



Mastering **COMPETITIVE PROGRAMMING**





Competitive Programming Module

Content

Chapter No.	Chapter Name	Page
1	Generic Programming & STL	1
2	Mathematics	29
3	Number Theory	62
4	Binary Search (Divide & Conquer)	93
5	Greedy Algorithms	101
6	Recursion & Backtracking	125
7	Segment Tree (Divide & Conquer)	135
8	Binary Indexed Tree/Fenwick Tree	161
9	Dynamic Programming	175
10	MO's Algorithm	229
11	Graph Algorithms	240
12	Game Theory	286
13	Geometric Algorithms	296
14	Fast Fourier Transform	309
15	Heavy Light Decomposition (HLD)	318

1

Generic Programming & Standard Template Library

Before we start with C++ Standard Template Library, we must know some basics about generic programming in C++ and how templated classes are made. Let's us start our journey with C++ templates.

C++ Templates

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

C++ STL has some containers (pre-build data structures) like **vectors**, **iterators**, **pairs** etc. These are all generic class which can be used to represent collection of any data type.

Generic Programming using C++ Templates

Every program we write revolves around 3 things - data, algorithms and containers.

Let us try to understand the role of each.

- 1. Data :** Every program has some input and output data. For e.g.: A program to control the movement of self-driving car will have some input data like images, visual data from surrounding and output could be int/floating value denoting the required acceleration of the car.
- 2. Algorithm :** Algorithm is the required logic which operates on the input to generate desired output. These algorithms can be made generic using templates in C++.
- 3. Container :** A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements. The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators. C++ provides various containers like list, vector(dynamic array), stack, queue etc which are used in algorithms for storing in data.

Making Generic Functions & Classes in C++

Let us consider a simple function to search an element in an array.

```
int search(int arr[], int n, int elementToSearch) {
    for (int pos = 0; pos < n; ++pos) {
        if (arr[pos] == elementToSearch)
```

```

        return pos;
    }
}
return -1;
}

```

Observation :

The search function above works only for an *integer array*. However the functionality *search*, is logically separate from array and applicable to all data types, i.e. searching a char in a char array or searching is applicable in searching in a linked list.

Hence, data, algorithms and containers are logically separate but very strongly connected to each other.

Generic Programming enables programmer to achieve this logical separation by writing general *algorithms that work on all containers with all data types*.

Separating Data

A function can be made general to all data types with the help of templates.

Templates are a blueprint based on which compiler generates a function as per the requirements.

To create a template of search function, we *replace int with a type T* and tells compiler that *T is a type* using the *statement template <class T>*

```

template <class T>
int search(T arr[], int n, T elementToSearch) {
    for (int pos = 0; pos < n; ++pos) {
        if (arr[pos] == elementToSearch) {
            return pos;
        }
    }
    return -1;
}

```

Now the search function runs for all types of arrays for which statement 4 is defined.

Now the search function runs for all types of arrays for which statement 4 is defined.

```

int arrInt[100];
char arrChar[100];
float arrFloat[100];
Book arrBook[100], X; //X is a book

```

```
search(arrInt, 100, 5); //T is replaced by int
search(arrChar, 100, 'A'); //T is replaced by char
search(arrFloat, 100, 1.24); //T is replaced by float
search(arrBook, 100, X); //T is replaced by Book.
```

So, one function can be run on different data types. This makes our function *general* for all types of data.

Note

In the actual code that is produced after compilation, 4 different functions will be produced based on the template with T replaced accordingly.

You could use `<typename T>` or `<class T>` in the template statement 1. The keywords `typename` and `class` serve the same purpose.

Separating Algorithm

The search function will not work for Book objects if computer doesn't know how to compare 2 books. So our function is limited in some sense. To work it for all data types, we use a concept of *comparator* (also see *predicate*).

Let's rewrite the templated search function again

```
template <class T, class Compare>
int search(T arr[], int n, T elementToSearch, Compare obj) {
    //compare is a class that has () operator overloaded
    for (int pos = 0; pos < n; ++pos) {
        if (obj(arr[pos], elementToSearch) == true) {
            //obj compares elements of type T
            return pos;
        }
    }
    return -1;
}
```

To use the search function for integers, you shall now write:

```
//defining a class compare
class compareInt {
public:
    bool operator()(int a, int b) {
```

```
    return a == b ? true : false;
}
};

//calling search templated function
compareInt obj;
    //obj is the object of class compareInt
search(arrInt, 100, 20, obj); //T replaced with int
    //Compare replaced with compareInt
```

Line 4 works since obj(xint, yint) is defined by the class compareInt.

To use search function for a book class, we should write a compareBook class.

```
class compareBook{
public:
bool operator()(const Book& B1, const Book& B2){
return B1.getIsbn() == B2.getIsbn();
}

search(arrBook, 100, X, compareBook);
    //calling search function
```

The same search function now operators for Books just by writing a small compare class.

However, search function still works only for arrays. However the functionality of searching extends to list equally. To make it general for all containers(here array) we introduce a concept of iterators in our discussion.

Separating Containers

Iterators

Visualise iterators as an entity using which you can access the data within a container with certain restrictions.

These are classified into 5 categories

Input Iterator : An entity through which you can read from container and move ahead.

Que : What sort of container will posses an input iterator???

Sol : A keyboard.

Output Iterator An entity through which you can write into the container and move ahead.

Container like printer or monitor will have such an iterator.

Forward Iterator Iterator with functionalities of both Input and Output iterator in single direction.
Singly linked list will possess a forward entity since we can read/write only in the forward direction.

Bidirectional Iterator Forward iterator that can move in both the directions.

Doubly linked list will possess a bidirectional iterator.

Random Access Iterator Iterator that can read/write in both directions plus can also take jumps.

An array will have random access iterator. Since, you can jump by writing arr[5], which means jump to the 5th element.

Entity that does this, behaves like a pointer in some sense.

To write search function that is truly independent of data and the underlying container, we use iterator.

```
template<class ForwardIterator, class T, class Compare>
```

```
ForwardIterator search(ForwardIter beginOfContainer, ForwardIter endOfContainer,
T elementToSearch, Compare Obj) {
```

```
    while (beginOfContainer != endOfContainer) {
```

```
        if (Obj(*beginOfContainer), elementToSearch) == true) break; //Iterators are like  

        pointers!
```

```
        ++beginOfContainer;
```

```
}
```

```
    return beginOfContainer;
```

```
}
```

Here, beginOfContainer is a *ForwardIterator*, i.e., beginOfContainer must know how to read/write and move in *forward* direction.

So, if a container has at least *ForwardIterator*, the algorithm works. Hence, it works for list, doubly linked list and array as well thus achieving generality over container.

```
//search for book in an array
```

```
search(arr, arr + n, X, compareBook);
```

```
//search for book in a list
```

```
list<Book> lb; // see list
```

```
search(lb.begin(), lb.end(), X, compareBook);
```

```
//begin and end are member function of the class list.
```

Summary

1. Using templates, we achieve freedom from data types
2. Using comparators, we achieve freedom from operation(s) acting on data
3. Using iterators, we achieve freedom from underlying data structure (container).

Working with C++ STL Containers

C++ provides a powerful Standard Template Library(STL) template library, which is a set of C++ template classes to provide common programming data structures and functions.

- Algorithms**: There are inbuilt algorithms for tasks like sorting, searching etc. We will discuss these in the later part of the chapter.
- Containers**: Containers or container classes store objects and data. They are divided into following categories.

Sequence Containers: Implement data structures which can be accessed in a sequential manner.

- | | | | |
|--|----------|-----------|------------|
| (a) Vector | (b) List | (c) Deque | (d) Arrays |
| (e) Forward list (Introduced in C++11) | | | |

Container Adaptors: provide a different interface for sequential containers.

- | | | |
|-----------|--------------------|-----------|
| (f) queue | (g) priority queue | (h) stack |
|-----------|--------------------|-----------|

Associative Containers: Implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

- | | | | |
|---------|--------------|---------|--------------|
| (i) Set | (j) Multiset | (k) Map | (l) Multimap |
|---------|--------------|---------|--------------|

String

C++ provides a powerful alternative for the `char *`. It is not a built-in data type, but is a container class in the **Standard Template Library**. String class provides different string manipulation functions like concatenation, find, replace etc. Let us see how to construct a string type.

```
string s1; // s1 = ""
string s2 (s1); // s2 = "Hello"
string s3 (s1, 1, 2); // s3 = "el"
string s4 ("Hello World", 5); // s4 = "Hello"
string s5 (5, '*'); // s5 = "*****"
string s6 (s1.begin(), s1.begin() + 3); // s6 = "Hel"
```

Here are Some Member Functions :

- **append()**: Inserts additional characters at the end of the string. (can also be done using '+' or '+=' operator). Its time complexity is $O(N)$ where N is the size of the new string.
- **begin()**: Returns an iterator pointing to the first character. Its time complexity is $O(1)$.
- **clear()**: Erases all the contents of the string and assign an empty string ("") of length zero. Its time complexity is $O(1)$.
- **compare()**: Compares the value of the string with the string passed in the parameter and returns an integer accordingly. Its time complexity is $O(N + M)$ where N is the size of the first string and M is the size of the second string.

- **copy():** Copies the substring of the string in the string passed as parameter and returns the number of characters copied. Its time complexity is $O(N)$ where N is the size of the copied string.
- **empty():** Returns a boolean value, true if the string is empty and false if the string is not empty. Its time complexity is $O(1)$.
- **end():** Returns an iterator pointing to a position which is next to the last character. Its time complexity is $O(1)$.
- **erase():** Deletes a substring of the string. Its time complexity is $O(N)$ where N is the size of the new string.
- **find():** Searches the string and returns the first occurrence of the parameter in the string. Its time complexity is $O(N)$ where N is the size of the string.
- **insert():** Inserts additional characters into the string at a particular position. Its time complexity is $O(N)$ where N is the size of the new string.
- **length():** Returns the length of the string. Its time complexity is $O(1)$.
- **size():** Returns the length of the string. Its time complexity is $O(1)$.
- **substr():** Returns a string which is the copy of the substring. Its time complexity is $O(N)$ where N is the size of the substring.

Vector

Vectors are sequence containers that have dynamic size. In other words, vectors are dynamic arrays. Just like arrays, vector elements are placed in contiguous storage location so they can be accessed and traversed using iterators. To traverse the vector we need the position of the first and last element in the vector which we can get through `begin()` and `end()` or we can use indexing from 0 to `size()`.

```
vector<int> a; // empty vector of ints
vector<int> b (5, 10); // five ints with
                      // value 10
vector<int> c (b.begin(), b.end());
                      // iterating through second
vector<int> d (c); // copy of c
```

Some of the Member Functions of Vectors are :

- **at():** Returns the reference to the element at a particular position (can also be done using '[]' operator). Its time complexity is $O(1)$.
- **back():** Returns the reference to the last element. Its time complexity is $O(1)$.
- **begin():** Returns an iterator pointing to the first element of the vector.
- **empty():** It's time complexity is $O(1)$.
- **clear():** Deletes all the elements from the vector and assign an empty vector. Its time complexity is $O(N)$ where N is the size of the vector.

- ❑ **empty()** : Returns a boolean value, true if the vector is empty and false if the vector is not empty.
Its time complexity is O(1).
- ❑ **end()** : Returns an iterator pointing to a position which is next to the last element of the vector.
Its time complexity is O(1).
- ❑ **erase()** : Deletes a single element or a range of elements. It's time complexity is $O(N + M)$ where N is the number of the elements erased and M is the number of the elements moved.
- ❑ **front()** : Returns the reference to the first element. It's time complexity is O(1).
- ❑ **insert()** : Inserts new elements into the vector at a particular position. Its time complexity is $O(N + M)$ where N is the number of elements inserted and M is the number of the elements moved.
- ❑ **pop_back()** : Removes the last element from the vector. Its time complexity is O(1).
- ❑ **push_back()** : Inserts a new element at the end of the vector. Its time complexity is O(1).
- ❑ **resize()** : Resizes the vector to the new length which can be less than or greater than the current length. Its time complexity is O(N) where N is the size of the resized vector.
- ❑ **size()** : Returns the number of elements in the vector. Its time complexity is O(1).

List

List is a sequence container which takes constant time in inserting and removing elements. List in STL is implemented as Doubly Link List. The elements from List cannot be directly accessed. For example to access element of a particular position, you have to iterate from a known position to that particular position.

```
list <int> LI;  
list<int> LI(5, 100)  
//here LI will have 5 int elements of  
value 100
```

Some of the Member Function of List:

- ❑ **begin()** : It returns an iterator pointing to the first element in List. Its time complexity is O(1).
- ❑ **end()** : It returns an iterator referring to the theoretical element(doesn't point to an element) which follows the last element. Its time complexity is O(1).
- ❑ **empty()** : It returns whether the list is empty or not. It returns 1 if the list is empty otherwise returns 0. Its time complexity is O(1).
- ❑ **back()** : It returns reference to the last element in the list. Its time complexity is O(1).
- ❑ **assign()** : It assigns new elements to the list by replacing its current elements and change its size accordingly. Its time complexity is O(N).
- ❑ **erase()** : It removes a single element or the range of element from the list. Its time complexity is O(N).

- ❑ **front()** : It returns reference to the first element in the list. Its time complexity is O(1).
- ❑ **push_back()** : It adds a new element at the end of the list, after its current last element. Its time complexity is O(1).
- ❑ **push_front()** : It adds a new element at the beginning of the list, before its current first element. Its time complexity is O(1).
- ❑ **remove()** : It removes all the elements from the list, which are equal to given element. Its time complexity is O(N).
- ❑ **pop_back()** : It removes the last element of the list, thus reducing its size by 1. Its time complexity is O(1).
- ❑ **pop_front()** : It removes the first element of the list, thus reducing its size by 1. Its time complexity is O(1).
- ❑ **insert()** : It inserts new elements in the list before the element on the specified position. Its time complexity is O(N).
- ❑ **reverse()** : It reverses the order of elements in the list. Its time complexity is O(N).
- ❑ **size()** : It returns the number of elements in the list. Its time complexity is O(1).

Pair

Pair is a container that can be used to bind together two values which may be of different types. Pair provides a way to store two heterogeneous objects as a single unit.

```
pair<int, char> p1; // default
pair<int, char> p2(1, 'a'); // value initialization
pair<int, char> p3(p2); // copy of p2
```

We can also initialize a pair using `make_pair()` function. `make_pair(x, y)` will return a pair with first element set to `x` and second element set to `y`.

```
p1 = make_pair(2, 'b');
```

To access the elements we use keywords, `first` and `second` to access the first and second element respectively.

```
cout << p2.first << " " << p2.second << endl;
```

Set and Multiset

Sets are containers which store only unique values and permit easy lookups. The values in the sets are stored in some specific order (like ascending or descending). Elements can only be inserted or deleted, but cannot be modified. We can access and traverse set elements using iterators just like vectors.

Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.

In a multiset, the value of an element also identifies it (the value is itself the key, of type T). The value of the elements in a multiset cannot be modified once in the container (the elements are always constant), but they can be inserted or removed from the container.

```
set<int> s1; // Empty Set  
int a[] = {1, 2, 3, 4, 5, 5};  
set<int> s2 (a, a + 6); // s2 = {1, 2, 3, 4, 5}  
set<int> s3 (s2); // Copy of s2  
set<int> s4 (s3.begin(), s3.end()); // Set created using iterators  
multiset<int> first; // empty multiset of ints.  
int myints[] = {10, 20, 30, 20, 20};  
multiset<int> second (myints, myints + 5); // pointers used as iterators  
multiset<int> third (second); // a copy of second  
multiset<int> fourth (second.begin(), second.end()); //multiset created using iterators
```

Some of the Member Functions of Set are :

- **begin()** : Returns an iterator to the first element of the set. Its time complexity is O(1).
- **clear()** : Deletes all the elements in the set and the set will be empty . Its time complexity is O(N) where N is the size of the set.
- **count()** : Returns 1 or 0 if the element is in the set or not respectively. Its time complexity is O(logN) where N is the size of the set.
- **empty()** : Returns true if the set is empty and false if the set has at least one element. Its time complexity is O(1).
- **end()** : Returns an iterator pointing to a position which is next to the last element. Its time complexity is O(1).
- **Erase ()** : Deletes a particular element or a range of elements from the set. Its time complexity is O(N) where N is the number of element deleted.
- **Find ()** : Searches for a particular element and returns the iterator pointing to the element if the element is found otherwise it will return the iterator returned by end(). Its time complexity is O(logN) where N is the size of the set.
- **Insert ()** : insert a new element. Its time complexity is O(logN) where N is the size of the set.
- **size()** : Returns the size of the set or the number of elements in the set. Its time complexity is O(1).

```

Set Example :
#include<iostream>
#include<set>
using namespace std ;
int main(){
    //Create a Set - Set Stores unique entries in sorted order
    set<int> s;
    //Insert in Set
    s.insert(10);
    s.insert(12);
    s.insert(10);
    s.insert(3);
    s.insert(8);
    s.insert(12);
    //Deletion
    s.erase(12);
    //Searching
    auto f = s.find(3);
    if(f!=s.end()){
        cout<<"3 exists"<<endl; // 3 exists
    }
    //Iterating using a for each loop
    for(auto no:s){
        cout<<no<<" "; // 3 8 10
    }
}

```

Maps

Maps are containers which store elements by mapping their value against a particular key. It stores the combination of key value and mapped value following a specific order. Here key value are used to uniquely identify the elements mapped to it. The data type of key value and mapped value can be different. Elements in map are always in sorted order by their corresponding key and can be accessed directly by their key using bracket operator ([]).

In map, key and mapped value have a pair type combination, i.e both key and mapped value can be accessed using pair type functionalities with the help of iterators.

```
map <char,int> mp;  
mp['b']=1;
```

In map mp , the values will be in sorted order according to the key.

Some Member Functions of Map :

- **at()** : Returns a reference to the mapped value of the element identified with key. Its time complexity is $O(\log N)$.
- **Count()** : Searches the map for the elements mapped by the given key and returns the number of matches. As map stores each element with unique key, then it will return 1 if match is found otherwise return 0. Its time complexity is $O(\log N)$.
- **clear()** : Clears the map, by removing all the elements from the map and leaving it with its size 0. Its time complexity is $O(N)$.
- **begin()** : Returns an iterator(explained above) referring to the first element of map. Its time complexity is $O(1)$.
- **end()** : Returns an iterator referring to the theoretical element(doesn't point to an element) which follows the last element. Its time complexity is $O(1)$.
- **empty()** : Checks whether the map is empty or not. It doesn't modify the map. It returns 1 if the map is empty otherwise returns 0. Its time complexity is $O(1)$.
- **erase()** : Removes a single element or the range of element from the map.
- **find()** : Searches the map for the element with the given key, and returns an iterator to it, if it is present in the map otherwise it returns an iterator to the theoretical element which follows the last element of map. Its time complexity is $O(\log N)$.
- **insert()** : Insert a single element or the range of element in the map. Its time complexity is $O(\log N)$, when only element is inserted and $O(1)$ when position is also given.

