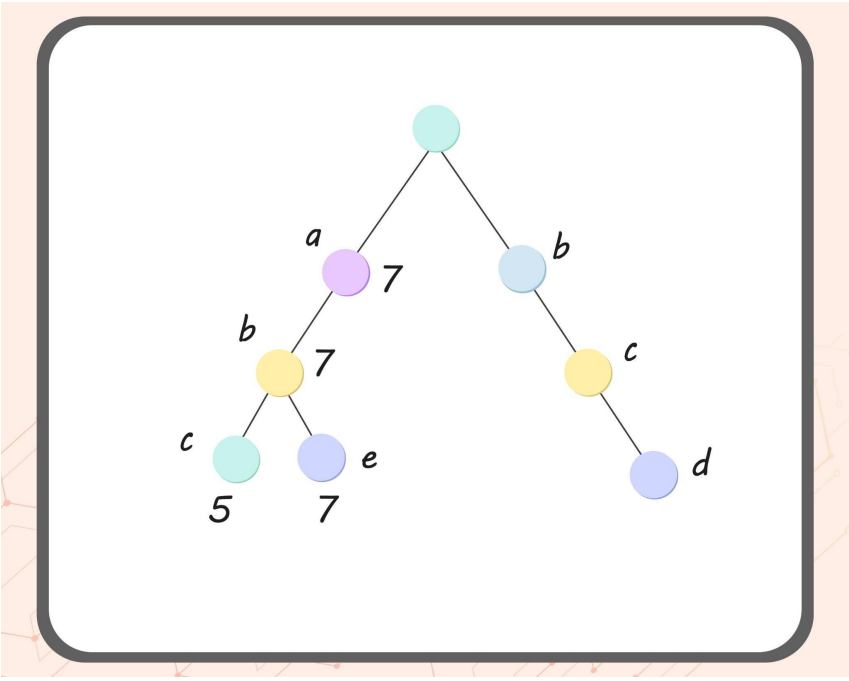
If we go to the left child, the current xor would be 1 (as the left child means 0 and p=1, 0(xor)1=1). So, if we go to the left child we can still find a number with xor smaller than k, therefore we move on to the left child.

Search Engine

Problem Statement: We are given N strings along with their weights. We get Q queries and in each query we get a prefix and we have to find the string with the best weight that contains that prefix.

Explanation:

We need the node in the trie to store the optimal weight among the prefixes that is formed by its children. Suppose we are given strings “abc”, “bcd”, “abe” with weights 5, 6, 7 respectively. Let’s take a look at the image given below to see how the trie is formed for this example:



Now we get the query along with the prefix “ab” so “ab” occurs in “abc” and “abe”. So as shown in the diagram at node a and b in the LHS, we store the optimal weight as the answer which in this case is 7 from the string “abe”.

Computational Geometry

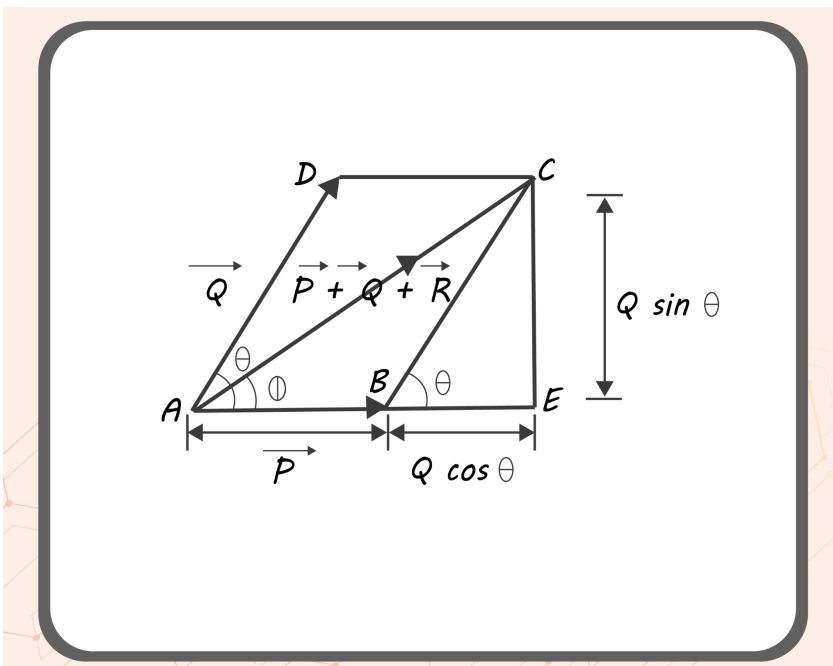
Introduction

To proceed with the applications of geometry in competitive programming, we need to have the basic knowledge of operations like vector addition, dot product, cross product, etc.

Let's take a look at some of them.

Vector Addition:

According to the parallelogram law of vector addition, if two vectors P and Q are represented by two adjacent sides of a parallelogram both pointing outwards, then the diagonal drawn through the intersection of the two vectors represents the resultant.



the magnitude of the resultant is given by:

$R = (P^2 + Q^2 + 2PQ\cos\Theta)^{½}$

The direction of the resultant vector is determined as follows:

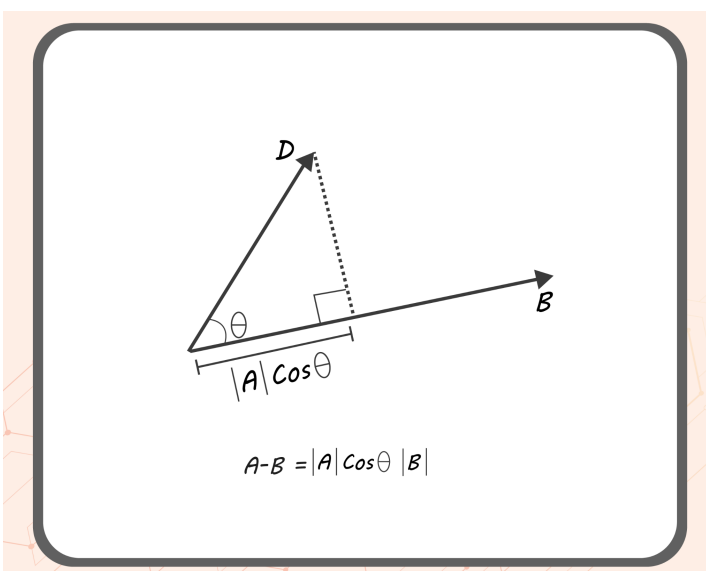
$\tan^{-1} (Q\sin\Theta / (P+Q\cos\Theta))$

Vector Subtraction:

Let us consider that vector a is to be subtracted from vector b. This can also be inferred as the addition of vector a and vector b (in the opposite direction). Thus the formula for addition can be used since (-b) is nothing but b in the reversed direction.

Dot Product:

Geometrically it is a product of the length of the first vector by the length of the projection of the second vector onto the first one.



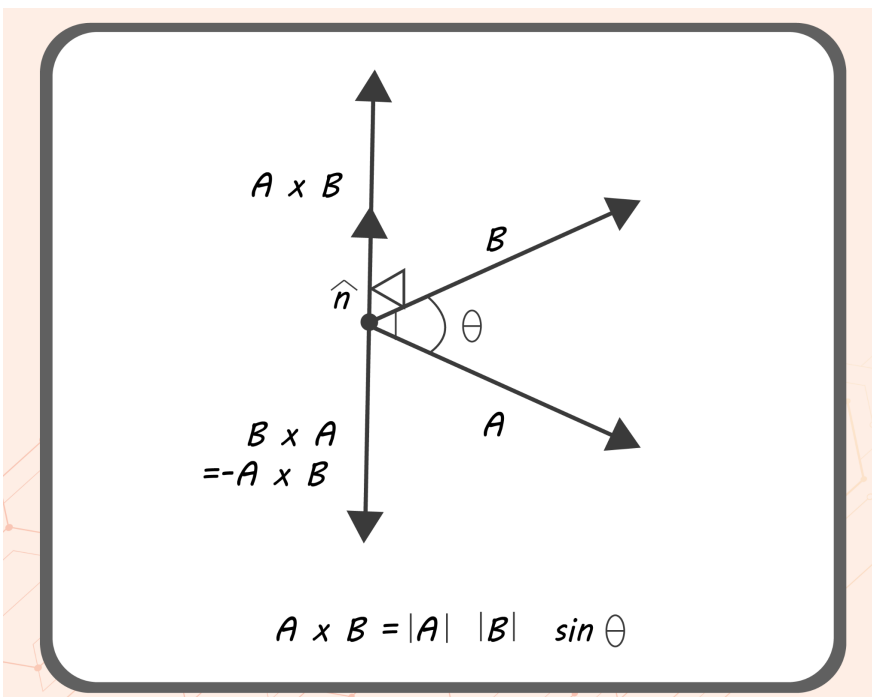
$A \cdot B = |A|\cos\theta \cdot |B|$

The dot product holds some notable properties:

1. $A \cdot B = B \cdot A$
2. $(\alpha \cdot A) \cdot B = \alpha \cdot (A \cdot B)$
3. $(A+B) \cdot C = A \cdot C + B \cdot C$

Cross Product:

The vector product or cross product of two vectors A and B is denoted by A × B, and its resultant vector is perpendicular to the vectors A and B.



$$A \times B = |A|\ |B| \sin \theta$$

Consider two vectors,

A = ai + bj + ck

B = xi + yj + zk

We know that the standard basis vectors i, j, and k satisfy the below-given equalities.

$i \times j = k$ and $j \times i = -k$

$j \times k = i$ and $k \times j = -i$

$k \times i = j$ and $i \times k = -j$

Now, $A \times B = (ai + bj + ck) \times (xi + yj + zk) = ax(i \times i) + ay(i \times j) + az(i \times k) + bx(j \times i) + by(j \times j) + bz(j \times k) + cx(k \times i) + cy(k \times j) + cz(k \times k)$

By applying the above mentioned equalities,

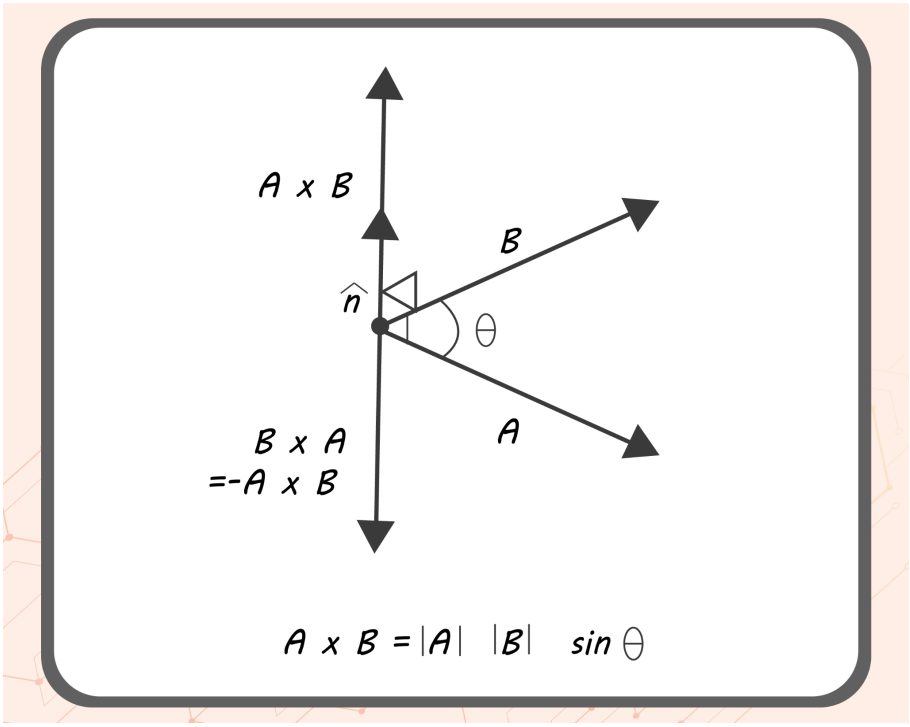
$A \times B = ax(0) + ay(k) + az(-j) + bx(-k) + by(0) + bz(i) + cx(j) + cy(-i) + cz(0)$