Unordered Maps

Unordered maps fall under the subset of associative containers that use a pair of a key and a mapped value to store the corresponding elements. In an unordered map, the key value is usually used to uniquely identify the element, while the mapped value stores the content associated to this key. These data structures allow for fast retrieval of individual contained elements based on their mapped keys. Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).

Note : Maps and Unordered Maps have almost the same functions, but have different underlying implementations. We say unordered maps take $O(1)$ time for search, insert and erase in average case, hence are very useful.

	Map	Unordered_map
Element ordering	strict weak	n/a
common implementation	balanced tree or red-black tree	hash table
search time	$\log(n)$	$O(1)$ if no has collisions, upto to $O(n)$ if there are hash collisions, $O(n)$ when hash is same for any key
insertion time	$\log(n) + \text{rebalance}$	Same as search
deletion time	$\log(n) + \text{rebalance}$	Same as search
needs comparators	only operator <	only operator --
needs has function	no	yes
common use case	when good has is not possible or too slow. Or when order is required	In most other cases

- ❑ **find()** : Searches the container for an element with k as key and returns an iterator to it if found, otherwise it returns an iterator to `unordered_map::end`.
- ❑ **rehash()** : Sets the number of buckets in the container to n .
- ❑ **insert()** : inserts a new key-value pair into the container.
- ❑ **erase()** : Removes from the `unordered_map` container either a single element or a range of elements
- ❑ **count()** : This function returns 1 if an element with that key exists in the container, and zero otherwise.
- ❑ **load_factor()** : This function returns a floating value denoting current load factor in the `unordered_map` container.

$$\text{load_factor} = \text{current_size} / \text{bucket_count}$$

- ❑ **clear()** : Clears the map, by removing all the elements from the map and leaving it with its size 0. Its time complexity is $O(N)$.
- ❑ **begin()** : Returns an iterator(explained above) referring to the first element of map. Its time complexity is $O(1)$.
- ❑ **end()** : Returns an iterator referring to the theoretical element(doesn't point to an element) which follows the last element. Its time complexity is $O(1)$.
- ❑ **operator[]** : If k matches the key of an element in the container, the function returns a reference to its mapped value.

Example :

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

/*
class Fruit{
    price;
    color;
    sweetness;
    state;
    id;
    vendor;
}

int main(){
    unordered_map<string,int> h;
    //unordered_map<string,Fruit> h2;
    //Insertion
    h["Mango"] = 100;
    //Updation
    h["Mango"] = 80;
    //Print the value if Mango Exists
    if(h.count("Mango")!=0){
        cout<<h["Mango"]<<endl;
    }
    //Another Way to insert
    h.insert(make_pair("Kiwi",170));
    //Searching for a given fruit
    string f;
    cin>>f;
    if(h.count(f)){
        cout<<"Fruit costs"<<h[f]<<endl;
    }
    else{
        cout<<"Fruit doesn't exist";
    }
    //Deleting a Fruit(key)
    h.erase("Mango")
}
```

```

//Print all the elements
for(auto p:h){
    cout<<p.first<<" and "<<p.second<<endl;
}
return 0;
}

```

Stack

Stack is a container which follows the LIFO (Last In First Out) order and the elements are inserted and deleted from one end of the container. The element which is inserted last will be extracted first.

```
stack <int> s;
```

Some of the Member Functions of Stack are:

- ❑ **push()** : Insert element at the top of stack. Its time complexity is O(1).
- ❑ **pop()** : Removes element from top of stack. Its time complexity is O(1).
- ❑ **top()** : Access the top element of stack. Its time complexity is O(1).
- ❑ **empty()** : Checks if the stack is empty or not. Its time complexity is O(1).
- ❑ **size()** : Returns the size of stack. Its time complexity is O(1).

Queue

Queue is a container which follows FIFO order (First In First Out) . Here elements are inserted at one end (rear) and extracted from another end(front).

```
queue <int> q;
```

Some member function of Queues are:

- ❑ **push()** : Inserts an element in queue at one end(rear). It's time complexity is O(1).
- ❑ **pop()** : Deletes an element from another end if queue(front). It's time complexity is O(1).
- ❑ **front()** : Access the element on the front end of queue. It's time complexity is O(1).
- ❑ **empty()** : Checks if the queue is empty or not. It's time complexity is O(1).
- ❑ **size()** : Returns the size of queue. Its time complexity is O(1).

Priority Queue

A priority queue is a container that provides constant time extraction of the largest element, at the expense of logarithmic insertion. It is similar to the heap in which we can add element at any time but only the maximum element can be retrieved. In a priority queue, an element with high priority is served before an element with low priority.

```
priority_queue<int> pq;
```

To make a min-priority queue, declare priority queue as:

```
#include <functional> //for greater <int>
//min priority queue
priority_queue < int, vector < int >, greater < int > pq;
```

Some Member Functions of Priority Queues are :

- ❑ **empty()** : Returns true if the priority queue is empty and false if the priority queue has at least one element. Its time complexity is O(1).
- ❑ **pop()** : Removes the largest element from the priority queue. Its time complexity is O(logN) where N is the size of the priority queue.
- ❑ **push()** : Inserts a new element in the priority queue. Its time complexity is O(logN) where N is the size of the priority queue.
- ❑ **size()** : Returns the number of element in the priority queue. Its time complexity is O(1).
- ❑ **top()** : Returns a reference to the largest element in the priority queue. Its time complexity is O(1).

Example :

```
#include<iostream>
#include<queue>
#include<vector>
#include<functional>
#include<cstring>
using namespace std;
//To Compare Integers
class myComparison{
public:
    bool operator()(int a,int b){
        return a<b;
    }
};

class Person{
public:
    char name[20];
    int money;
```

```

Person(){
    name[0] = '\0';
    money = 0;
}
Person(char *n,int m){
    money = m;
    strcpy(name,n);
}
void print(){
    if(money>1000){
        cout<<name<<" is Rich"<<endl;
    }
}

```

DeQue

Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

```

deque<int> first; // empty deque of integer
deque<int> second(4,100); // four ints with value 100
deque<int> third(second.begin(),second.end()); // iterating through second
deque<int> fourth(third); // a copy of third

```

Some Member Functions of Deque are:

- assign()** : Assigns new contents to the deque container, replacing its current contents, and modifying its size Accordingly.
- at(n)** : Returns a reference to the element at position n in the deque container object.
- back()** : Returns a reference to the last element in the container.
- begin()** : Returns an iterator pointing to the first element in the deque container.
- empty()** : Returns whether the deque container is empty (i.e. whether its size is 0).
- end()** : Returns an iterator referring to the past-the-end element in the deque container.
- erase()** : Removes from the deque container either a single element (position) or a range of elements ((first,last)).
- front()** : Returns a reference to the first element in the deque container.
- pop_back()** : Removes the last element in the deque container, effectively reducing the container size by one.

- **pop_front()** : Removes the first element in the deque container, effectively reducing its size by one.
- **push_back()** : Adds a new element at the end of the deque container, after its current last element.
- **push_front()** : Inserts a new element at the beginning of the deque container, right before its current first element.
- **size()** : Returns the number of elements in the deque container.

Iterator

An iterator is any object that, points to some element in a range of elements (such as an array or a container) and has the ability to iterate through those elements using a set of operators (with at least the increment (++) and dereference (*) operators).

For Vector:

```
vector <int>::iterator it;
```

For List:

```
list <int>::iterator it;
```

etc....

TIME TO TALK ABOUT ALGORITHMS !

<algorithm>

The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.

binary_search(first,last,val)

Returns true if any element in the range [first,last) is equivalent to val, and false otherwise.

```
binary_search (v.begin(), v.end(), 3))  
//v is a vector
```

find(first,last,val)

Returns an iterator to the first element in the range [first,last) that compares equal to val. If no such element is found, the function returns last.

```
it = find (myvector.begin(), myvector.end(), 30); //it is an iterator
```

lower_bound(first,second,val)

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than val.

```
it = lower_bound (v.begin(), v.end(), 20); //v is a vector
```

upper_bound(first,second,val)

Returns an iterator pointing to the first element in the range [first,last) which compares greater than val.

`it = upper_bound (v.begin(), v.end(), 20); //v is a vector`

max(a,b)

Returns the largest of a and b. If both are equivalent, a is returned.

`cout << max(a,b);`

min(a,b)

Returns the smallest of a and b. If both are equivalent, a is returned.

`cout << min(a,b);`

reverse(first,last)

Reverses the order of the elements in the range [first,last).

`reverse(myvector.begin(),myvector.end());`

rotate(first,middle,last)

Rotates the order of the elements in the range [first,last), in such a way that the element pointed by middle becomes the new first element.

`rotate(myvector.begin(),myvector.begin()+3,myvector.end());`

sort(first,last)

Sorts the elements in the range [first,last) into ascending order.

`sort(v.begin(),v.end());`

`sort(a,a+n);`

`sort(v.begin(),v.end(),comparator);`

`sort(a,a+n,comparator);`

swap(a,b)

Exchanges the values of a and b.

`swap(a,b);`

next_permutation(first,last)

Rearranges the elements in the range [first,last) into the next lexicographically greater permutation.

`next_permutation(v.begin(),v.end());`

SOME EXAMPLES

1. Sort Game, HackerBlocks

(#sorting, #vectors, #pairs)

Sanju needs your help! He gets a list of employees with their salary. The maximum salary is 100. Sanju is supposed to arrange the list in such a manner that the list is sorted in decreasing order of salary. And if two employees have the same salary, they should be arranged in lexicographical manner such that the list contains only names of those employees having salary greater than or equal to a given number x .

Help Sanju prepare the same!

Input : On the first line of the standard input, there is an integer x . The next line contains integer N , denoting the number of employees. N lines follow, which contain a string and an integer, denoting the name of the employee and his salary.

79

4

Eve 78

Bob 99

Suzy 86

Alice 86

Output : Bob 99

Alice 86

Suzy 86

Solution :**Code :**

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;
bool myCompare(pair<string,int> p1,pair<string,int> p2){
    ///first = Name, second = Salary
    /// Preference Salary > Name
    if(p1.second==p2.second){
        return p1.first < p2.first;
    }
    return p1.second > p2.second;
}
void Sort(emp[]){
    sort(emp.begin(),emp.end(),myCompare);
}
```

```

        for(int i=0;i<n;i++){
            if( myCompare(emp[i],emp[i+1])){
                swap(emp[i],emp[i+1]);
            }
        }
    }

int main(){
    int min_salary,n;
    pair<string,int> emp[100005];
    cin>>min_salary;
    cin>>n;
    string name;
    int salary;
    for(int i=0;i<n;i++){
        cin>>name>>salary ;
        emp[i].first = name;
        emp[i].second = salary;
    }
    sort(emp,emp+n,myCompare);
    //Print
    for(int i=0;i<n;i++){
        if(emp[i].second>=min_salary){
            cout<<emp[i].first <<" "<<emp[i].second<<endl;
        }
    }
    return 0;
}

```

2. ArraySub, Spoj

(#sliding-window, #deque)

Given an array and an integer k, find the maximum for each and every contiguous subarray of size k

Input : The number n denoting number of elements in the array then after a new line we have the numbers of the array and then k in a new line

9

1 2 3 1 4 5 2 3 6

3

Output: 3 3 4 5 5 5 6

Solution :

Code :

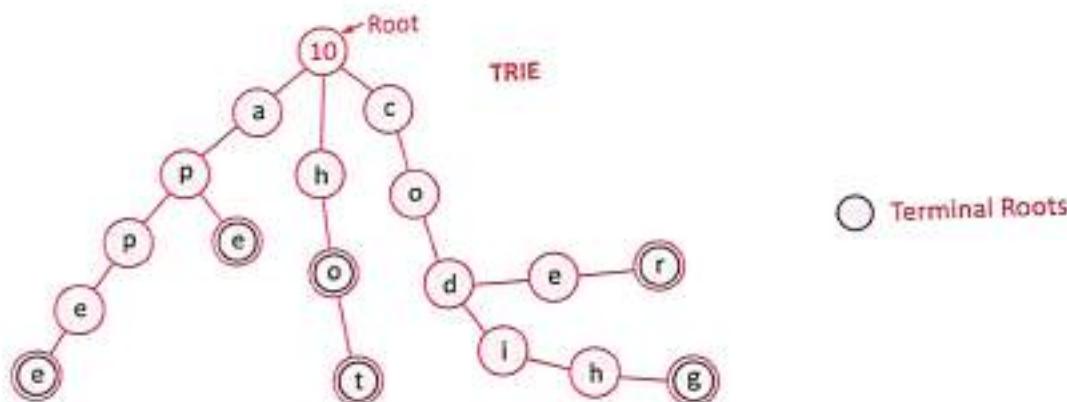
```
#include<iostream>
#include<cstdio>
#include<deque>
using namespace std;
int a[1000001];
int main(){
    int n,k,i;
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    scanf("%d",&k);
    deque<int> Q(k);
    for(i=0;i<k;i++)
    {
        while(!Q.empty()&&a[i]>=a[Q.back()])
            Q.pop_back();
        Q.push_back(i);
    }
    for(;i<n;i++)
    {
        printf("%d ",a[Q.front()]);
        while((!Q.empty())&&(Q.front()<=i-k))
            Q.pop_front();
        while(!Q.empty()&&a[i]>=a[Q.back()])
            Q.pop_back();
        Q.push_back(i);
    }
    printf("\n%d",a[Q.front()]);
    return 0;
}
```

Some more Data Structures

Data Structures like Trie, Graphs are not directly available in STL but can be implemented easily using other data structures.

Trie Data Structure :

1. Trie is an information retrieval data structure.
2. It is also called radix/prefix tree.
3. It is used for efficient searching of keys in the container. If the keys are strings, then a particular string can be searched in $O(n)$ length where n denotes the length of the string to be searched.
4. Each node of the trie has multiple branches; a node where a word ends is marked with is Terminal = true.



```
#include<iostream>
#include<unordered_map>
using namespace std;
#define hashmap unordered_map<char,node*>
class node{
public:
    char data;
    hashmap h;
    bool isTerminal;
    node(char d){
        data = d;
        isTerminal = false;
    }
};
class Trie{
```

Standard Template Library

```
node *root;
public:
    Trie(){
        root = new node('\0');
    }
    void addWord(char *word){
        node *temp = root;
        for(int i=0;word[i]!='\0';i++){
            char ch = word[i];
            if(temp->h.count(ch)==0){
                node * child = new node(ch);
                temp->h[ch] = child;
                temp = child;
            }
            else{
                temp = temp->h[ch];
            }
        }
        temp->isTerminal = true;
    }
    bool search(char *word){
        node *temp = root;
        for(int i=0;word[i]!='\0';i++){
            char ch = word[i];
            if(temp->h.count(ch)){
                temp = temp->h[ch];
            }
            else{
                return false;
            }
        }
        cout<<temp->data<<" ";
    }
    return temp->isTerminal;
}
```

```

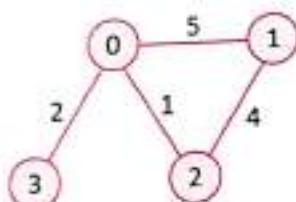
};

int main(){
    char word[10][100] = {"apple","ape","coder","coding blocks","no"};
    Trie t;
    for(int i=0;i<5;i++){
        t.addWord(word[i]);
    }
    char searchWord[100];
    cin.getline(searchWord,100);
    if(t.search(searchWord)){
        cout<<searchWord<<" found "<<endl;
    }
    else{
        cout<<"not found !"<<endl;
    }
    return 0;
}

```

Graph Data Structure

An adjacency list implementation of graph can be easily represented using a vector. Example of DFS traversal on weighted graph.



```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
vector < pair < int,int > > graph[100005]; //graph ( using array of vectors >
int visited[100005]; //visited array
void dfs(int cur)
{
    //dfs method
    if(visited[cur])
        return;
    visited[cur] = 1;
    for(int i=0;i<graph[cur].size();i++)
        dfs(graph[cur][i].first);
}

```

```
cout<<cur<<" "; //printing current node
visited[cur] = 1; //setting current node as visited
for(int i=0;i<graph[cur].size();++i)
{
    dfs(graph[cur][i].first);
}
int main()
{
    int n,m;
    int a,b,w;
    cin>>n>>m;
    for(int i=0;i<m;++i)
    {
        cin>>a>>b>>w; //edge between a and b with weight w
        graph[a].push_back(make_pair(b,w));
        graph[b].push_back(make_pair(a,w));
    }
    cin>>a;
    dfs(a);
    return 0;
}
```

SELF STUDY NOTES

2 Mathematics

Birthday Paradox - Warmup Problem!

What is the minimum number of people that should be present in a room so that there's 50% chance of two people having same birthday ? Probability that 2 people in the room will have same birthday

In a room of just ___ people there's a 50-50 chance of two people having the same birthday. In a room of ___ there's a 99.9% chance of two people matching.

HINT :

If there are two people in a room, Probability that two will have same birthday

$$= 1/365 = 0.00274 = 0.274\%$$

Probability that two will have different birthdays = $1 - (\text{probability that two have same birthday}) = 1 - 0.00274 = 0.9973 = 99.73\%$. Now take your time and think about the approach.

SOLUTION:

23 for 50% probability

70 for 99.9% probability

Code:

```
#include<iostream>
using namespace std;
int main(){
    float p = 1;
    //p denotes prob of 2 ppl having different birthday
    // same bday = 1 - p
    float num = 365;
    float denom = 365;
    int people = 0;
    while(p>0.5){
        p *= (num/denom);
        num--;
        people++;
    }
}
```

```
    }
    return 0;
}
```

Types of Problems in Mathematics

- Adhoc/Formula Based/Brute Force
- Big Integers
- Exponentiation
- Number Systems/Series
- Pigeonhole Principle
- Inclusion-Exclusion Principle
- Probability & Expectation
- Combinatorics

ADHOC/ BRUTE FORCE/ COMPLETE SEARCH

These are relatively simpler problems based upon some formula or complete search.

Let us see one example.

German Lotto :

In the German Lotto you have to select 6 numbers from the set {1,2,...,49}. A popular strategy to play Lotto - although it doesn't increase your chance of winning — is to select a subset S containing k ($k > 6$) of these 49 numbers, and then play several games with choosing numbers only from S .

Input :

For example, for $k = 8$ and $S = \{1; 2; 3; 5; 8; 13; 21; 34\}$ there are 28 possible games: [1,2,3,5,8,13],

Output :

Your job is to write a program that reads in the number k and the set S and then prints all possible games choosing numbers only from S .

[1,2,3,5,8,21], [1,2,3,5,8,34], [1,2,3,5,13,21], ..., [3,5,8,13,21,34].

Solution :

Brute Force, Sort the array and use 6 Loops to pick all possible combinations 6 numbers.