$= (bz - cy)i + (cx - az)j + (ay - bx)k$

Distance of a Point

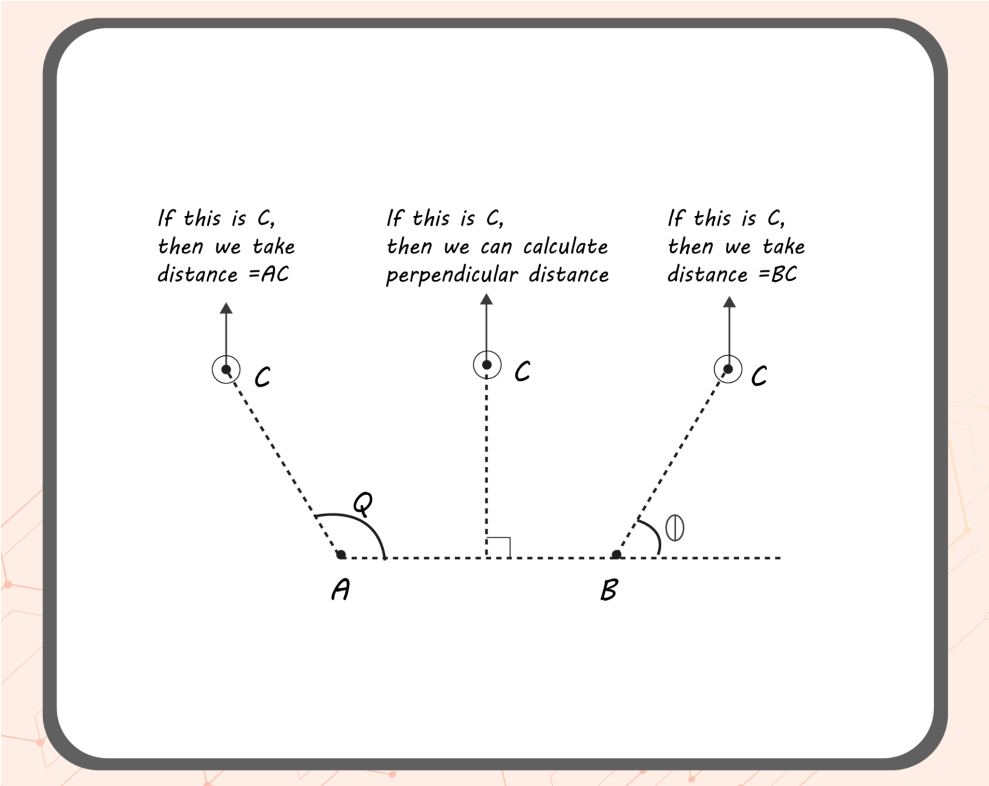
Distance of a point from a line:

Consider a line segment AB and a point C not on the line extending AB. We have to calculate the perpendicular distance of point C from the line.



Distance of a point from a line segment:

If we can't extend the line segment then there can be three cases in regards to calculating the distance. The position of C can be divided in three categories shown in the figure given below:

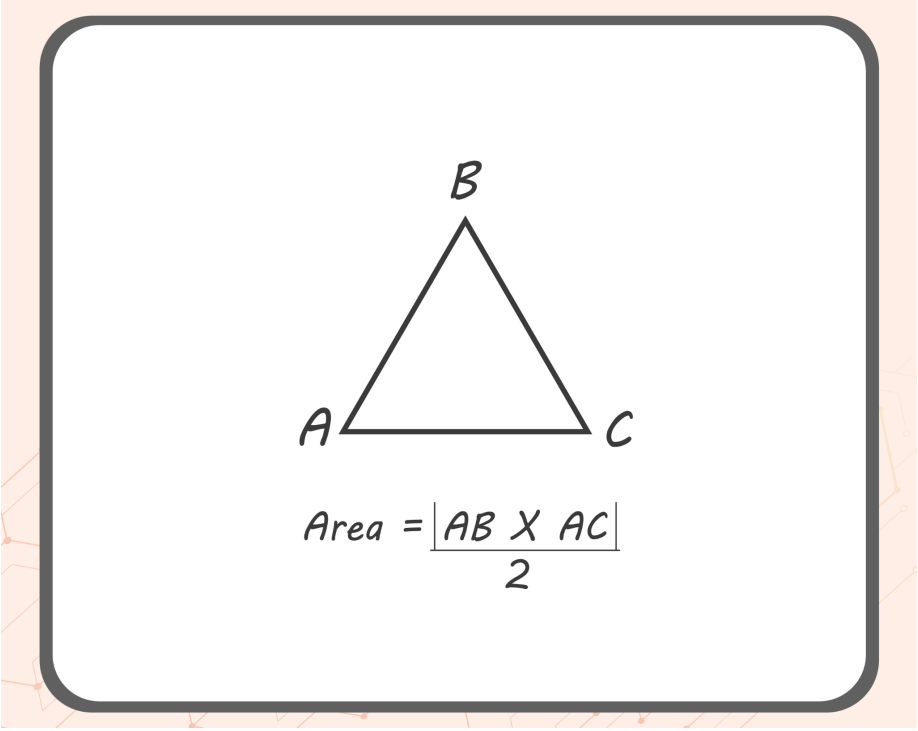


Now, to calculate where C is, we use the concept of the dot product. Let us say the angle between AC and AB is Θ. Then if **AB.AC** is negative, then Θ >= 90°, which means that point C is nearest to pt A on the line segment. If AB.AC is positive, then we check for the angle between AB and BC.

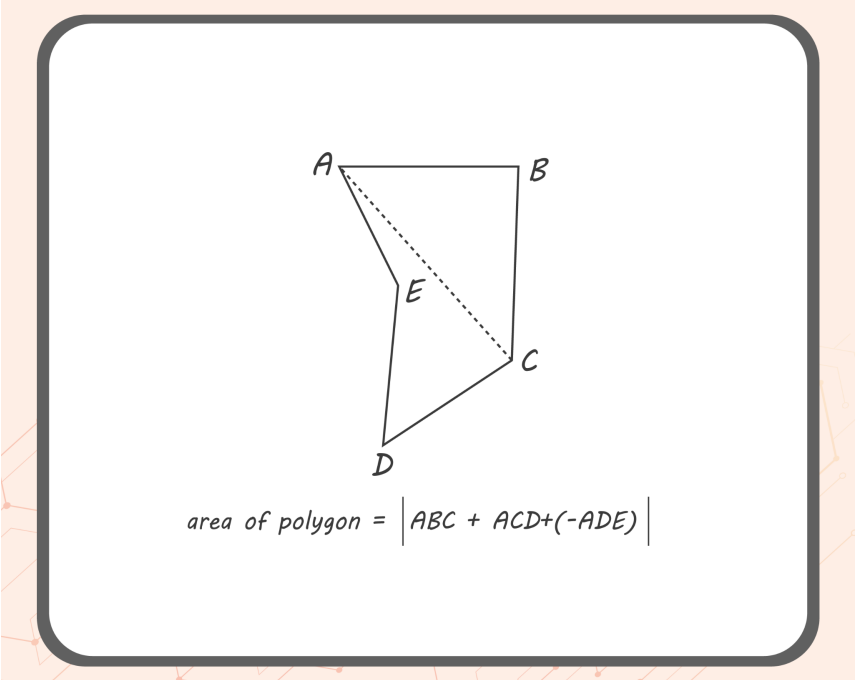
If both the conditions don't hold true then we can calculate the perpendicular distance.

Area of a Polygon

We know that the area of a **triangle ABC** is calculated as **(|ABxAC|)/2**.



This concept/formula can be used to calculate the area of a polygon. We divide the polygon into triangles and the summation of the areas will give us the area of the polygon.