Code :

```
#include <iostream>
using namespace std;
int main() {
    //Numbers from 1 to 49
    //Choose a subset of 6 Numbers
    int a[] = {1,2,4,5,6,7,8,10,12}; //assuming the array after sorting it
    int n = sizeof(a)/sizeof(int);
    for(int i=0;i<n-5;i++){
        for(int j=i+1;j<n-4;j++){
            for(int k=j+1;k<n-3;k++){
                for(int l=k+1;l<n-2;l++){
                    for(int m = l+1;m<n-1;m++){
                        for(int o= m+1;o<n;o++){
                            cout<<a[i]<<","<<a[j]<<","<<a[k]<<","<<a[l]<<","<<a[m]<<","<<a[o]<<endl;
                        }
                    }
                }
            }
        }
    }
    return 0;
}
```

Big Integers

Problems involving big integers are quite common in online competitions. In Java, Python it is easy to work with big integers but in C++ it's difficult because the *long long int* datatype can store only at max 18 digits.

So, for problems involving Big Numbers(containing 100's of digits) we either use *Java Big Integer Class* or Python or we use Arrays in C++ ! Let us see one example.

Note : There is a BOOST C++ Library which allows us to work with big integers as well.

Computing Large Factorials in C++

```
Code :  
  
#include<iostream>  
using namespace std;  
void multiply(int *a,int &n,int no){  
    int carry = 0;  
    for(int i=0;i<n;i++){  
        int product = a[i]*no + carry;  
        a[i] = product%10;  
        carry = product/10;  
    }  
    while(carry){  
        a[n] = carry%10;  
        carry = carry/10;  
        n++;  
    }  
}  
void big_factorial(int number){  
    //Assuming max 1000 digits  
    int *a = new int[1000]{0};  
    a[0] = 1;  
    int n = 1; //n denotes the array index  
    for(int i=2;i<=number;i++){  
        multiply(a,n,i);  
    }  
    for(int i=n-1;i>=0;i--){  
        cout<<a[i];  
    }  
    cout<<endl;  
}  
int main(){  
    big_factorial(100);  
    return 0;  
}
```

The Java BigInteger Class

In Java, the BigInteger class is very powerful and supports lots of operations on big numbers (having 100's of digits) like :

- | | | | |
|-----------|---------------------------|-----------|---------------------------------|
| 1. | Modular Arithmetic | 2. | Base Conversion |
| 3. | GCD Calculation | 4. | Power Calculation |
| 5. | Prime Generation | 6. | Bit-masking, Bitwise Operations |
| 7. | Other Miscellaneous Tasks | | |

It is important to learn about this class, to make our work easy in Programming Contests

Examples :

Code :

```
import java.math.BigInteger;
import java.util.Scanner;
public class Main{
    static void playWithInt(){
        String s;
        Scanner sc = new Scanner(System.in);
        String s1 = sc.next();
        String s2 = sc.next();
        //The second parameter denotes the base
        BigInteger one = new BigInteger(s1,2);
        BigInteger two = new BigInteger(s2,2);
        System.out.println(one);
        System.out.println(two);
        //Number of Set Bits
        System.out.println(one.bitCount());
        //Number of total bits
        System.out.println(one.bitLength());
        //To add we use add()
        one = one.add(two);
        //To multiply
        one = one.multiply(two);
        System.out.println(one);
```

```
//Computing Factorial  
//Computing GCD  
BigInteger b1 = new BigInteger("15");  
BigInteger b2 = new BigInteger("6");  
System.out.println(b1.gcd(b2));  
System.out.println(b1.add(b2));  
System.out.println(b1.multiply(b2));  
//Next probable prime - Generates the next available prime  
BigInteger b3 = new BigInteger("25");  
System.out.println(b3.nextProbablePrime());  
//Power Function  
BigInteger b4 = new BigInteger("3");  
System.out.println(b4.pow(5));  
//value of - Int/Long Int to Big Integer  
BigInteger b5 = BigInteger.valueOf(100);  
System.out.println(b5);  
//Base Conversion, interprets 1001 in base 2  
BigInteger b6 = new BigInteger("1001",2);  
System.out.println(b6);  
}  
public static void main(String [] args){  
    playWithInt();  
}  
}
```

Factorial of Big Number in Java

Code :

```
import java.math.BigInteger;  
import java.util.Scanner;  
public class Main {  
    static BigInteger fact(int N){  
        BigInteger b = new BigInteger("1");
```

```

        for(int i=2;i<=N;i++){
            b = b.multiply(BigInteger.valueOf(i));
        }
        return b;
    }

    public static void main(String args[]) {
        int N = 100;
        System.out.println(fact(N));
    }
}

Python Code for Factorial :
def fact(n):
    ans = 1
    for i in range(1,n+1):
        ans = ans*i
    return ans
print fact(100)

```



Time to Try

Problem - *Klaudia and Natalia have 10 apples together, but Klaudia has two apples more than Natalia. How many apples does each of the girls have?*

Julka said without thinking: Klaudia has 6 apples and Natalia 4 apples. The teacher tried to check if Julka's answer wasn't accidental and repeated the riddle every time increasing the numbers. Every time Julka answered correctly. The surprised teacher wanted to continue questioning Julka, but with big numbers she couldn't solve the riddle fast enough herself. Help the teacher and write a program which will give her the right answers.

Problem Statement : <http://www.spoj.com/problems/JULKA/>

Solution : <http://cb.lk/code/JULKA>

Number Series and Sequences

- Most of the sequences are based upon some formula or some recurrence.
- The sequence may contain - AP, GP, HP, Polynomial Sequence, Linear Recurrence etc.
- Other commonly used sequences are - Fibonacci Sequence, Binomial Series, Catalan Numbers etc.

OEIS - Online Sequence Finder !!

You can always refer <https://oeis.org/> to find out any sequence and its formula. So You only need to generate output for small inputs and then search for that sequences at OEIS (The Online Encyclopedia of Integer Sequences) ! Try to search it for following sequences.

Try Searching these on OEIS

Example 1 : 1, 2, 6, 24

Example 2 : 1, 2, 6, 10

Example 3 : 1, 2, 5, 14, 42, 132, 429

Binomial Coefficients

Binomial Coefficient " C_k^n " denotes the number of ways of selecting k items from n items.

$$C(n,k) = n! / (n-k)! k!$$

Computing $C(n,k)$ becomes difficult when n and k are large. We might prefer to use dynamic programming or Pascal's Triangle to Compute some are all values of $C(n, k)$

$$C(n, k) = C(n-1, k) + C(n-1, k-1)$$

Using this formula we can also build the Pascal's Triangle in a bottom up way

```
1  
1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1  
1 5 10 10 5 1  
.  
.
```

And so on, Binomial Coefficients are frequently used in problems involving Combinatorics.

Catalan Numbers (Very Important Series)

Let's us start with one example.

How many ways are there to construct a Binary Search Tree with ' n ' nodes numbered from 1 to N ?

Hint:

Make every possible i^{th} node as the root node and recursively count for number of BST's in its left half and right-half. Do it for every value of i ($1 \leq i \leq n$) and sum it up.

The formula generating after adding the series is the n^{th} Catalan Number !!

It is defined using binomial coefficient notation nC_k as :

$$\text{Cat}(n) = {}^{2n}C_n / (n+1)$$

$$\text{Cat}(0) = 1$$

Using the above formula , the first few terms of series are :

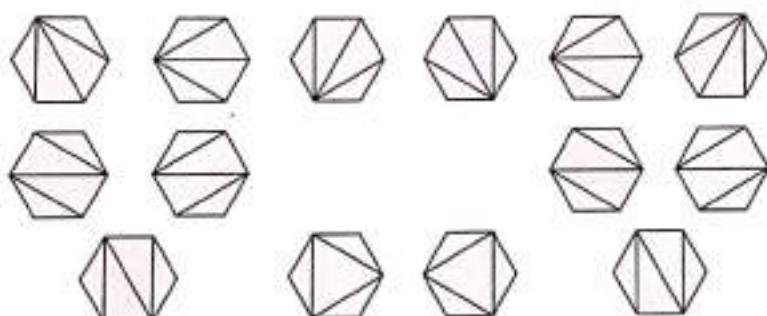
1, 1, 2, 5, 14, 42, 132, 429, 1430

Another Recursive Formula is :

$$C_0 = 1 \text{ and } C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \text{ for } n \geq 0;$$

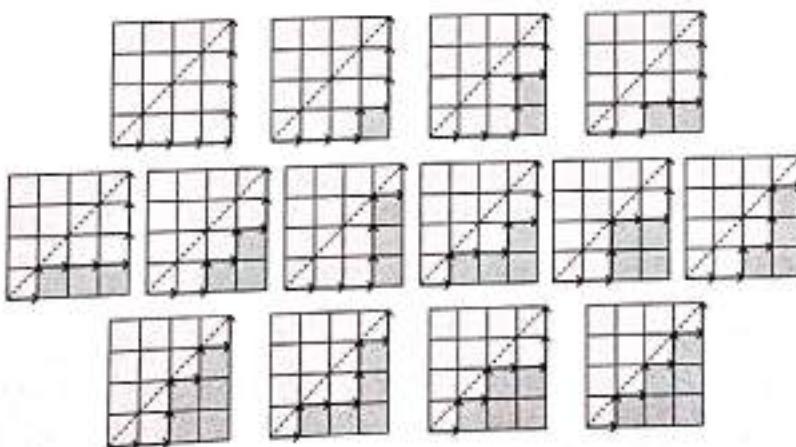
Applications of Catalan Numbers :

- Number of possible Binary Search Trees with n keys.
- Number of expressions containing n pairs of parentheses which are correctly matched. For $n = 3$, possible expressions are $((())), ((()()), ()()), (())(), ()())$
- Number of Ways $n + 1$ factors can be completely parenthesized, for e.g. $N = 3$ and $3 + 1$ factors : {a, b, c, d}, we have: $(ab)(cd)$, $a(b(cd))$, $((ab)c)d$, $(a(bc))(d)$ and $a((bc)d)$.
- Number of ways a convex polygon of $n+2$ sides can split into triangles by connecting vertices.



- Number of different Unlabelled Binary Trees can be there with n nodes

The number of paths with $2n$ steps on a rectangular grid from bottom left, i.e., $(n-1, 0)$ to top right $(0, n-1)$ that do not cross above the main diagonal.



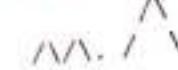
- Given n horizontal strokes below the horizon.

 $n = 0$ 

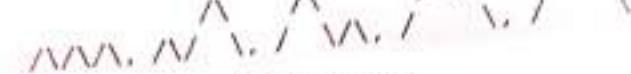
1 way

 $n = 1$ 

1 way

 $n = 2$ 

2 ways

 $n = 3$ 

5 ways

Mountain Ranges

strokes and n down-strokes that all stay below the horizon is that the mountains will never go

Solving Linear Recurrences

The problem is generally asking you the n -th term of a linear recurrence. It is possible to solve with dynamic programming if n is small, problem arises when n is very large.

Linear Recurrence :

A linear recurrence relation is a function or a sequence such that each term is a linear combination of previous terms. Each term can be described as a function of the previous terms.

A famous example is the Fibonacci sequence: $f(i) = f(i - 1) + f(i - 2)$. Linear means that the previous terms in the definition are only multiplied by a constant (possibly zero) and nothing else. So, this sequence: $f(i) = f(i - 1) * f(i - 2)$ is not a linear recurrence.

Problem :

Given f , a function defined as a linear recurrence relation. Compute $f(N)$. N may be very large.

How to Solve ?

Break the problem in four steps. Fibonacci sequence will be used as an example

Step 1 : Determine K , the number of terms on which $f(i)$ depends

More precisely, K is the minimum integer such that $f(i)$ doesn't depend on $f(i - M)$, for all $M > K$.

For Fibonacci sequence, because the relation is: $f(i) = f(i - 1) + f(i - 2)$, therefore, $K = 2$.

In this way, be careful for missing terms though, for example, this sequence:

$f(i) = 2f(i - 2) + f(i - 4)$ has $K = 4$,

because it can be rewritten explicitly as: $f(i) = 0f(i - 1) + 2f(i - 2) + 0f(i - 3) + 1f(i - 4)$.

Step 2 : Determine the F_1 vector the initial values

If each term of a recurrence relation depends on K previous terms, then it must have the first K terms defined, otherwise the whole sequence is undefined. For Fibonacci sequence ($K = 2$), the well-known initial values are:

$$f(1) = 1$$

$$f(2) = 1$$

Note: We are indexing Fibonacci from 1, $f(0) = 0$.

$$F_1 = \begin{bmatrix} f(1) \\ f(2) \\ \vdots \\ f(k) \end{bmatrix}$$

We define a column vector F_i as a $K \times 1$ matrix whose first row is $f(i)$, second row is $f(i + 1)$, and so on, until K -th row is $f(i + K - 1)$. The initial values of f are given in column vector F_1 that has values $f(1)$ through $f(K)$:

Step 3 : Determine T , the transformation matrix. Construct a $K \times K$ matrix T , called transformation matrix, such that

$$TF_1 = F_i + 1$$

Suppose,

$$f(i) = c_1 f(i-1) + c_2 f(i-2) + c_3 f(i-3) + \dots + c_k f(i-k)$$

$$f(i) = \sum_{j=1}^k c_j f(i-j)$$

Putting $i = k + 1$

$$f(k+1) = c_1 f(k) + c_2 f(k-1) + c_3 f(k-2) + \dots + c_k f(1)$$

Hence, the transformation matrix is:

$$T = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}$$

Example For Fibonacci :

$$c_1 = 1, c_2 = 1$$

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Step 4: Determine the recurrence relation

$$F_2 = T F_1 = T^2 F_1$$

$$F_3 = T^{n-1} F_1$$

Therefore, the original problem is now (almost) solved: compute F_N as above, and then we can obtain $f(N)$: it is exactly the first row of F_N . In case of our Fibonacci sequence, the N -th term in Fibonacci sequence is the first row of:

$$\begin{bmatrix} 0 & 1 \end{bmatrix}^{N-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Use Fast Exponentiation!

To compute T^{N-1} use exponentiation by squaring method that works in $O(\log N)$ time, with this recurrence:

- $A^p = A$, if $p = 1$,
- $A^p = A * A^{p-1}$, if p is odd
- $A^p = X^2$, where $X = A^{p/2}$, otherwise.

Multiplying two matrices takes $O(K^3)$ time using standard method, so the overall time complexity to solve a linear recurrence is $O(K^3 \log N)$.

RECURSIVE SEQUENCE (SPOJ)

<http://www.spoj.com/problems/SEQ/>

Sequence (a_i) of natural numbers is defined as follows:

$$a_i = b_i \text{ (for } i \leq k\text{)}$$

$$a_i = c_1 a_{i-1} + c_2 a_{i-2} + \dots + c_k a_{i-k} \text{ (for } i > k\text{)}$$

where b_j and c_j are given natural numbers for $1 \leq j \leq k$. Your task is to compute a_n for given n and output it modulo 10^9 .

Solution:**Code :**

```
#include <iostream>
#include <vector>
using namespace std;
#define ll long long
#define MOD 1000000000
```

```

ll k;
vector<ll> a,b,c;
//Multiply two matrices
vector<vector<ll>> multiply(vector<vector<ll>> A,vector<vector<ll>> B){
    //third matrix mei result store
    vector<vector<ll>> C(k+1,vector<ll>(k+1));
    for(int i=1;i<=k;i++){
        for(int j=1;j<=k;j++){
            for(int x=1;x<=k;x++){
                C[i][j] = (C[i][j] + (A[i][x]*B[x][j])%MOD)%MOD;
            }
        }
    }
    return C;
}
vector<vector<ll>> pow(vector<vector<ll>> A,ll p){
    //Base case
    if(p==1){
        return A;
    }
    //Rec Case
    if(p&1){
        return multiply(A, pow(A,p-1));
    }
    else{
        vector<vector<ll>> X = pow(A,p/2);
        return multiply(X,X);
    }
}
ll compute(ll n){
    //Base case
    if(n==0){
        return 0;
    }
}

```

```

//Suppose n<=k
if(n<=k){
    return b[n-1];
}
//Otherwise we use matrix exponentiation, indexing 1 se
vector<ll> F1(k+1);
for(int i=1;i<=k;i++){
    F1[i] = b[i-1];
}
//2. Transformation matrix
vector<vector<ll>> T(k+1,vector<ll>(k+1));
// Let init T
for(int i=1;i<=k;i++){
    for(int j=1;j<=k;j++){
        if(i<k){
            if(j==i+1){
                T[i][j] = 1;
            }
            else{
                T[i][j] = 0;
            }
            continue;
        }
        //Last Row - store the Coefficients in reverse order
        T[i][j] = c[k-j];
    }
}
// 3. T^n-1
T = pow(T,n-1);
// 4. multiply by F1
ll res = 0;
for(int i=1;i<=k;i++){
    res = (res + (T[1][i]*F1[i])%MOD)%MOD;
}

```

```

        return res;
    }

int main() {
    ll t,n,num;
    cin>>t;
    while(t--){
        cin>>k;
        //Init Vector F1
        for(int i=0;i<k;i++){
            cin>>num;
            b.push_back(num);
        }
        //Coefficients
        for(int i=0;i<k;i++){
            cin>>num;
            c.push_back(num);
        }
        // the value of n
        cin>>n;
        cout<< compute(n)<<endl;
        b.clear();
        c.clear();
    }
    return 0;
}

```

Variation :

The recurrence relation may include a constant i.e., the function is of the form

$$f(i) = \sum_{j=1}^k c_j f(i-j) + d$$

In this variant, the F vector is enhanced to remember the value of d.

It is of size $(K + 1) \times 1$ now :

$$\mathbf{F}_i = \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \\ d \end{bmatrix}$$

We now need to construct the T matrix, of size $(K \times 1)(K \times 1)$ such that

$$T\mathbf{F}_i = \mathbf{F}_{i+1}$$

$$[T] \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \\ d \end{bmatrix} \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \\ d \end{bmatrix}$$

Hence, the transformation matrix is :

$$T = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_K & c_{K-1} & c_{K-2} & c_{K-3} & \cdots & c_1 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$



TIME TO TRY

- Generate the Transformation matrix for the given sequence.

$$f(i) = 2f(i-1) + 3f(i-2) + 5$$

- Generate the Transformation matrix for the given sequences and write code to compute nth term.

$$f(i) = f(i-1) + 2i^2 + 3i + 5$$

$$f(i) = f(i-1) + 2i^2 + 5$$

- Fibonacci Number (HackerBlocks) :**

Write an efficient code to compute nth Fibonacci Number where $N \leq 10^{18}$.

- Recursive Sequence - Version-II (Spoj) :**

Read the problem statement at Spoj.

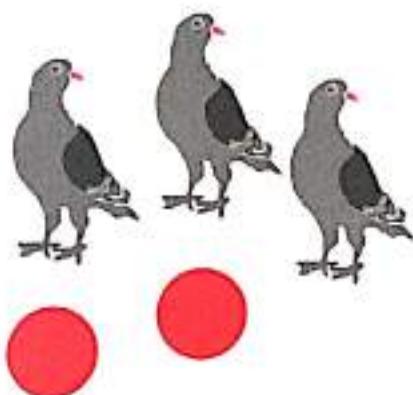
<http://www.spoj.com/problems/SPP/>

5. Fast Ladders (HackerBlocks) :

Given a ladder of containing N steps, a person standing at the foot of the ladder can take at max a jump of K steps at every point. Find out the number of ways to reach the top of that Ladder.

Fast Tiling Problem (HackerBlocks) :

Given n , and grid of size $4 \times n$, you have to find out the number of ways of filling the grid using 1×4 tiles.

Pigeonhole Principle

The pigeonhole principle is a fairly simple idea to grasp. Say that you have 7 pigeons and 6 pigeonholes. So, now you decide to start putting the pigeons one by one into each pigeonhole.

$|p| |p| |p| |p| |p| |p| |p|$

So, now, you have one pigeon left, and you can put it into any of the pigeonholes.

$|pp| |p| |p| |p| |p| |p| |p|$

The point is that when the **number of pigeons > number of pigeonholes**, there will be at least one pigeonhole with at least two pigeons.

Hair counting problem :

If the amount of hair is expressed in terms of the number of hair strands, the average human head has about 150,000 hair strands. It is safe to assume, then, that no human head has more than 1,000,000 strands of hair. Since the population of Delhi is more than 1,000,000 at least two people in Delhi have the same amount of hair.

EXAMPLES :**1. Divisible Subset(Codechef) :**

To find a non-empty subset of the given multiset with the sum of elements divisible by the size of original multiset.

<https://www.codechef.com/problems/DIVSUBS>

How to use Pigeonhole Principle ?

$$a \% N = x$$

$$b \% N = x$$

Then, $(b - a) \% N = (b \% N - a \% N) = (x - x) = 0$

Let's denote $a_1 + a_2 + \dots + a_k$ by b_k . So, we obtain:

$$b_0 = 0$$

$$b_1 = a_1$$

$$b_2 = a_1 + a_2$$

.

$$\dots$$

$$b_n = a_1 + a_2 + a_3 + a_4 + \dots + a_N$$

$$\text{So, } a_L + a_{L+1} + \dots + a_R = b_R - b_{L-1}$$

Therefore, if there are two values with equal residues modulo N among b_0, b_1, \dots, b_n then we take the first one for L-1 and the second one for R and the required subsegment is found.

There are $N + 1$ values of b_i and N possible residues for N. So, according to the pigeonhole principle the required subsegment will always exist.

2. The Gray Similar Code(Codechef) :

Given 'N' 64 bit integers such that any two successive numbers differ at exactly '1' bit. We have to find out 4 integers such that their XOR is equal to 0.

<https://www.codechef.com/problems/GRAYSC>

Hint: If we take XOR of any two successive numbers, we will get a number with only 1 set bit and all others will be 0.

How to use Pigeonhole Principle ?

For $N = 130$, we have '65' pairs i.e. $\{X_1, X_2\}, \{X_3, X_4\}, \{X_5, X_6\} \dots \{X_{129}, X_{130}\}$. But there exists only 64 possible position for the set bit '1', by pigeonhole principle at least two bits will be set at same positions say $\{X_i, X_{i+1}\}$ and $\{X_j, X_{j+1}\}$. If we take x or y of pair of these four numbers, we will get 0.

Thus, by pigeonhole principle for all $n \geq 130$, we will always find 4 integers such that their XOR is 0. For $n < 130$, we can iterate for 3 values of $A[i], A[j], A[k]$ and do a binary search to find 4th number which is $(A[i] \wedge A[j] \wedge A[k])$

3. Holiday Accommodation (Spoj) - Graph + Pigeonhole :

Given a weighted tree, consider there are N people in N nodes. You have to rearrange these N people such that everyone is in a new node, and no node contains more than one person under the constraint that the distance travelled for each person must be maximized. There are N cities having $N-1$ highways connecting them.

<http://www.spoj.com/problems/HOLI/>

HINT: In order to maximize cost:

- All edges will be used to travel around.
- We need to maximize the use of every edge used. Once we know how many time each edge is used, we can calculate the answer.

How to apply Pigeonhole principle ?

Now for any edge E_i , we can partition the whole tree into two subtrees, if one side has n nodes, the other side will have $N - n$ nodes. Also, note that, $\min(n, N-n)$ people will be crossing the edge from each side. Because if more people cross the edge, then by pigeon-hole principle in one side, we will get more people than available node which is not allowed in the problem statement. So, E_i will be used a total of $2 * \min(n, N-n)$ times.

$$\text{cost} = \sum 2 * \min(n_i, N - n_i) * \text{weight}(E_i)$$

for every edge E_i

Code :

```
#include<bits/stdc++.h>
using namespace std;
class Graph{
    int V;
    list<pair<int,int> > *l;
public:
    Graph(int v){
        V = v;
        l = new list<pair<int,int> >[V];
    }
    void addEdge(int u,int v,int cost,bool bidir=true){
        l[u].push_back(make_pair(v,cost));
        if(bidir){
            l[v].push_back(make_pair(u,cost));
        }
    }
    int dfsHelper(int node,bool *visited,int *count,int &ans){
        visited[node] = true;
        count[node] = 1;
        for(auto neighbour:l[node]){
            int v = neighbour.first;
            if(!visited[v]){
                ans += 2 * min(count[v], V - count[v]) * neighbour.second;
                dfsHelper(v,visited,count,ans);
            }
        }
    }
}
```

```

        ,visited,count,ans);
        count[v]) * neighbour.second;
    }
}
return count[node];
}
int dfsMain(){
    bool *visited = new bool[V]{0};
    int *count = new int[V]{0};
    int ans = 0 ;
    dfsHelper(0,visited,count,ans);
    return ans;
}
int main(){
    Graph g(4);
    g.addEdge(0,1,3);
    g.addEdge(1,2,2);
    g.addEdge(3,2,2);
    cout<<g.dfsMain();
}

```



TRY IT YOURSELF !

1. Divisible Subarrays(HackerBlocks)

Find the number subarrays of the given multiset with the sum of elements divisible by the size of original multiset in linear time.

The Inclusion-Exclusion Principle

Every group of objects(or set) A can be associated with a quantity - denoted $|A|$ - called the number of elements in A or cardinality of A.

If $X = A \cup B$ and $A \cap B = \emptyset$, then $|X| = |A| + |B|$.

If A and B are not disjoint, we get the simplest form of the Inclusion-Exclusion Principle:
 $|A \cup B| = |A| + |B| - |A \cap B|$.

$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|$

In the more general case where there are n different sets A_i , the

EXAMPLES :**1. Prime Looking Numbers - HackerBlocks :**

How many numbers are there < 1000 such that they are Prime Looking

i.e. composite but not divisible by 2, 3 or 5 (Ex- 49, 77, 91). Given that there are 168 primes upto 1000.

For any positive number N and m , the number of integers divisible by m which are less than N is $\text{floor}((N-1)/m)$.

Solution :

divisible by 2 = $\text{floor}(999/2) = 499$; divisible by 3 = $\text{floor}(999/3) = 333$

divisible by 5 = $\text{floor}(999/5) = 199$; divisible by 2.3 = $\text{floor}(999/6) = 166$

divisible by 2.5 = $\text{floor}(999/10) = 99$

divisible by 3.5 = $\text{floor}(999/15) = 66$

divisible by 2.3.5 = $\text{floor}(999/30) = 33$

$$\begin{aligned}|2 \cup 3 \cup 5| &= |2| + |3| + |5| - |2 \cap 3| - |2 \cap 5| - |3 \cap 5| + |2 \cap 3 \cap 5| \\&= 499 + 333 + 199 - 166 - 99 - 66 + 33 = 733\end{aligned}$$

So, there exists 733 integers upto 1000 which have at-least 2, 3 or 5 as divisor. This includes {2, 3, 5}.

Total number not having 2, 3 or 5 as divisor = $999 - 733 = 266$.

Note that this set does not include 2, 3 or 5.

Since there are 168 prime numbers upto 1000, but we have already excluded 2, 3 and 5, number of prime looking numbers upto 1000 = $266 - 165 - 1$ (Since 1 is neither prime nor composite) = 100

The above generalised can be implemented using the following method :

Inclusion-Exclusion Using Bitmasks :

```
#include<iostream>
using namespace std;
int countBits(int n){
    int ans = 0 ;
    while(n){
        n = n&(n-1);
        ans++;
    }
    return ans;
}
int main(){
    //Given a array of numbers of size k
```

```

//we are finding the number
int arr = {2,3,5};
int k = 3;
int n = 999;
int ans = 0;
for(int i=1;i<8;i++){
    int mask = i;
    int bits = countBits(mask);
    int temp = 1;
    int pos = 0;
    while(mask>0){
        int lastBit = (mask&1);
        if(lastBit){
            temp = temp*a[pos];
        }
        mask = mask>>1;
        pos++;
    }
    cout<<endl;
    if(bits&1){
        ans += n/temp;
    }
    else{
        ans -= n/temp;
    }
}
cout<<ans<<endl;
return 0;
}

```

1. Sereja & LCM - Codechef (Hard, Long Contest 9th Question) :

We have to find the possible number of arrays: $A[1], A[2], A[3], \dots, A[N]$ such that $A[i] \geq 1$ and $A[i] \leq M$ and $\text{LCM}(A[1], A[2], \dots, A[N])$ is divisible by D. We have to find the sum of the answers with $D = L, L+1, \dots, R$ modulo $10^9 + 7$.

A/Q we have to find the number of array whose LCM is a multiple of a given number(say ' x ').

Using negation calculate the number of arrays whose LCM is not a multiple of x (say ' y ').

Hence, $\text{ans} = (\text{possible array with } m \text{ numbers}) - y$.

Note: The maximum value of the array elements can be 1000, the maximum number of distinct prime factors possible is 4 ($2 * 3 * 5 * 7 * 11 > 1000$).

Let the prime factors of x be p, q, r, s

$$x = (p^a) * (q^b) * (r^c) * (s^d)$$

$$p^a: P$$

$$q^b: Q$$

$$r^c: R$$

$$s^d: S$$

To calculate y :

None of element of array have any prime factor that x has OR it may have some of it missing.

So, calculate the number of arrays such that either (P or its multiple are not present) OR (Q or its multiple are not present) OR (R or its multiple are not present) OR (S or its multiple are not present).

$$y = |\text{not}(P) \cup \text{not}(Q) \cup \text{not}(R) \cup \text{not}(S)|$$

Applying Principle of Inclusion-Exclusion Principle :

$$A = \text{power}(m - m/P, n) + \text{power}(m - m/Q, n) + \text{power}(m - m/R, n) + \text{power}(m - m/S, n);$$

$$B = \text{power}(m - m/P - m/Q + m/(P*Q), n) + \text{power}(m - m/Q - m/R + m/(Q*R), n) \dots$$

$$C = \text{power}(m - m/P - m/Q - m/R + m/(P*Q) + m/(Q*R) + m/(P*R) - m/(P*Q*R), n) \dots$$

$$D = \text{powmod}(m - m/P - m/Q - m/R - m/S + m/(P*Q) + m/(Q*R) + m/(P*R) + m/(R*S) + m/(P*S) + m/(Q*S) - m/(P*Q*R) - m/(Q*R*S) - m/(P*Q*S) - m/(P*R*S) + m/(P*Q*R*S), n);$$

Final Answer will be:

$$y = A - B + C - D$$

$$\text{ans} = m^n - y$$

$$= m^n - A + B - C + D$$

Mathematical Expectation and Bernoulli Trial

Mathematically, for a discrete variable X with probability function $P(X)$, the expected value $E(X)$ is given by $\sum x_i P(x_i)$ the summation runs over all the distinct values x_i that the variable can take.

For example, for a dice-throw experiment, the set of discrete outcomes is $\{1, 2, 3, 4, 5, 6\}$ and each of this outcome has the same probability $1/6$. Hence, the expected value of this experiment will be $1/6 * (1+2+3+4+5+6) = 21/6 = 3.5$.

- Mathematical expectation is some sort of average value of our random variable.
- Expected value is not same as "most probable value" - rather, it need not even be one of the probable values. For example, in a dice-throw experiment, the expected value, viz 3.5 is not one of the possible outcomes at all.
- Rather the most probable value is the value with max probability.
- The rule of "linearity of the expectation" says that $E[aX_1 + bX_2] = aE[X_1] + bE[X_2]$.

Bernoulli Trial

In the theory of probability and statistics, a Bernoulli trial (or binomial trial) is a random experiment with exactly two possible outcomes, "success" and "failure", in which the probability of success is the same every time the experiment is conducted.

1. What is the expected number of coin flips for getting a head?

Let the expected number of coin flips be x . Then we can write an equation for it :

- (a) If the first flip is the head, then we are done. The probability of this event is $1/2$ and the number of coin flips for this event is 1 .
- (b) If the first flip is the tails, then we have wasted one flip. Since consecutive flips are independent events, the solution in this case can be recursively framed in terms of x - The probability of this event is $1/2$ and the expected number of coins flips now onwards is x . But we have already wasted one flip, so the total number of flips is $x + 1$.

The expected value x is the sum of the expected values of these two cases. Using the rule of linearity of the expectation and the definition of Expected value, we get

$$x = (1/2)(1) + (1/2)(1+x)$$

Solving, we get $x = 2$.

Thus the expected number of coin flips for getting a head is 2.

2. What is the expected number of coin flips for getting two consecutive heads?

Let the expected number of coin flips be x . The case analysis goes as follows:

- (a) If the first flip is a tails, then we have wasted one flip. The probability of this event is $1/2$ and the total number of flips required is $x + 1$.
- (b) If the first flip is a heads and second flip is a tails, then we have wasted two flips. The probability of this event is $1/4$ and the total number of flips required is $x + 2$.
- (c) If the first flip is a heads and second flip is also heads, then we are done. The probability of this event is $1/4$ and the total number of flips required is 2.

Adding, the equation that we get is $x = (1/2)(x + 1) + (1/4)(x + 2) + (1/4)2$

Solving, we get $x = 6$.

Thus, the expected number of coin flips for getting two consecutive heads is 6.

3. What is the expected number of coin flips for getting N consecutive heads, given N?

Let the expected number of coin flips be x . Based on previous exercises, we can wind up the whole case analysis in two basic parts

(a) If we get 1st, 2nd, 3rd,...,n'th tail as the first tail in the experiment, then we have to start all over again.

(b) Else we are done.

For the 1st flip as tail, the part of the equation is $(1/2)(x+1)$

For the 2nd flip as tail, the part of the equation is $(1/4)(x+2)$

...
For the k'th flip as tail, the part of the equation is $(1/(2^k))(x+k)$

...
For the N'th flip as tail, the part of the equation is $(1/(2^N))(x+N)$

The part of equation corresponding to case (b) is $(1/(2^N))(N)$

Adding, $x = (1/2)(x+1) + (1/4)(x+2) + \dots + (1/(2^k))(x+k) + \dots + (1/(2^N))(x+N) + (1/(2^N))(N)$

Solving this equation is left as an exercise to the reader. The entire equation can be very easily reduced to the following form: $x = 2^{N+1} - 2$

Thus, the expected number of coin flips for getting N consecutive heads is $(2^{N+1} - 2)$.

4. Candidates are appearing for interview one after other. Probability of each candidate getting selected is 0.16. What is the expected number of candidates that you will need to interview to make sure that you select somebody?

This is very similar to Q1, the only difference is that in this case the coin is biased. (The probability of heads is 0.16 and we are asked to find number of coin flips for getting a heads).

Let x be the expected number of candidates to be interviewed for a selection. The probability of first candidate getting selected is 0.16 and the total number of interviews done in this case is 1. The other case is that the first candidate gets rejected and we start all over again. The probability for that is $(1 - 0.16)*(x + 1)$. The equation thus becomes : $x = 0.16 + (1-0.16)*(x+1)$.

Solving, $x = 1/0.16$, i.e. $x = 6.25$

5. (Generalized version of Q4) - The queen of a honey bee nest produces off-springs one-after-other till she produces a male offspring. The probability of producing a male offspring is p. What is the expected number of off-springs required to be produced to produce a male offspring?

This is same as the previous question, except that the probability of success is p . Now the equation now becomes : $x = p + (1 - p)(x + 1)$

Solving, $x = 1/p$

Thus, observe that in the problems where there are two events, where one event is desirable and other is undesirable, and the probability of desirable event is p , then the expected number of trials done to get the desirable event is $1/p$.

Generalizing on the number of events - If there are K events, where one event is desirable and all others are undesirable, and the probability of desirable event is p , then also the expected number of trials done to get the desirable event is $1/p$.

The next question uses this generalization.

6. What is the expected number of dice throws required to get a "four"?

Let the expected number of throws be x . The desirable event (getting 'four') has probability $1/6$ (as each face is equiprobable). There are 5 other undesirable events ($K=5$). Note that the value of x does not depend on K . The answer is thus $1/(1/6) = 6$.

7. Candidates are appearing for interview one after other. Probability of k -th candidate getting selected is $1/(k+1)$. What is the expected number of candidates that you will need to interview to make sure that you select somebody?

The result will be the sum of infinite number of cases.

Case 1: First candidate gets selected. The probability of this event is $1/2$ and the number of interviews is 1.

Case 2: Second candidate gets selected. The probability of this event is $1/6$ ($= 1/2 \times 1/3$ of first candidate not getting selected and $1/3$ of second candidate getting selected, multiplied together gives $1/6$) and the number of interviews is 2.

Case 3: Third candidate gets selected. The probability of this event is $1/2 \times 2/3 \times 1/4 = 1/12$ ($=$ first not getting selected and second not getting selected and third getting selected) and the number of interviews is 3.

...

Case k : k 'th candidate gets selected. The probability of this event is $1/2 \times 2/3 \times 3/4 \times \dots \times (k-1)/k \times 1/(k+1)$. (The first $k-1$ candidates get rejected and the k 'th candidate is selected). This evaluates to $1/(k(k+1))$ and the number of interviews is k .

...

[Note that similar to problem 4, here we can't just say - if the first candidate is rejected, then we will start the whole process again. This is not correct, because the probability of each candidate depends on its sequence number. Hence sub-experiments are not same as the parent experiment. This means that all the cases must be explicitly considered.]

The resultant expression will be :

$$\begin{aligned}x &= 1/(1*2) + 2/(2*3) + 3/(3*4) + 4/(4*5) + \dots + k/(k*(k+1)) + \dots \\&= 1/2 + 1/3 + 1/4 + \dots\end{aligned}$$

This is a well-known divergent series, which means that sum does not converge, and hence the expectation does not exist.