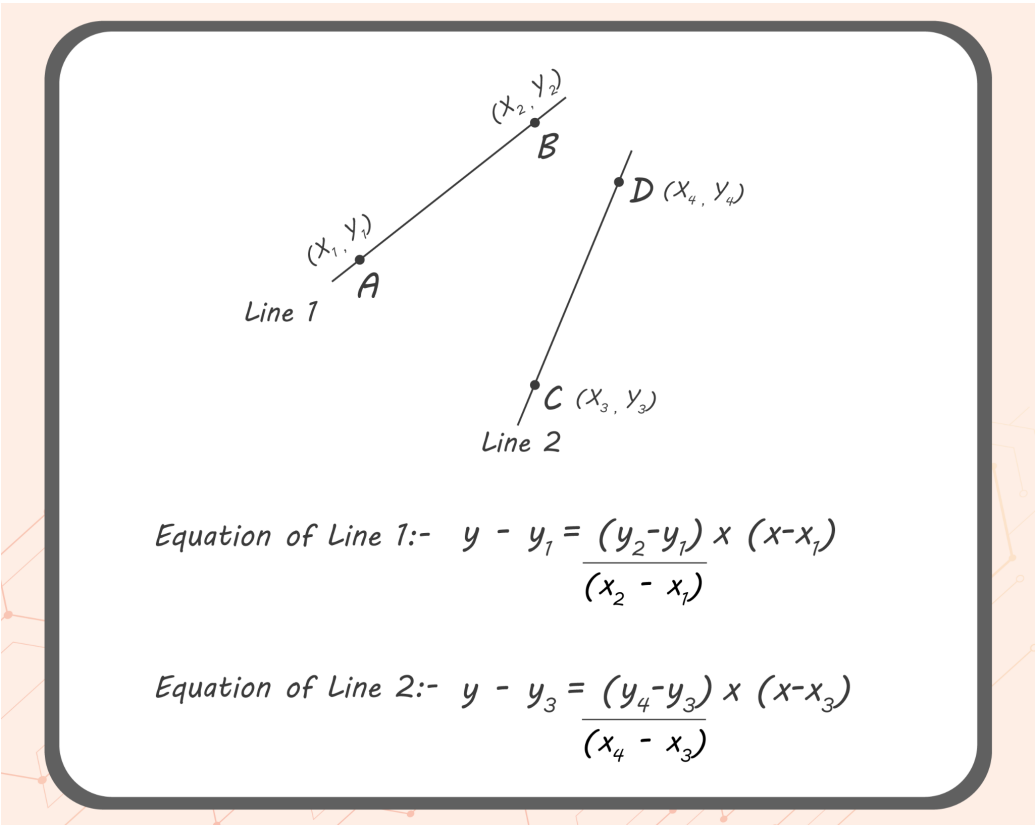
Note: We have to take the cross product because direction of the result also matters since there may be some triangles that form outside the polygon and might need to be subtracted.

Intersection of Two Lines

Problem Statement: We have 4 points A, B, C, D in a 2D plane. Points A and B are on one line and points C and D are on the other. We have to find the point where the two lines intersect each other.

Explanation:

If two points are on a line then the equation of the line can be formed as :



Now, comparing the equation of line 1 with the general equation of line, $\mathbf{a_1x + b_1y = c_1}$ and comparing equation of line 2 with the general equation of line $\mathbf{a_2x + b_2y = c_2}$, we get the values of a_1, a_2, b_1, b_2, c_1, c_2.

$$a_1 = (y_2 - y_1) \qquad a_2 = (y_4 - y_3)$$

$$b_1 = (x_2 - x_1) \qquad b_2 = (x_4 - x_3)$$

$$C_1 = a_1 x_1 \; b_1 y_1) \qquad C_2 = a_2 x_3 + \; b_2 y_3$$

Now that we have the variables we need to find the intersection point. Using Cramer's rule:

$$x = -\frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{c_1 \; b_2 - C_2 b_1}{a_1 \; b_2 - a_2 b_1},$$

$$y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = -\frac{a_1 \; c_2 - a_2 c_1}{a_1 \; b_2 - a_2 b_1},$$

If the denominator equals 0, then either the system has no solutions (the lines are parallel and distinct) or there are infinitely many solutions (the lines overlap). If we need to distinguish these two cases, we have to check if coefficients c are proportional

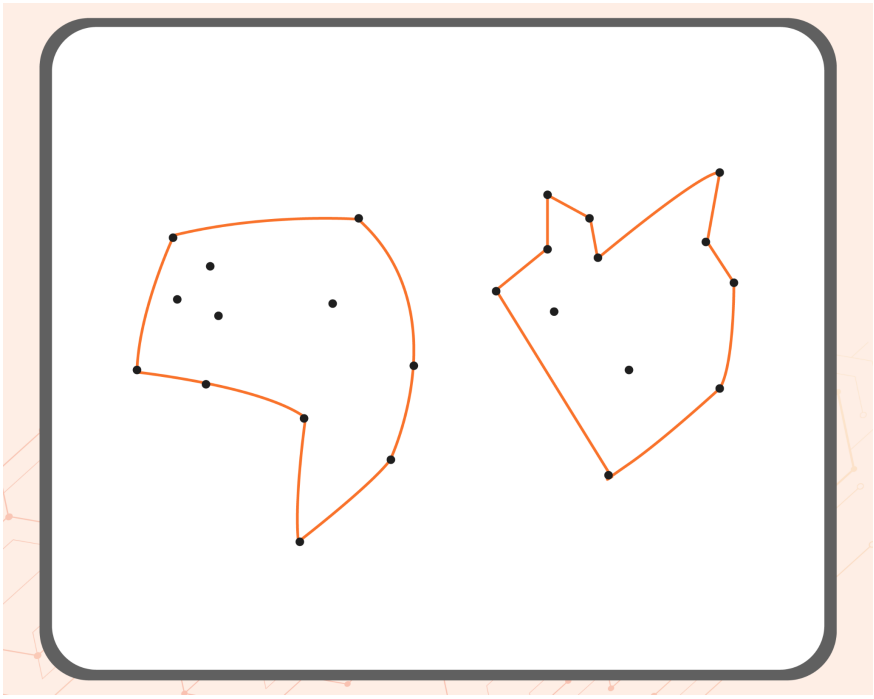
with the same ratio as the coefficients a and b. To do that we only have to calculate the following determinants, and if they both equal 0, the lines overlap.