8. A random permutation P of $[1\dots n]$ needs to be sorted in ascending order. To do this, at every step you will randomly choose a pair (i,j) where $i < j$ but $P[i] > P[j]$, and swap $P[i]$ with $P[j]$. What is the expected number of swaps needed to sort permutation in ascending order. (Idea: Topcoder)

This is a programming question, and the idea is simple - since each swap has same probability of getting selected, the total number of expected swaps for a permutation P ----- is

$$E[P] = (1/cnt) * \sum (E[P_s] + 1)$$

where cnt is the total number of swaps possible in permutation P , and P_s ----- is the permutation generated by doing swap 's'. Since all swaps are equiprobable, we simply sum up the expected values of the resultant permutations (of course add 1 to each to account for the swap done already) and divide the result by the total number of permutations. The base case will be for the array that has been already sorted - and the expected number of permutations for a sorted array is 0.

9. A fair coin flip experiment is carried out N times. What is the expected number of heads?

Consider an experiment of flipping a fair coin N times and let the outcomes be represented by the array $Z = \{a_1, a_2, \dots, a_n\}$ where each a_i is either 1 or 0 depending on whether the outcome was heads or tails respectively. In other words, for each i we have $a_i = 1$ if the i^{th} experiment gave head then 1 else 0.

Hence we have: Number of heads in $Z = a_1 + a_2 + \dots + a_n$

$$\text{Hence } E[\text{number of heads in } Z] = E[a_1 + a_2 + \dots + a_n] = E[a_1] + E[a_2] + \dots + E[a_n]$$

Since a_i corresponds to a coin-toss experiment, the value of $E[a_i]$ is 0.5 for each i . Adding this n times, the expected number of heads in Z comes out to be $n/2$.

10. (Bernoulli Trials) n students are asked to choose a number from 1 to 100 inclusive. What is the expected number of students that would choose a single digit number?

This question is based on the concept of Bernoulli trials. An experiment is called a Bernoulli trial if it has exactly two outcomes, one of which is desired. For example - flipping a coin, selecting a number from 1 to 100 to get a prime, rolling a dice to get 4 etc. The result of a Bernoulli trial can typically be represented as "yes/no" or "success/failure". We have seen in Q5 above that if the probability of success of a Bernoulli trial is p then the expected number of trials to get a success is $1/p$. is

This question is based on yet another result related to bernoulli trials - If the probability of a success in a bernoulli trial is p then the expected number of successes in n trials is $n \cdot p$. The proof is simple :

The number of successes in n trials = (if 1st trial is success then 1 else 0) + ... + (if n th trial is success then 1 else 0)

The expected value of each bracket is $1 \cdot p + 0 \cdot (1-p) = p$. Thus the expected number of successes in n trials is $n \cdot p$.

In the current case, "success" is defined as the experiment that chooses a single digit number. Since all choices are equiprobable, the probability of success is $9/100$. (There are 9 single digit numbers in 1 to 100). Since there are n students, the expected number of students that would contribute to success (i.e the expected number of successes) is $n \cdot 9/100$.

11. What is the expected number of coin flips to ensure that there are atleast N heads?

The solution can easily be framed in a recursive manner :

N heads = if 1st flip is a head then $N-1$ more heads, else N more heads.

The probability of 1st head is $1/2$. Thus $E[N] = (1/2)(E[N-1]+1) + (1/2)(E[N] + 1)$

Note that each term has 1 added to it to account for the first flip.

The base case is when $N = 1$: $E[1] = 2$ (As discussed in Q2)

Simplifying the recursive case, $E[N] = (1/2)(E[N-1] + 1 + E[N] + 1) = (1/2)(E[N-1] + E[N] + 2)$

$$\Rightarrow 2 \cdot E[N] = (E[N-1] + E[N] + 2) \Rightarrow E[N] = E[N-1] + 2$$

Since $E[1] = 2$, $E[2] = 4$, $E[3] = 6, \dots$, in general $E[N] = 2N$. Thus, the expected number of coin flips to ensure that there are atleast N heads in $2N$.

The next problem discusses a generalization :

12. What is the expected number of bernoulli trials to ensure that there are at least N successes, if the probability of each success is p ?

The recursive equation in this case is $E[N] = p(E[N-1] + 1) + (1-p)(E[N] + 1)$

Solving, $E[N] - E[N-1] = p$. Writing a total of $N-1$ equations:

$$E[N] - E[N-1] = 1/p$$

$$E[N-1] - E[N-2] = 1/p$$

$$E[N-2] - E[N-3] = 1/p$$

...

$$E[2] - E[1] = 1/p$$

Adding them all, $E[N] - E[1] = (n-1)/p$. But $E[1]$ is $1/p$ (lemma -1). Hence $E[N] = n/p$.

Moral: If probability of success in a Bernoulli trial is p , then the expected number of trials to guarantee N successes is N/p .

This completes the discussion on problems on Mathematical Expectation.

Reference : [Codechef](#)



TRY IT YOURSELF !

1. A game involves you choosing one number (between 1 to 6 inclusive) and then throwing three fair dice simultaneously. If none of the dice shows up the number that you have chosen, you lose \$1. If exactly one, two or three dice show up the number that you have chosen, you win \$1, \$3 or \$5 respectively. What is your expected gain?
2. There are 10 flowers in a garden, exactly one of which is poisonous. A dog starts eating all these flowers one by one at random. whenever he eats the poisonous flower he will die. What is the expected number of flowers he will eat before he will die?
3. A bag contains 64 balls of eight different colours, with eight of each colour. What is the expected number of balls you would have to pick (without looking) to select three balls of the same colour?
4. In a game of fair dice throw, what is the expected number of throws to make sure that all 6 outcomes appear atleast once?
5. What is the expected number of bernoulli trials for getting N consecutive successes, given N , if the probability of each success is p ?

Coupon Collector Problem

Problem Statement: A certain brand of cereal always distributes a coupon in every cereal box. The coupon chosen for each box is chosen randomly from a set of ' n ' distinct coupons. A coupon collector wishes to collect all ' n ' distinct coupons. What is the expected number of cereal boxes must the coupon collector buy so that the coupon collector collects all ' n ' distinct coupons?

Solution:

Let random variable X_i be the number of boxes it takes for the coupon collector to collect the i -th new coupon after the $i-1$ th coupon has already been collected. (Note: this does NOT mean assign numbers to coupons and then collect the i -th coupon. Instead, this means that after X_i boxes, the coupon collector would have collected i distinct coupons, but with only X_{i-1} boxes, the coupon collector would have only collected $i-1$ distinct coupons.)

Clearly $E(X_1)=1$, because the coupon collector starts off with no coupons. Now consider the i -th coupon. After the $i-1$ -th coupon has been collected, then there are $n-(i-1)$ possible coupons that could be the new i -th coupon. Each trial of buying another cereal box, "success" is getting any of the $n - (i - 1)$ uncollected coupons, and "failure" is getting any of the already collected $i-1$ coupons. From this point of view, we see that $p = (n-(i-1)) / n$.

Mathematics for Competitive Coding

This is a bernoulli trial ... probability of success p and failure $(1-p)$. In bernoulli trial, the expected number of trials for i -th success is $1/p$ i.e. $1/(success \text{ of the } i\text{-th outcome})$.

$$E(X_i) = 1/p = n/n - (i-1).$$

To compute the number of cereal boxes X , required by the coupon collector to collect all n distinct coupons:

$$E(X) = E(X_1 + X_2 + X_3 + X_4 + \dots + X_n)$$

$$E(X) = E(X_1) + E(X_2) + \dots + E(X_n)$$

$$E(X) = n(1 + 1/2 + 1/3 + 1/4 + \dots + 1/n)$$

EXAMPLE

Favorite Dice (Spoj)

What is the expected number of throws of N sided dice so that each number is rolled at least once?

Statement - <http://www.spoj.com/problems/FAVDICE>

Code :

```
#include <iostream>
#include<iomanip>
using namespace std;
int main() {
    int t;
    int n;
    cin>>t;
    while(t--){
        cin>>n;
        double ans = 0;
        for(int i=1;i<=n;i++){
            ans += n/(i*1.0);
        }
        cout<<fixed<<setprecision(6)<<ans<<endl;
    }
    return 0;
}
```

**TIME TO TRY****Fibonacci Sum (Spoj)**

Given two non-negative integers N and M, you have to calculate the sum $(F(N) + F(N + 1) + \dots + F(M)) \bmod 1000000007$ where $F(N)$ denotes the nth Fibonacci Number.

<http://www.spoj.com/problems/FIBOSUM/> (Matrix Exponentiation)

Modulo Sum (Codeforces) :

You are given a sequence of numbers a_1, a_2, \dots, a_n , and a number m .

Check if it is possible to choose a non-empty subsequence a_j such that the sum of numbers in this subsequence is divisible by m .

<http://codeforces.com/contest/577/problem/B>

Tavas and SaDDas (Codeforces) :

You are given a lucky number n . Lucky numbers are the positive integers whose decimal representations contain only the lucky digits 4 and 7. For example, numbers 47, 744, 4 are lucky and 5, 17, 467 are not. If we sort all lucky numbers in increasing order, what's the 1-based index of n ?

<http://codeforces.com/problemset/problem/535/B> (Maths, Counting)

Count the Binary Trees (HackerBlocks) :

Given n , you have to find the number of possible binary trees that can be made using N nodes.

Summing Sums (Spoj) :

Refer Spoj for the problem statement.

<http://www.spoj.com/problems/SUMSUMS/> (Mathematics)

Marbles (Spoj) :

Refer Spoj for the problem statement.

<http://www.spoj.com/problems/MARBLES/> (Maths)

SELF STUDY NOTES

3

Number Theory

In this chapter we are going to talk about important mathematical concepts, theorems and tricks. Let's get started.

Greatest Common Divisor(GCD) using Euclid's Algorithm

The Greatest Common Divisor (GCD) of two integers (a, b) denoted by $\text{gcd}(a,b)$, is defined as the largest positive integer d such that $d \mid a$ and $d \mid b$ where $x \mid y$ implies that x divides y .

Example of GCD: $\text{gcd}(4, 8) = 4$, $\text{gcd}(10, 5) = 5$, $\text{gcd}(20, 12) = 4$.

$\text{Gcd}(A,B) = \text{Gcd}(B,A \% B)$ // recurrence for gcd

$\text{Gcd}(A,0) = A$ // base case

Proof : If $a = bq + r$,

Let d be any common divisor of a and b which implies $d \mid a$ and $d \mid b \Rightarrow d \mid (a - bq) \Rightarrow d \mid r$.

Let e be any common divisor of b and $r \Rightarrow e \mid b$, $e \mid r \Rightarrow e \mid (bq + r) \Rightarrow e \mid a$. hence any common divisor of a and b must also be a common divisor of r and any common divisor of b and r must also be a divisor of $a \Rightarrow d$ is a common divisor of a and b iff d is a common divisor of b and r .

Similarly, the LCM of two integers (a, b) denoted by $\text{lcm}(a,b)$, is defined as the smallest positive integer l such that $a \mid l$ and $b \mid l$. Example of LCM : $\text{lcm}(4,8) = 8$, $\text{lcm}(10,5) = 10$, $\text{lcm}(20,12) = 60$.

$$\text{gcd}(a,b) * \text{lcm}(a,b) = a * b$$

```
int gcd(int a, int b) {
    return (b == 0 ? a : gcd(b, a % b));
}
int lcm(int a, int b) {
    return (a * (b / gcd(a, b)));
} // divide before multiply!
```

The GCD of more than 2 numbers, e.g. $\text{gcd}(a,b,c)$ is equal to $\text{gcd}(a,\text{gcd}(b,c))$, etc, and similarly for LCM. Both GCD and LCM algorithms run in $O(\log(N))$, where $n = \max(a,b)$.

Extended Euclid's Algorithm

Extended Euclid's is used to find out the solution of equations of the form $Ax + By = C$, where C is a multiple of divisor of A and B . Extended Euclid's works in the same manner as the euclid's algorithm. If $By = 1$ (we will find solutions of this equation let them be x' and y' given that $\text{gcd}(a,b) = 1$ then the solutions of equation $Ax + By = k$ where k is a multiple of $\text{gcd}(A,B)$ are given by $k*x'$ and $k*y'$.

$$Ax + By = 1 \quad \dots \dots (1)$$

$$Bx' + (A \% B)y' = 1 \quad \dots \dots (2) // \text{ using euclid's algo } \text{gcd}(a,b) = \text{gcd}(b,a \% b)$$

Compare coefficients of (1) and (2)

$$x = y'$$

$$y = x' - [A/B]y'$$

Hence we can recursively calculate x and y in the following manner:

```
void eeuclid(ll a, ll b){
    if(b==0){
        ex=1;ey=0;ed=a;
    }
    else{
        eeuclid(b,a%b);
        ll temp=ex;
        ex=ey;
        ey=temp-(a/b)*ey;
    }
}
```

Application of Extended Euclidean Algorithm :

- To calculate multiplicative modulo inverse of a w.r.t. m.

Let's see :

$$x \equiv a^{-1} \pmod{m}$$

$$a \cdot a^{-1} \equiv a \cdot x \pmod{m}$$

$$x \cdot a \equiv 1 \pmod{m}$$

$$\Rightarrow ax - 1 \equiv 0 \pmod{m}$$

$$\Rightarrow ax - qm = 1$$

This equation has solutions only if a and m are co-prime that is $\text{gcd}(a,m) = 1$.

We can calculate x and q using extended euclid's algorithm where x is the inverse of a modulo m .

Sieve of Eratosthenes

It is easy to find if some number (say N) is prime or not — you simply need to check if at least one number from numbers lower or equal \sqrt{n} is divisor of N. This can be achieved by simple code:

```
boolean isPrime( int n )
{
    if ( n == 1 )
        return false; // by definition, 1 is not prime number
    if ( n == 2 )
        return true; // the only one even prime
    for ( int i = 2; i * i <= n; ++i )
        if ( n % i == 0 )
            return false;
    return true;
}
```

So it takes \sqrt{n} steps to check this. Of course you do not need to check all even numbers, so it can be "optimized" a bit:

```
boolean isPrime( int n )
{
    if ( n == 1 )
        return false; // by definition, 1 is not prime number
    if ( n == 2 )
        return true; // the only one even prime
    if ( n % 2 == 0 )
        return false; // check if is even
    for ( int i = 3; i * i <= n; i += 2 ) // for each odd number
        if ( n % i == 0 )
            return false;
    return true;
}
```

So let say that it takes $0.5\sqrt{n}$ steps*. That means it takes 50,000 steps to check that 10,000,000,000 is a prime.

Time Complexity :

If we have to check numbers upto N, we have to check each number individually. So time complexity will be $O(N\sqrt{N})$.

Can we do better?

Ofcourse! we can use a sieve of numbers upto N. For all prime numbers $\leq \sqrt{N}$, we can make their multiple non-prime i.e. if p is prime, $2p, 3p, \dots, \lfloor n/p \rfloor * p$ will be non-prime.

Sieve code :

```
void primes(int *p)
{
    for(int i = 2;i<=1000000;i++)
        p[i] = 1;
    for(int i = 2;i<=1000000;i++)
    {
        if(p[i])
        {
            for(int j = 2*i;j<=1000000;j+=i)
            {
                p[j] = 0;
            }
        }
    }
    p[1] = 0;
    p[0] = 0;
    return;
}
```

Can we still do better?

Yeah sure! Here we don't need to check for even numbers. Instead of starting the non-prime loop from $2p$ we can start from p^2 .

Optimised code :

```
void primes(bool *p)
{
    for(int i = 3;i<=1000000;i += 2)
    {
        if(p[i])
        {
            for(int j = i*i;j <= 1000000; j += i)
            {
                p[j] = 0;
```

```

    }
}

p[1] = 0;
p[0] = 0;
return;
}
t = O(NloglogN)

```

Hence, we have significantly reduced our complexity from $N\sqrt{N}$ to approx linear time.

Optimizations!

We know that all the numbers which are even are non prime except 2. Hence we can mark only odd numbers as non prime in our sieve and jump to odd numbers always.

Code :

```

#define lim 100000000
vector<bool> mark(lim+1,1);
vector<ll> primes;
void sieve() // we need primes upto 10^8
{
    //ll times = 0;
    for(ll i=3;i<lim;i+=2)
    {
        //times++;
        if(mark[i] == 1)
        {
            for(ll j=i*i ; j < lim ; j += 2*i) //skip to odd numbers as i*i is odd
            {
                mark[j] = 0;
            }
        }
    }

    primes.pb(2);
    for(ll i=3;i<lim;i+=2)
    {
        if(mark[i])
            primes.pb(i);
    }
}

```

Factorization of a Number using this Sieve :

```
vector<ll> factorize(ll m)
{
    vector<ll> factors;
    factors.clear();
    ll i = 0;
    ll p = primes[i];
    while(p*p <= m)
    {
        if(m%p == 0)
        {
            factors.pb(p);
            while(m%p == 0)
                m = m/p;
        }
        i++;
        p = primes[i];
    }
    if(m!=1)
        factors.pb(m);
    return factors;
}
```

Segmented Sieve

We use this sieve when array of size N does not fit in memory and we want to compute prime numbers between a range l and r . Example : $l = 10^8$, $r = 10^9$.

```
void sieve()
{
    for(int i = 0;i<=10000000;i++)
        p[i] = 1;
    for(int i = 2;i<=10000000;i++)
    {
        if(p[i])
        {
            for(int j = 2*i;j<=10000000;j+=i)
                p[j] = 0;
```

Number Theory

```
}

}

// for(int i=2;i<=20;i++) cout<<i<<" "<<p[i]<<endl;

}

int segmented_sieve(long long a, long long b)
{
    sieve();
    bool pp[b-a+1];
    memset(pp, 1, sizeof(pp));
    for(long long i = 2; i*i<=b; i++)
    {
        for(long long j = a; j<=b; j++)
        {
            if(p[i])
            {
                if(j == i)
                    continue;
                if(j % i == 0)
                    pp[j-a] = 0;
            }
        }
    }
    int res = 1;
    for(long long i = a; i<b; i++)
        res += pp[i-a];
    return res;
}
```

Division

Let a and b be integers. We say a divides b , denoted by $a|b$, if there exists an integer c such that $b = ac$.

Linear Diophantine Equations

A Diophantine equation is a polynomial equation, usually in two or more unknowns, such that only the integral solutions are required. An Integral solution is a solution such that all the unknown variables take only integer values. Given three integers a , b , c representing a linear equation of the form : $ax + by = c$. Determine if the equation has a solution such that x and y are both integral values.

General solution (Infinitely many solutions)

$$(x, y) = (x_0 + b/d * t, y_0 - a/d * t)$$

We can use *Extended Euclidean Method* above to find the x_0, y_0 .

Chinese Remainder Theorem

Typical problems of the form "Find a number which when divided by 2 leaves remainder 1, when divided by 3 leaves remainder 2, when divided by 7 leaves remainder 5" etc can be reformulated into a system of linear congruences and then can be solved using Chinese Remainder theorem.