Convex Hull

Problem Statement: We are given several points on a 2D plane and we have to create a boundary using these points such that no point remains outside and also that the boundary is minimized.

Explanation:

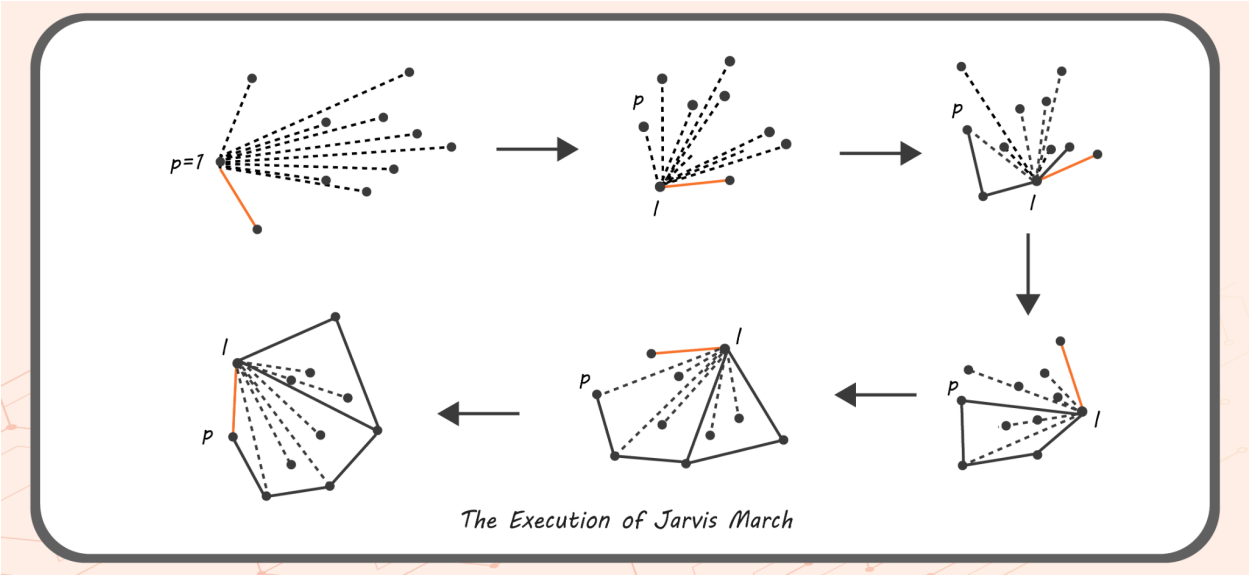
To understand what is meant by the boundary minimization let us take a look at the two possible boundaries that can be formed using the same set of points:



We use the **Jarvis Algorithm** to wrap the given set of points in the convex hull:

1. Initialize p as the leftmost point.
2. Do the following while we don't come back to the first (or leftmost) point.

a. The next point q is the point such that the triplet (p, q, r) is counterclockwise for any other point r. To find this, we simply initialize q as the next point, then we traverse through all points. For any point i, if i is more counterclockwise, i.e., orientation(p, i, q) is counterclockwise, then we update q as i. Our final value of q is going to be the most counterclockwise point.

b. next[p] = q (Store q as next of p in the output convex hull).

c. p = q (Set p as q for next iteration).