For example, the above problem can be expressed as a system of three linear congruences:

$$x \equiv 1 \pmod{2}, \quad x \equiv 2 \pmod{3}, \quad x \equiv 5 \pmod{7}.$$

$$x \% \text{num}[0] = \text{rem}[0],$$

$$x \% \text{num}[1] = \text{rem}[1],$$

.....

$$x \% \text{num}[k-1] = \text{rem}[k-1]$$

A Naive Approach is to find x is to start with 1 and one by one increment it and check if dividing it with given elements in `num[]` produces corresponding remainders in `rem[]`. Once we find such a x , we return it

Chinese remainder theorem

$$x = \sum_{0 \leq i \leq n-1} (\text{rem}[i] * \text{pp}[i] * \text{inv}[i]) \% \text{prod}$$

`rem[i]` is given array of remainders

`prod` is product of all given numbers

$$\text{prod} = \text{num}[0] * \text{num}[1] * \dots * \text{num}[k-1]$$

`pp[i]` is product of all but `num[i]`

$$\text{pp}[i] = \text{prod} / \text{num}[i]$$

`inv[i]` = Modular Multiplicative Inverse of

`pp[i]` with respect to `num[i]`

Code :

```
ll chinese_remainder_theorem(vector<ll> num, vector<ll> rem)
{
    // find pp vector
    vector<ll> pp; // product of all num array except num[i]
    pp.clear();
```

```

    ll prod = 1ll;
    for(ll i=0;i<num.size();++i)
        prod *= num[i];
    for(ll i=0;i<num.size();++i)
        pp.pb(prod/num[i]);
    // find inv[] vector
    // inv[i] is modular inverse of pp[i] with respect to num[i]
    vector<ll> inv;
    inv.clear();
    for(ll i=0;i<pp.size();++i)
        inv.pb(modular_inverse(pp[i],num[i]-2,num[i]));
    // (a^-1)%m when m is prime is (a^(m-2))%m using fermat's
    // now use the sum formula
    ll ans = 0ll;
    for(ll i=0;i<pp.size();++i)
    {
        ans = ans%prod + ( ((rem[i]*pp[i])%prod)*(inv[i])%prod )%prod;
        ans %= prod;
    }
    return ans;
}

```

Euler Phi Function

Euler's Phi function (also known as totient function, denoted by ϕ) is a function on natural numbers that gives the count of positive integers coprime with the corresponding natural number.

Thus, $\phi(8) = 4$, $\phi(9) = 6$

The value $\phi(n)$ can be obtained by Euler's formula :

Let $n=p_1^{a_1} * p_2^{a_2} * \dots * p_k^{a_k}$ be the prime factorization of n . Then

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

Code :

```

int phi[] = new int[n+1];
for(int i=2; i <= n; i++)
    phi[i] = i; //phi[1] is 0

```

```

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

```

Properties :

1. If P is prime then $\varphi(p^k) = (p - 1)p^{k-1}$.
2. φ function is multiplicative, i.e. if $(a,b) = 1$ then $\varphi(ab) = \varphi(a)\varphi(b)$.
3. Let d_1, d_2, \dots, d_k be all divisors of n (including n). Then $\varphi(d_1) + \varphi(d_2) + \dots + \varphi(d_k) = n$

For Example: The divisors of 18 are 1,2,3,6,9 and 18.

Observe that $\varphi(1) + \varphi(2) + \varphi(3) + \varphi(6) + \varphi(9) + \varphi(18) = 1 + 1 + 2 + 2 + 6 + 6 = 18$

4. Number of divisors of $n = p_1^{a_1} * p_2^{a_2} * \dots * p_n^{a_n}$:

$$d(n) = (a_1 + 1) * (a_2 + 1) * \dots * (a_n + 1)$$

5. Sum of divisors:

$$S(n) = \frac{p_1^{a_1+1}-1}{p_1-1} * \frac{p_2^{a_2+1}-1}{p_2-1} * \dots * \frac{p_n^{a_n+1}-1}{p_n-1}$$

Wilson's Theorem

If p is a prime, then $(p - 1)! \equiv -1 \pmod{p}$

Problem : DCEPC11B (SPOJ)**Hint :**

This can be solved using Wilson theorem

1. If $n >= p$ ans would be 0
2. Else we have to use Wilson's theorem

$$(p - 1)! \equiv -1 \pmod{p}$$

$$1 * 2 * 3 * \dots * (n - 1) * (n) * \dots * (p - 1) \equiv -1 \pmod{p}$$

$$n! \equiv (n + 1) * \dots * (p - 1) \equiv -1 \pmod{p}$$

$$n! \equiv -1 * [(n + 1) * \dots * (p - 2) * (p - 1)] \pmod{p}$$

Lucas Theorem

In number theory, Lucas's theorem expresses the remainder of division of the binomial coefficient $\binom{m}{n}$ by a prime number p in terms of base p expansions of integers m and n .

Formulation :

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod{p},$$

where $m = m_k p^k + m_{k-1} p^{k-1} + \dots + m_1 p + m_0$ and $n = n_k p^k + n_{k-1} p^{k-1} + \dots + n_1 p + n_0$

Problem : Compute ${}^nC_r \% p$.

Given three numbers n , r and p , compute the above value of ${}^nC_r \% p$.

Using Lucas Theorem ${}^nC_r \% p$

Lucas theorem basically suggests that the value of nC_r can be computed by multiplying results of ${}^nC_{r_i}$ where n_i and r_i are individually same-positioned digits in base p representations of n and r respectively. The idea is to one by one compute ${}^nC_{r_i}$ for individual digits n_i and r_i in base p .

Code :

```
#include<bits/stdc++.h>
using namespace std;
int Cal_nCr_mod_p(int n, int r, int p)
{
    int C[r+1];
    memset(C, 0, sizeof(C));
    C[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = min(i, r); j > 0; j--)
            C[j] = (C[j] + C[j-1])%p;
    }
    return C[r];
}
```

```

int LucasApproach(int n, int r, int p)
{
    if(r==0)
        return 1;
    else
    {
        int n_i = n%p, r_i = r%p;
        int result = (LucasApproach(n/p, r/p, p)*Cal_nCr_mod_p(n_i,r_i,p))%p;
        return result;
    }
}

int main()
{
    int n,r,p;
    cin>>n>>r>>p;
    int result = LucasApproach(n,r,p);
    cout<<"nCr mod p is "<<result;
}

```

Fermat's little Theorem

Fermat's little theorem states that if p is a prime number, then for any integer a , the number $a^p - a$ is an integer multiple of p . In the notation of modular arithmetic, this is expressed as

$$a^p \equiv a \pmod{p}.$$

For example, if $a = 2$ and $p = 7$, $2^7 = 128$, and $128 \equiv 2 \pmod{7}$ is an integer multiple of 7.

If a is not divisible by p , Fermat's little theorem is equivalent to the statement that $a^{p-1} - 1$ is an integer multiple of p , or in symbols

$$a^{p-1} \equiv 1 \pmod{p}$$

For example, if $a = 2$ and $p = 7$ then $2^6 = 64$ and $64 \equiv 1 \pmod{7}$ is thus a multiple of 7.

Problem based on Fermat's Theorem (Light's New Car) :

Statement : Given A and B, we have to find $(A \text{ power } B) \% (10^9 + 7)$ where $A, B \leq 10^{100000}$

First let's deal with the base A. Now what we have to find is $A\%(10^9 + 7)$. This is because, let's suppose $B = n$ (where n is an integer). A can be expressed as

$A = [a * (\text{mod}) + b]$ (where $\text{mod} = 10^9 + 7$, a and b are integers). This implies that
 $(A \uparrow n) \% \text{mod} = [(a * (\text{mod}) + b) \uparrow n] \% \text{mod} = [(a * (\text{mod}) + b) \uparrow \% \text{mod}] *$
 $\dots \dots \dots n \text{ times } \% \text{mod} = b \uparrow n$ (where b is nothing but $A \% \text{mod}$)

Using modulo properties :

$$(a * b) \% m = ((a \% m) * (b \% m)) \% m = (a \% m + b \% m) \% m$$

Now $A \% (10^9 + 7)$ can be found by iterating over the string A and generating an integer from it but at the same time taking its modulo with $(10^9 + 7)$ to prevent overflow.

Now let's deal with the power B. We can use the concept of fermat's little theorem as

$$x^{p-1} \% p = 1 \text{ (where } p \text{ is a prime number)}$$

B can be presented as $B = a * (p - 1) + b$ (where a and b are integers and $p = (10^9 + 7)$)

$$\text{Hence } A^B \% p = A^{(a * (p-1) + b)} \% p = [(A^{a * (p-1)}) \% p * [A^b] \% p] \% p = (A^b) \% p$$

Here b is nothing but $B \% (p - 1)$

Using Fermat's little theorem and modulo properties

Now $B \% (p - 1)$ can be found in the similar way as $A \% (10^9 + 7)$

Finally

$$\text{Let } x = A \% (10^9 + 7), n = B \% (10^9 + 7 - 1)$$

Required answer would be $x \% (10^9 + 7)$ which can be found out easily by fast modulo exponentiation in $O(\log n)$.

Miller - Rabin Primality Test

The Miller-Rabin primality test or Rabin-Miller primality test is a primality test: an algorithm which determines whether a given number is prime.

This method is a probabilistic method to find Prime number.

Input #1 : $n > 3$, an odd integer to be tested for primality;

Input #2 : k , a parameter that determines the accuracy of the test

Output: Composite if n is composite, otherwise probably prime

write $n - 1$ as $2^r \cdot d$ with d odd by factoring powers of 2 from $n - 1$

WitnessLoop : repeat k times : pick a random integer a in the range $[2, n - 2]$

$x \leftarrow a^d \bmod n$

if $x = 1$ or $x = n - 1$ then

continue WitnessLoop

repeat $r - 1$ times:

```

x ←  $x^2 \bmod n$ 
if x = 1 then
    return composite
if x = n - 1 then
    continue WitnessLoop
return composite
return probably prime

```

Example :

Input: n = 13, k = 2

1. Computed d and r such that $d \cdot 2r = n - 1$,
d = 3, r = 2.
2. Call millerTest k times.

1st Iteration:

1. Pick a random number 'a' in range [2, n - 2]
Suppose a = 4
2. Compute: $x = \text{pow}(a, d) \% n$
 $x = 4^3 \% 13 = 12$
3. Since $x = (n - 1)$, return prime.

2nd Iteration:

1. Pick a random number 'a' in range [2 + n - 2]
Suppose a = 5
2. Compute: $x = \text{pow}(a, d) \% n$
 $x = 5^3 \% 13 = 8$
3. x neither 1 nor 12
4. Do following $(r - 1) = 1$ times
(a) $x = (x * x) \% 13 = (8 * 8) \% 13 = 12$ (b) Since $x = (n - 1)$, return true.

Since both iterations return true, we return prime.



POWPOW2, SPOJ

Problem :

Given three integers a, b, n, $1 \leq a, b, n \leq 10^{15}$

$a^{(b(f(n)))} \bmod 1000000007$, where $f(n) = (^nC_0^2 + ^nC_1^2 + \dots + ^nC_n^2)$

Dealing with $f(n)$:

The function f complicates the problem. Notice that $f(n) = {}^{2n}C_n$. It's easy to find proofs online, e.g. here, so I'll skip that.

Reducing the Exponents:

$b^{(2n, n)}$ is a huge number and we need to reduce it to a more tractable number. Euler's theorem states that if a and m are coprime, then $a^{\varphi(m)} \equiv 1 \pmod{m}$, where $\varphi(m)$ is Euler's totient function. This is useful because $a^y \equiv a^{y \pmod{\varphi(m)}} \pmod{m}$.

The repeated $\varphi(m)$ factors in the exponent will yield a bunch of 1s.
 $m = 10^9 + 7$ which is a prime number, so $\varphi(m) = m - 1 = 10^9 + 6 = 2 \times 500000003$

So, we have $a^{y \pmod{100000006}} \pmod{1000000007}$. The main difficulty of this problem is that our y is also an exponential, $y = b^{(2n, n)}$. In order to find the result, we need first to calculate $b^{(2n, n)} \pmod{1000000006}$.

Finding $b^{(2n, n)} \pmod{100000006}$ when b is odd

Suppose b is odd. Then, we can apply Euler's theorem because b and 1,000,000,006 are coprime (recall that $b \leq 10^5$ so the 500000003 factor will always be coprime with b).

$$\begin{aligned} b^{(2n, n)} &= b^{(2n, n) \pmod{\varphi(100000006)}} \pmod{1000000006} \\ \varphi(100000006) &= \varphi(2) \times (500000003) = (2-1) \times (500000003 - 1) = 500000002 \\ 500000002 &= 2 \times 41^2 \times 148721500000002 = 2 \times 41^2 \times 148721 \end{aligned}$$

So, we need to find $(2n, n) \pmod{500000002}$ which is not prime. Therefore, we need to use another tool: the Chinese Remainder Theorem (CRT). We can calculate

$$(2n, n) \pmod{2}$$

$$(2n, n) \pmod{41^2}$$

$$(2n, n) \pmod{148721}$$

and use CRT to get the result modulo 500000002.

Finding $b^{(2n, n)} \pmod{1000000006} \pmod{b^{(2n, n)}}$ when b is even

Unfortunately, if b is even, b and 1000000006 are not coprime.

Therefore, we need CRT again. Our modulus is the product of two primes: 2 and 500,000,003. So, we shall find $(2n, n) \pmod{2}$ and $(2n, n) \pmod{500000003}$ and use CRT to get the result modulo 1,000,000,006.

Note that when b is even the result modulo 2 is always 0. So, we only need to calculate the result modulo 500000003 and $\varphi(500000003) = \varphi(100000006)$, so this part is equal to the case when b is odd. The only difference is using CRT.

Adding everything together

After finding $y = b^{(2n, n)} \bmod 1000000006$, we can calculate $a^y \bmod 1000000007$ normally to get the final result.

Code :

```
#include<bits/stdc++.h>
#define ll long long int
int t;
ll a, b, n;
ll fact[200005];
ll mod = 1000000007;
long long int c_pow(ll i, ll j, ll mod)
{
    if (j == 0)
        return 1;
    ll d;
    d = c_pow(i, j / (long long)2, mod);
    if (j % 2 == 0)
        return (d*d) % mod;
    else
        return ((d*d) % mod * i) % mod;
}
ll InverseEuler(ll n, ll MOD)
{
    return c_pow(n, MOD - 2, MOD);
}
ll fact_14[1700][1700];
ll fact_B[150000];
ll min1(ll a, ll b)
{
    return a > b ? b : a;
}
void calc_fact()
{
    fact[0] = fact[1] = 1;
    ll tmd = 148721;
    for (int i = 2; i < 200003; ++i)
    {
        fact[i] = (fact[i - 1] * i);
        if (fact[i] >= (tmd))
```

```

fact[1] = (int);
}

}

ll fact_41[200005];
ll fact_41_p[200005];
void do_func()
{
    fact_41[0] = 1;
    fact_41_p[0] = 0;
    for (int i = 1; i < 200005; ++i)
    {
        ll y = i;
        fact_41_p[i] = fact_41_p[i - 1];
        while (y % 41 == 0)
        {
            y = y / 41;
            fact_41_p[i]++;
        }
        fact_41[i] = (y * fact_41[i - 1]) % 1681;
    }
}
ll fact_2[200005];
void do_func2()
{
    fact_2[0] = 1;
    for (int i = 1; i < 200005; ++i)
    {
        fact_2[i] = (i * fact_2[i - 1]) % 2;
    }
}
ll get_3rd(ll n, ll r, ll MOD)
{
    ll ans = (InverseEuler(fact[r], MOD) * InverseEuler(fact[n - r], MOD)) % MOD;
    ans = (fact[n] * ans) % MOD;
    return ans;
}
ll inverse2(ll m1, ll p1)

```

```

{
    ll i = 1;
    while (1)
    {
        if ((m1*i) % p1 == 1)
            return i;
        i++;
    }
}

ll chinese_remainder_2(ll n1, ll n2, ll n3)
{
    ll p1 = 2, p2 = 1681, p3 = 148721;
    ll m1, m2, m3;
    ll i1, i2, i3;
    ll m;
    ll ans;
    m = p1*p2*p3;
    m1 = m / p1; m2 = m / p2; m3 = m / p3;
    i1 = InverseEuler(m1, p1); i2 = inverse2(m2, p2); i3 = InverseEuler(m3, p3);
    //printf("i1 = %lld i2 = %lld\n",i1,i2);
    ans = (n1*m1*i1) % m + (n2*m2*i2) % m + (n3*m3*i3) % m;
    ans = ans%m;
    return ans;
    //printf("%d\n",ans);
}
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    calc_fact();
    do_func();
    do_func2();
    cin >> t;
    while (t--)
    {
        cin >> a >> b >> n;
        if (a == 0 && b == 0)
        {
}

```

```

    cout << "1\n";
    continue;
}

if (b == 0)
{
    cout << "1\n";
    continue;
}
ll a1 = (n == 0) ? 1 : 0;
ll a2 = (fact_41[2 * n] * inverse2(fact_41[n], 1681)) % 1681;
a2 = (a2 * inverse2(fact_41[n], 1681)) % 1681;
a2 = (a2 * c_pow(41, fact_41_p[2 * n] - 2 * fact_41_p[n], 1681)) % 1681;
ll a3 = get_3rd(2 * n, n, 148721);
//cout << a1 << " " << a2 << " " << a3 << "\n";
ll ans = chinese_remainder_2(a1, a2, a3);
if (ans == 0) ans = 500000002;
ll y1 = c_pow(b, ans, md - 1);
cout << y1 << "\n";
ll z = c_pow(a, y1, md);
cout << z << "\n";
}
return 0;
}

```

Best Method for nC_r

We want to compute $C(n,r)\%p$ where p is prime and $N,R \leq 10^8$:

```

#include<iostream>
using namespace std;
#include<vector>
/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD)
{
    long long x=1,y=a;
    while(b > 0)
    {
        if(b%2 == 1)

```

```

    {
        x=(x*y);
        if(x>MOD) x%=MOD;
    }
    y = (y*y);
    if(y>MOD) y%=MOD;
    b /= 2;
}
return x;
}
/* Modular Multiplicative Inverse
Using Euler's Theorem
 $a^{\phi(m)} \equiv 1 \pmod{m}$ 
 $a^{-1} = a^{m-2} \pmod{m}$  */
long long InverseEuler(int n, int MOD)
{
    return pow(n,MOD-2,MOD);
}
long long C(int n, int r, int MOD)
{
    vector<long long> f(n + 1,1);

    for (int i=2; i<=n;i++)
        f[i]= (f[i-1]*i) % MOD;
    return (f[n]*((InverseEuler(f[r], MOD) * InverseEuler(f[n-r],MOD))% MOD)) %
MOD;
}
int main()
{
    int n,r,p;
    while (~scanf("%d%d%d",&n,&r,&p))
    {
        printf("%lld\n",C(n,r,p));
    }
}

```

Modular Exponentiation :

We can now clearly see that this approach is very inefficient, and we need to come up with something better. We can take care of this problem in $O(\log_2 b)$ by using a technique called exponentiation by squaring. This uses only $O(\log_2 b)$ squarings and $O(\log_2 b)$ multiplications. This is a major improvement over the most naive method.

Code :

```
ans=1 //Final answer which will be displayed
while(b != 0) {
    /*Finding the right most digit of 'b' in binary form, if it is 1, then multiply
     the current value of a
    in ans.*/
    if(b&1) { //rightmost digit of b in binary form is 1.
        ans = ans*a;
        ans = ans%c; //at each iteration if value of ans exceeds then
        //reduce it to modulo c.
    }
    a = a*a;
    a %= c; //at each iteration if value of a exceeds then reduce
    it to modulo c.
    b >>= 1; //Trim the right-most digit of b in binary form.
}
```

Questions :

- Find sum of divisors of all the numbers from 1 to n . $n \leq 10^5$.

(SPOJ DIVSUM)

Here n is relatively small so we can precompute all the divisor sum using sieve like approach which runs in $O(NLOGN)$.

Code :

```
ll sum[5000005];
void sieve()
{
    F(i,1,5000002)
    {
        for(ll j=i;j<=5000002;j+=i) // every j has i as a divisor
        {
            sum[j] += i;
        }
    }
}
```

```

        }
    }
    // subtract number itself from sum if proper divisors are required
    F(i, 1, 5000002)
    sum[i] -= i;
}

```

2. Find sum of divisors of a number. $n \leq 10^{16}$.
(SPOJ-DIVSUM2)

Explanation :

Here our previous approach will fail since we cannot create such large array. Hence we have to factorize n in the form of $p_1^{k_1} * p_2^{k_2}...$

Now we can get the sum of divisors using the formula mentioned in this booklet above.

Code :

```

/*input
3
2
10
1000000000000000
*/
#include <bits/stdc++.h>
#include<stdio.h>
using namespace std;
#define F(i,a,b) for(ll i = a; i <= b; i++)
#define RF(i,a,b) for(ll i = a; i >= b; i--)
#define pii pair<ll,ll>
#define PI 3.14159265358979323846264338327950288
#define ll long long
#define ff first
#define ss second
#define pb(x) push_back(x)
#define mp(x,y) make_pair(x,y)
#define debug(x) cout << #x << " = " << x << endl
#define INF 1000000009
#define mod 1000000007
#define S(x) scanf("%d",&x)
#define S2(x,y) scanf("%d%d",&x,&y)

```

```

#define P(x) printf("%d\n",x)
#define all(v) v.begin(),v.end()
#define lim 100000002
vector<bool> mark(lim+2,1);
vector<ll> primes;
void sieve() // we need primes upto 10^8
{
    //ll times = 0;
    for(ll i=3;i<=lim;i+=2)
    {
        //times++;
        if(mark[i] == 1)
        {
            for(ll j=i*i ; j <= lim ;j += 2*i)
            {
                //times++;
                mark[j] = 0;
            }
        }
    }
    //debug(times);
    primes.pb(2);
    for(ll i=3;i<=lim;i+=2)
    {
        if(mark[i])
            primes.pb(i);
    }
}
ll power(ll a,ll b) // find a to the power b
{
    ll ans = 1ll;
    while(b > 0)
    {
        if(b&1)
            ans = ans*a;
        a = a*a;
        b /= 2;
    }
    return ans;

```

```

}

ll factorize(ll n) // find multiplication of all  $(p^{(k+1)-1})/(p-1)$  where k is
the power of p in n
{
    // exhaust powers of 2 first
    ll c = 0;
    while(n%2 == 0)
    {
        c++;
        n = n/2;
    }
    ll i=0 , ans = 1ll;
    if(c>0)
        ans = (power(2ll,c+1) - 1);
    ll times = 0;
    ll p = primes[0];
    while(p*p <= n)
    {
        //times++;
        if(n%p == 0) // if p is a prime factor of n
        {
            ll cnt = 0; // find power of p
            while(n%p == 0)
            {
                //times++;
                n /= p;
                cnt++;
            }
            // update ans;
            ll numerator = power(p,cnt+1) - 1;
            //debug(numerator);
            ll denom = p - 1;
            ll curans = numerator/denom;
            ans = ans * (curans);
        }
        //debug(n);
        if(n == 1)
    }
}

```

```

        break;
    i++;
    p = primes[i];
}
//debug(times);
if(n != 1)
    ans = ans * (n+1);
return ans;
}
int main()
{
    std::ios::sync_with_stdio(false);
    sieve();
    //cout<<primes.size(); //5761455 primes less than 10^8
    ll t;
    cin>>t;
    while(t--)
    {
        ll n;
        cin>>n;
        ll ans = factorize(n);
        ans -= n;
        cout<<ans<<endl;
    }
    return 0;
}

```

3. Find the value of $1! * 2! * 3! \dots N!$ modulo P where $P = 109546051211$.

FACTMUL SPOJ, Chinese Remainder Thm

Explanation :

The naive approach for calculating this value under modulo p will fail since $(a*b)\%p$ will overflow coz p is itself large.

Here is the trick :-

$$P = p_1 * p_2 \text{ where } p_1 = 186583 \text{ } p_2 = 587117$$

Let $a = 1! * 2! * 3! \dots N!$

```
x1 = a%p1
```

```
x2 = a%p2
```

This is a set of equations satisfying the CRT criteria hence we can calculate the value of a using CRT.

Code :

```
/*input
5
*/
#include <bits/stdc++.h>
#include<stdio.h>
using namespace std;
#define F(i,a,b) for(ll i = a; i <= b; i++)
#define RF(i,a,b) for(ll i = a; i >= b; i--)
#define pii pair<ll,ll>
#define PI 3.14159265358979323846264338327950288
#define ll long long
#define ff first
#define ss second
#define pb(x) push_back(x)
#define mp(x,y) make_pair(x,y)
#define debug(x) cout << #x << " = " << x << endl
#define INF 1000000009
#define mod 109546051211 // 186583*587117
#define S(x) scanf("%d",&x)
#define S2(x,y) scanf("%d%d",&x,&y)
#define P(x) printf("%d\n",x)
#define all(v) v.begin(),v.end()
ll power(ll a,ll b,ll m)
{
    ll ans = 1ll;
    while(b > 0)
    {
        if(b&1)
            ans = ans*a;
        a = a*a;
        ans% = m;
        a% = m;
    }
}
```

```

        b /= 2;
    }
    return ans;
}
int main()
{
    std::ios::sync_with_stdio(false);
    ll n;
    cin >> n;
    //use crt since MOD = p1*p2
    ll p1 = 186583ll;
    ll p2 = 587117ll;
    // find first value with respect to p1 and second value with respect to p2
    // ans_p1 = 1ll , ans_p2=1ll , curfactorial_p1 = 1ll , curfactorial_p2 = 1ll;
    F(1,2,n)
    {
        curfactorial_p1 = curfactorial_p1*i;
        curfactorial_p1 %= p1;
        curfactorial_p2 = curfactorial_p2*i;
        curfactorial_p2 %= p2;
        ans_p1 = ans_p1*curfactorial_p1;
        ans_p1 %= p1;
        ans_p2 = ans_p2*curfactorial_p2;
        ans_p2 %= p2;
    }
    //debug(ans_p1);
    //debug(ans_p2);
    // num[0] = p1 , num[1] = p2
    // rem[0] = ans_p1 rem[1] = ans_p2
    // pp[0] = p2 pp[1] = p1
    // prod = p1*p2
    // inv[i] = mpdular multiplicative inverse of pp[i] with respect to num[i]
    // inv[0] = inverse of p2 w.r.t p1 , inv[1] = inverse of p1 w.r.t p2
    // first remainder is ans_p1 and second remainder is ans_p2
    // (x%p1) = ans_p1
    // (x%p2) = ans_p2 , we can combine these two to find x
}

```

```

// x = rem[0]*inv[0]*pp[0] + rem[1]*inv[1]*pp[1]
// inv_zero = power(p2,p1-2ll,p1); // fermats
// inv_first = power(p1,p2-2ll,p2); // fermats
// ans = (((ans_p1*inv_zero)%mod)*(p2%mod))%mod +
((ans_p2*inv_first)%mod)*p1%mod)%mod;
ans %= mod;
cout<<ans<<endl;
return 0;
}

```



TRY YOURSELVES

<http://www.spoj.com/problems/MAIN74/> //Find first few values and observe pattern
<http://www.spoj.com/problems/DIVSUM/> // precomputation or multiplicative formula
<http://www.spoj.com/problems/DIVSUM2/> // multiplicative formula
<http://www.spoj.pl/problems/NDIVPHI/> // can be solved only using BIG INTEGER or in PYTHON
<http://www.codechef.com/problems/THREEDIF> // very simple
<http://www.spoj.com/problems/LCPCP2/> // very simple
<http://www.spoj.com/problems/GCD2/> // tricky
<http://www.spoj.com/problems/FINDPRM/>
<http://www.spoj.com/problems/TDKPRIME/> // simple sieve and precompute
<http://www.spoj.com/problems/TDPRIMES/> // same as TDKPRIME
<http://www.spoj.com/problems/PRIME1/> //segmented sieve
<http://www.spoj.com/problems/FACTMUL/> // CRT
<http://www.spoj.com/problems/FACTCG2/> // factorization
<http://www.spoj.com/problems/ALICESIE/> // formula
<http://www.spoj.com/problems/AMR10C/> // factorization
<http://www.spoj.com/problems/DCEPC11B/> // Wilson Theorem
<http://www.codechef.com/problems/SPOTWO>
<http://www.spoj.com/problems/DCEPC13D/> // CRT + LUCAS + FERMAT
<http://www.spoj.com/problems/CUBEFR/>
<http://www.spoj.com/problems/NFACTOR/>
<http://www.spoj.com/problems/CSQUARE/>
<http://www.spoj.com/problems/CPRIME/>
<http://www.spoj.com/problems/ANARC09C/>
<http://www.spoj.com/problems/GCDEX/> read this :-
<https://discuss.codechef.com/questions/72953/a-dance-with-mobius-function>

Number Theory

<http://www.spoj.com/problems/AMR11E/> // easy sieve

<https://www.hackerearth.com/challenge/competitive/code-monk-number-theory-i/problems/>

<https://www.hackerearth.com/challenge/competitive/code-monk-number-theory-ii/problems/>

<https://www.hackerearth.com/challenge/competitive/code-monk-number-theory-iii/problems/>

Advanced Problem :- <https://www.hackerrank.com/challenges/ncr/problem>

SELF STUDY NOTES

4

Binary Search

Divide and Conquer

Searching an element in a Sorted Sequence

```
binary_search(A, target):
    lo = 1, hi = size(A)
    while lo <= hi:
        mid = lo + (hi-lo)/2
        if A[mid] == target:
            return mid
        else if A[mid] < target:
            lo = mid+1
        else:
            hi = mid-1
```

First Occurrence of an Element

```
FirstOccur(A, target):
    lo = 1, hi = size(A), result=-1;
    while lo <= hi:
        mid = lo + (hi-lo)/2
        if A[mid] == target:
            result=mid; high=mid-1;
        else if A[mid] < target:
            lo = mid+1
        else:
            hi = mid-1
    return res;
```

Last Occurrence of an Element

```
LastOccur(A, target):
    lo = 1, hi = size(A), result=-1;
    while lo <= hi:
        mid = lo + (hi-lo)/2
```

Binary Search

```

if A[mid] == target:
    result=mid; low=mid+1;
else if A[mid] < target:
    lo = mid+1
else:
    hi = mid-1
return res;

```

Beyond Sorted arrays

Binary search obviously works on searching for elements in a sorted array. But if you think about the reason why it works is because the array itself is monotonic (either increasing or decreasing). So, if you are at a particular position, you can make a definite call whether the answer lies in the left part of the position or the right part of it. But Most importantly you need to know the range [start,end].

Similar thing can be done with monotonic functions (monotonically increasing or decreasing) as well.

Lets say we have $f(x)$ which increases when x increases.

So, given a problem of finding x so that $f(x) = p$, I can do a binary search for x .

/Take Example/

Therefore

1. if $f(\text{current_position}) > p$, then I will search for values lower than current position.
2. if $f(\text{current_position}) < p$, then I will search for values higher than current position
3. if $f(\text{current_position}) = p$, then I have found my answer.

Optimisation Problems

Consider an optimisation problem where you need to minimise a quantity X satisfying some constraints such that:

- If X satisfies the constraint, anything more than X will satisfy the constraint too.
- For a given value of X , you know how to check whether the constraint is satisfied.

Painter's Partition Problem

You have to paint N boards of length $\{A_0, A_1, A_2, A_3 \dots A_{N-1}\}$. There are K painters available and you are given how much time a painter takes to paint 1 unit of board. You have to get this job done as soon as possible under the constraints that any painter will only paint contiguous sections of board.

/Take example/

Observations :

- The lowest possible value for costmax must be the maximum element in A (name this as lo).
- The highest possible value for costmax must be the entire sum of A , (name this as hi).

- As costmax increases, x decreases. The opposite also holds true.

```

int painterNum(vector<int> &C, long long X){
    long long sum = 0;
    long long num = 1;
    for(auto c: C){
        if(sum + c > X){
            sum = c;num++;
        }else sum += c;
    }
    return num;
}

int paint(int A, int B, vector<int> &C) {

    int high=0,low=0;
    for(int x: C){low = max(x, low); high += x;}
    while(low < high){
        long long mid = low + (high-low)>>1;
        if(painterNum(C, mid) <= A)high = mid;
        else low = mid+1;
    }

    return low;
}

```

Allocate Books

N number of books are given.

The ith book has P_i number of pages.

You have to allocate books to M number of students so that maximum number of pages allotted to a student is minimum. A book will be allocated to exactly one student. Each student has to be allocated at least one book. Allotment should be in contiguous order, for example: A student cannot be allocated book 1 and book 3, skipping book 2.

Return -1 if a valid assignment is not possible

Problem :**Aggressive Cows :**

<http://www.spoj.com/problems/AGRCOW/>

Approach:

Define the following function:

$\text{Func}(x) = 1$ if it is possible to arrange the cows in stalls such that the distance between any two cows is at least x

$\text{Func}(x) = 0$ otherwise

The problem satisfies the monotonicity condition necessary for binary search. Why??

Check that if $\text{Func}(x)=0$, $\text{Func}(y)=0$ for all $y > x$ and if $\text{Func}(x)=1$,

$\text{Func}(y)=1$ for all $y < x$

Find low and high values

$\text{Func}(0)=1$ trivially since the distance between any two cows is at least 0. Also, since we have at least two cows, the best we can do is push them towards the stalls at the end - so there is no way we can achieve better. Hence $\text{Func}(\text{maxbarnpos}-\text{minbarnpos}+1)=0$.

```
#include <bits/stdc++.h>
using namespace std;
int n, c;
int func(int num, int array[])
{
    int cows=1, pos=array[0];
    for (int i=1; i<n; i++)
    {
        if (array[i]-pos>=num)
        {
            cows++;
            if (cows==c)
                return 1;
            pos=array[i];
        }
    }
    return 0;
}
int bs(int array[])
```

```
{  
    int ini=0,last=array[n-1],max=-1;  
    while (last>ini)  
    {  
        int mid=(ini+last)/2;  
        if (func(mid,array)==1)  
        {  
            if (mid>max)  
                max=mid;  
            ini=mid+1;  
        }  
        else  
            last=mid;  
    }  
    return max;  
}  
int main()  
{  
    int t;  
    scanf("%d",&t);  
    while (t-)  
    {  
        scanf("%d %d",&n,&c);  
        int array[n];  
        for (int i=0; i<n; i++)  
            scanf("%d",&array[i]);  
        sort(array,array+n);  
        int k=bs(array);  
        printf("%dn",k);  
    }  
    return 0;  
}
```



Time to Try

- EKO (Spoj)
- PRATA (Spoj)
- BEAUTIFUL TRIPLETS (Hackerearth)

SELF STUDY NOTES

5

Greedy Algorithm

Greedy Algorithms are one of the most intuitive algorithms. Whenever we see a problem we first try to apply some greedy strategy to get the answer (we humans are greedy, aren't we :P ?).

Greedy approaches are quite simple and easy to understand/formulate. But many times the proving part might be difficult.

- Greedy Algorithm always makes the choice that looks best at the moment.
- You hope that by choosing a local optimum at each step, you will end up at a global optimum.

Counting Money

Problem Statement: Suppose you want to count out a certain amount of money, using the fewest possible notes or coins.

Greedy Approach

At each step, take the largest possible note or coin that does not overshoot the sum of money and include it in the solution.

Example :

To make Rs 39 with fewest possible notes or coins,

Rs 10 note, to make 29

Rs 10 note, to make 19

Rs 10 note, to make 9

Rs 5 note, to make 4

Rs 2 coin, to make 2

Rs 2 coin, to make 0

Total is 4 notes and 2 coins, which is the optimum solution.

Note : For Indian currency, the greedy algorithm, even after Demonetization always gives the optimum solution.

Is this true for any currency?

Now that Donald Trump is the new President of US, he decides to do something extraordinary like PM Modi. So he decides to change all the currency notes to 1 dollars, 7 dollars and 10 dollars.

Greedy Algorithm

Suppose you went to US and used the same greedy approach to count minimum numbers of notes and coins in exchange for a Burger which costs \$15.

To make \$15:

1 \$10 note

5 \$1 notes

Total = 6 notes

Can we do better than 6?

\$7 + \$7 + \$1 – Only 3 notes required!

PROBLEMS

BUSYMAN

Activity Scheduling Problem

<http://www.spoj.com/problems/BUSYMAN/>

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
struct cmp
{
    bool operator()(pair<int,int> l, pair<int,int> r)
    {
        return l.second < r.second;
    }
}cmp;
int main()
{
    int t,n,s,e,res;
    scanf("%d",&t);
    while(t--)
    {
        res = 1;
        vector<pair<int,int> > activity;
        scanf("%d",&n);
        for(int i = 0;i<n;i++)
        {
            scanf("%d %d",&s,&e);
```

```

    activity.push_back(make_pair(s,e));
}
//Sort the activities according to their finish time
sort(activity.begin(),activity.end(),cmp);
//fin denotes the finish time of the chosen activity
//i.e. activity with least finish time
int fin = activity[0].second;
for(int i = 1;i<activity.size();i++)
{
    //To find the next compatible activity with
    //least finish time
    //Find the first activity whose start time
    //is more than finish time of previous activity
    if(activity[i].first >= fin)
    {
        //update the finish time as the finish time
        //of this activity
        fin = activity[i].second;
        //Since we have chosen a new activity,
        //increment the count by 1
        res++;
    }
}
printf("%d\n",res);
}
return 0;
}

```

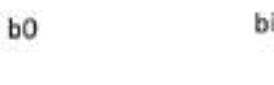
□ CONNECTING WIRES

- There are n white dots and n black dots, equally spaced, in a line.
- You want to connect each white dot with some one black dot, with a minimum total length of "wire".

Greedy Approach : Suppose you have a sorted list of the white dot positions and the black dot positions. Then you should match the i -th white dot with the i -th black dot.

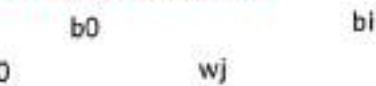
Why should greedy work?

Let b_0 and w_0 be the first black and white balls respectively and let b_i and w_j be the next white and black balls,

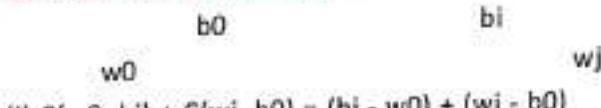
Case 1: $w_0 < b_0$ and $w_j < b_0$ 

- (i) $C(w_0, b_i) + C(w_j, b_0) = (b_i - w_0) + (b_0 - w_j)$
- (ii) $C(w_0, b_0) + C(w_j, b_i) = (b_0 - w_0) + (b_i - w_j)$

Both are same.

Case 2: $w_0 < b_0$ and $b_0 < w_j < b_i$ 

- (i) $C(w_0, b_i) + C(w_j, b_0) = (b_i - w_0) + (w_j - b_0)$
- (ii) $C(w_0, b_0) + C(w_j, b_i) = (b_0 - w_0) + (b_i - w_j)$
- (i) - (ii) = $2(w_j - b_0)$ is extra

Case 3: $w_0 < b_0$ and $b_i < w_j$ 

- (i) $C(w_0, b_i) + C(w_j, b_0) = (b_i - w_0) + (w_j - b_0)$
- (ii) $C(w_0, b_0) + C(w_j, b_i) = (b_0 - w_0) + (w_j - b_i)$
- (i) - (ii) = $2(b_i - b_0)$ is extra

Hence, in all the cases connecting 1st b to 1st w and 2nd b to 2nd w yeilds more optimal solution.

□ BIASED STANDINGS

In a competition, each team is be able to enter their preferred place in the ranklist. Suppose that we already have a ranklist. For each team, compute the distance between their preferred place and their place in the ranklist. The sum of these distances will be called the badness of this ranklist. Find one ranklist with the minimal possible badness.

<http://www.spoj.com/problems/BAISED/>

This question is similar to the connecting wires problem, where the dseired ranks denote the black balls and the actual ranks denote the white balls. Hence, we can apply the greedy approach by sorting the desired ranks and calculating the distance between i-th desired rank and i-th actula rank.

```

Code :
#include <bits/stdc++.h>
using namespace std;
#define abs(a,b) a > b ? a - b : b - a
#define ll long long
int main()
{
    int t;
    cin>>t;
    while(t--)
    {
        vector<int> v;
        string name;
        ll sum=0;
        cin>>n;
        for(int i=0;i<n;i++)
        {
            cin>>name;
            cin>>temp;
            //Insert the desired rank in vector v
            v.push_back(temp);
        }
        //Sort the vector containing the ranks
        sort(v.begin(),v.end());
        //This is the greedy approach
        for(int i=1;i<=n;i++)
        {
            //Take the difference between the i-th actual rank
            //and the i-th desired rank
            sum += abs(v[i-1],i);
        }
        printf("%lld\n",sum);
    }
    return 0;
}

```

$T = O(n \log n)$

Greedy Algorithm

Can we do better?

```
#include <bits/stdc++.h>
using namespace std;
#define abs(a,b) a > b ? a - b : b - a
#define ll long long
int arr[100000+10];
int main()
{
    int t,n,temp;
    cin>>t;
    while(t--)
    {
        memset(arr,0,sizeof arr);
        string name;
        ll sum=0;
        cin>>n;
        for(int i=0;i<n;i++)
        {
            cin>>name;
            cin>>temp;
            //Increment the desired rank index in array arr
            arr[temp]++;
        }
        int pos = 1;
        //This is the greedy approach
        for(int i=1;i<=n;i++)
        {
            //If the i-th rank was desired by atleast one team
            //Assign and increment the actual rank index 'pos'
            //till this rank is desired
            while(arr[i])
            {

```

```

    sum += abs(pos,i);
    arr[i]--;
    pos++;
}
}
printf("%lld\n",sum);
}
return 0;
}

```

$T = O(n)$

□ LOAD BALANCER

Rebalancing proceeds in rounds. In each round, every processor can transfer at most one job to each of its neighbors on the bus. Neighbors of the processor i are the processors $i-1$ and $i+1$ (processors 1 and N have only one neighbor each, 2 and $N-1$ respectively). The goal of rebalancing is to achieve that all processors have the same number of jobs. Determine the minimal number of rounds needed to achieve the state when every processor has the same number of jobs, or to determine that such rebalancing is not possible.

<http://www.spoj.com/problems/BALIFE/>

Greedy Approach: First find the final load that each processor will have. We can find it by $\text{sum}(\text{arr}[1], \text{arr}[2], \dots, \text{arr}[n]) / n$, let us call it load .

So in the final state, each of the processor will have final load = load .

For each index i from 1 to $n-1$, create a partition of $(1\dots i)$ and $(i+1\dots n)$ and find the amount of load that is to be shared between these two partitions. The answer will be maximum of all the loads shared between all the partitions with i varying from 1 to $n-1$.

Example :

Initial state: 4, 8, 12, 16

Final state: 10 10 10 10

$i = 1$:

partition: (4) (8,12,16)

Load \rightarrow (4) needs 6 and (8,12,16) needs to give 6.

max_load = 6

$i = 2$:

Greedy Algorithm

```
partition: (4,8) (12,16)
Load → (4,8) needs 8 and (12,16) needs to give 8.
max_load = 8
t = 3:
partition: (4,8,12) (16)
Load → (4,8,12) needs 6 and (16) needs to give 6.
max_load = 6
Final answer = 8
```

Why does it work?

In all the partitions where less than max_load is transferred, we can internally transfer the load between these partitions when max_load is being transferred between the max_load partition.

t = 2s:

when load = 2 is transferred between (4,8) and (12,16)
we can transfer load = 2 between (8) → (4) and (16) → (12)

State: 6 8 12 14

t = 4s:

when load = 2 is transferred between (6,8) and (12,14)
we can transfer load = 2 between (8) → (6) and (14) → (12)

State: 8 8 12 12

t = 6s:

when load = 2 is transferred between (8,8) and (12,12)

State: 8 10 10 12

t = 8s:

when load = 2 is transferred between (8,10) and (10,12)
we can transfer load = 2 between (10) → (8) and (12) → (10)

State: 10 10 10 10

Code :

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
```

```

int arr[9000], n, i, val, diff;
while(1)
{
    int max_load = 0, load = 0;
    cin >> n;
    if(n == -1)
        break;
    for(int i = 0; i < n; i++)
    {
        cin >> arr[i];
        load += arr[i];
    }
    //If we cannot divide the load equally
    if(load % n)
    {
        cout << -1 << endl;
        continue;
    }
    //Find the load that is to be divided equally
    load /= n;
    //Greedy step
    for(int i = 0; i < n; i++)
    {
        //At each iteration, find the value
        //of difference between final load to be assigned
        //and current load
        //Keep adding this difference in 'diff'
        diff += (arr[i] - load);
        //If the net difference is negative i.e.
        //we need diff amount till i-th index
        if(diff < 0)
            max_load = max(max_load, -1 * diff);
    }
}

```

```
//If diff is positive i.e. we have to  
//give diff amount to (n-i) processors  
else  
    max_load = max(max_load, diff);  
//calculate the max of load that can be given or  
//taken at each iteration.  
}  
cout<<max_load<<endl;  
}  
return 0;  
}
```

$T = O(n)$

□ DEFENSE OF A KINGDOM, SPOJ

Problem Statement

Given H,W of a field, and location of towers which guard the horizontal and the vertical lines corresponding to their positions. Find out the largest unbounded area(white rect). Refer the diagram on spoj.

<http://www.spoj.com/problems/DEFKIN/>

Greedy Approach: Given w, h as width and height of the playing field, and the coordinates of the towers as $(x_1, y_1) \dots (x_N, y_N)$, split the coordinates into two lists $x_1 \dots x_N, y_1 \dots y_N$, sort both of those coordinate lists.

Then calculate the empty spaces, e.g. $dx[] = \{x_1, x_2 - x_1, \dots, x_N - x_{N-1}, (w + 1) - x_N\}$. Do the same for the y coordinates: $dy[] = \{y_1, y_2 - y_1, \dots, y_N - y_{N-1}, (h + 1) - y_N\}$. Multiply $\max(dx)-1$ by $\max(dy)-1$ and you should have the largest uncovered rectangle. You have to decrement the delta values by one because the line covered by the higher coordinate tower is included in it, but it is not uncovered.

Code :

```
#include<iostream>  
#include<algorithm>  
using namespace std;  
int point_x[40000+10], point_y[40000+10];  
int main()  
{
```

```

int t,w,h,n,x,y;
scanf("%d",&t);
while(t--)
{
    scanf("%d %d %d",&w,&h,&n);
    for(int i = 0;i<n;i++)
    {
        scanf("%d %d",&point_x[i],&point_y[i]);
    }
    //sort the x-coordinates of the list
    sort(point_x, point_x + n);
    //sort the y-coordinates of the list
    sort(point_y, point_y + n);
    //dx --> maximum uncovered tiles in x coordinate
    //dy --> maximum uncocered tiles in y coordinate
    //Initially dx and dy are the first guars's position
    int dx = point_x[0],dy = point_y[0];
    //calculate the maximum uncovered gap
    //in x and y coordinate
    for(int i = 1;i<n;i++)
    {
        dx = max(dx,point_x[i] - point_x[i-1]);
        dy = max(dy,point_y[i] - point_y[i-1]);
    }
    dx = max(dx, w + 1 - point_x[n-1]);
    dy = max(dy, h + 1 - point_y[n-1]);
    printf("%d\n",((dx-1) * (dy-1)));
}
return 0;
}

```

$T = O(n \log n)$

□ QUES. CHOPSTICKS

Given N sticks of length $L[1], L[2], \dots, L[N]$ and a positive integer D. Two sticks can be paired if the difference of their length is at most D.

Greedy Algorithm

Find the max number of pairs.

<https://www.codechef.com/problems/TACHSTCK>

Greedy Approach :

- ❑ Sort the list of sticks according to their lengths.
- ❑ If $L[1]$ and $L[2]$ cannot be paired, $L[1]$ is useless.
- ❑ Else pair $L[1]$ and $L[2]$ and remove them from the list.
- ❑ Repeat steps 2-3 till the list become empty.

Why does this works?

- ❑ By pairing starting stick to its immediate next, we have max number of options left for next pairing.
- ❑ If $L[1]$ and $L[2]$ can be paired, but instead we pair $(L[1], L[M])$ and $(L[2], L[N])$.

Code :

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
typedef long long ll;
int main()
{
    ll n,d,a;
    vector<ll> v;
    cin>>n>>d;
    for(int i = 0;i<n;i++)
    {
        cin>>a;
        v.push_back(a);
    }
    //Sort the list of sticks
    sort(v.begin(),v.end());
    int res = 0;
    for(int i = 0;i<n-1;)
    {
        //If the adjacent difference is less than d
        //we can make a pair
        //Now remove these two sticks from the list
    }
}
```

```

if(v[i+1] - v[i] <= d)
{
    res++;
    i+=2;
}
//if we cannot make a pair with adjacent stick
//This stick is useless, remove this stick from the list
else
    i++;
}
cout<<res<<endl;
return 0;
}

```

$T = O(n \log n)$

□ EXPEDITION, SPOJ

A damaged truck needs to travel a distance of L . The truck leaks one unit of fuel for every unit distance it travels. There are N ($\leq 10^4$) fuel stops on the path, what is the minimum number of refueling stops that the truck needs to make given that it has P amount of fuel initially.

<http://www.spoj.com/problems/EXPEDI/>

The first observation is, if you want to reach the city, say, point L , you have to ensure that every single point between the current position and city must also be reachable.

Now, the task is to minimize the number of stoppages for fuel, which is at most 10000. So, we sort the fuel stations, and start from current position. For every fuel station, if we want to reach it, we must have fuel f more than or equal to the distance d . Also, using the larger capacities will always reduce the number of stations we must stop.

How does greedy work?

If we make sure that we only stop at the largest capacity fuel stations upto the point where the fuel capacity of truck becomes 0, number of stops will be minimized.

Example:

Initial capacity = 11

Fuel Stations: 3 8 10 12

Capacity: 4 10 2 3

Initially, we don't have enough fuel to reach 12.
So we can reach upto maximum 11th city.
In order to minimize our stops, we will stop at 8th city
where fuel capacity is maximum.

Code:

```
#include<iostream>
#include<vector>
#include<queue>
#include<algorithm>
using namespace std;
bool cmpr(pair<int,int> l,pair<int,int> r)
{
    return l.first > r.first;
}
int main()
{
    int n,t,x,d,f,D,F,prev = 0;
    scanf("%d",&t);
    while(t--)
    {
        int flag = 0,ans = 0;
        vector<pair<int,int>> v;
        priority_queue<int> pq;
        scanf("%d",&n);
        for(int i = 0;i<n;i++)
        {
            scanf("%d %d",&d,&f);
            //Insert the city index and fuel capacity
            v.push_back(make_pair(d,f));
        }
        //Sort the cities according to their location
        sort(v.begin(),v.end(),cmpr);
        scanf("%d %d",&D,&F);
        //Calculate the difference between the current city and the
        //destination i.e. v[i] = j means that we need to travel j
```

```

//units to reach our destination
for(int i = 0;i<n;i++)
{
    v[i].first = D - v[i].first;
}
//prev denotes the previous city visited
prev = 0;
//x will denote the current city that we are in
x = 0;
while(x < n)
{
    //cout<<x<<" "<<F<<" "<<v[x].first<<" "<<prev<<" "<<endl;
    //If we have enough fuel to travel from prev to current city
    //Push this fuel station to priority_queue
    //Reduce the amount of fuel used
    //update the previous city
    if(F >= (v[x].first - prev))
    {
        F -= (v[x].first - prev);
        pq.push(v[x].second);
        prev = v[x].first;
    }
    //If we dont have enough fuel to visit
    //the current city
    //Find the max capacity fuel station between
    //prev and current city and use it to refuel
    else
    {
        //If no fuel station is left
        //i.e. we have used all of
        //the fuel stations and still not able
        //to reach the city
        //return FAIL!
        if(pq.empty())
        {

```

```
flag = 1;
break;
}
//Increment the fuel capacity of truck
//by the maximum fuel station capacity
F += pq.top();
//Remove that fuel station from heap
pq.pop();
//Increment the number of used fuel
//station by 1
ans++;
continue;
}
//If we have visited the current city
//Visit next city
x++;
}
if(flag)
{
printf("-1\n");
continue;
}
//Find the distance between the destination
//and last city
//Check if it is possible to visit the destination
//from the last city.
D = D - v[n-1].first;
if(F >= D)
{
printf("%d\n",ans);
continue;
}
while(F < D)
{
if(pq.empty()){


```

```

    flag = 1;
    break;
}
F += pq.top();
pq.pop();
ans++;
}
if(flag){
    printf("-1\n");
    continue;
}
printf("%d\n",ans);
}
return 0;
}

```

$T = O(N \log N)$

□ GREEDY KNAPSACK PROBLEM, CODECHEF

You are given N items, each item has two parameters: the weight and the cost. Let's denote M as the sum of the weights of all the items. Your task is to determine the most expensive cost of the knapsack, for every capacity $1, 2, \dots, M$. The capacity C of a knapsack means that the sum of weights of the chosen items can't exceed C .

<https://www.codechef.com/problems/KNPSK>

The key observation here is that weight of each item is either 1 or 2.

Greedy Approach: Greedily pickup the most costliest item with weight ≤ 2 , which can be taken in the knapsack.

Case 1: W is even

Select the most expensive item with sum of weight = 2. This can be done in two ways:

- Take the most expensive item of weight 2
- Or take at most two most expensive item of weight 1

Note that after picking up the most expensive item with sum of weight ≤ 2 , we will remove the item taken and will recursively select the items to fill the most expensive elements.

Case 2: W is odd

We can simply select the most expensive item of weight 1. Now, we don't consider this item again. Now we have to select the most expensive weights from the remaining items. Since W-1 is even, we can solve this problem similar to case 1.

Example:

1 7 1 100 2 70

2 44 1 56 2 44

1 56 1 33 2 1

2 1 1 18

1 18

1 33

2 70

Case 1: Even

$$W = 2: (1,100) + (1,56) = 156$$

$$W = 4: 156 + (2,70) = 226$$

$$W = 6: 226 + (1,33) + (1,18) = 277$$

$$W = 8: 277 + (2,44) = 321$$

$$W = 10: 321 + (2,1) = 322$$

Case 2: Odd

$$W = 1: (1,100) = 100$$

$$W = 3: 100 + (1,56) + (1,33) = 189$$

$$W = 5: 189 + (2,70) = 259$$

$$W = 7: 259 + (2,44) = 303$$

$$W = 9: 303 + (1,18) = 321$$

Code :

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<stdio.h>
using namespace std;
long long cost[200000+10];
int main()
{
    int n,w,c,m,W = 0;
```

```

//Separate one and two for even and odd cases
vector<int> one,two,One,Two;
scanf("%d",&n);
for(int i = 0;i<n;i++)
{
    scanf("%d %d",&w,&c);
    //Insert weight 1 items in one vector
    if(w == 1)
        one.push_back(c);
    //Insert weight 2 items in two vector
    else
        two.push_back(c);
    W += w;
}
//sort the two vectors
sort(one.begin(),one.end());
sort(two.begin(),two.end());
One = one;
Two = two;
long long sum = 0,cur = 0,res = 0;
//even case
for(int i = 2;i<=W;i+=2)
{
    //res1 --> when we take atmost two 1 wight items
    //res2 --> when we take single 2 weight item
    long long res1 = 0, res2 = 0;
    //calc res2
    if(two.size() > 0)
    {
        res2 = two[two.size()-1];
    }
    //calculate res1
    if(one.size() > 1)
    {
        res1 = one[one.size()-1] + one[one.size()-2];
    }
}

```

```
else if(one.size() > 0)
{
    res1 = one[one.size() - 1];
}
//if res2 > res1 remove the 2 weight item
if(res2 > res1)
{
    two.pop_back();
    res += res2;
}
//remove at most two 1 weight items
else
{
    if(one.size() > 1)
    {
        one.pop_back();
        one.pop_back();
    }
    else
        one.pop_back();
    res += res1;
}
//update the max cost for weight i
cost[i] = res;
}
//odd case
//subtract 1 to make the weight sum even
res = 0;
//Find the most expensive 1 weight item
//and remove it from the list
if(One.size() > 0){
    res = One[One.size()-1];
    One.pop_back();
}
cost[1] = res;
//Similar to even case
```

```

for(int i = 3; i <= W; i += 2)
{
    long long res1 = 0, res2 = 0;
    if(Two.size() > 0)
    {
        res2 = Two[Two.size() - 1];
    }
    if(One.size() > 1)
    {
        res1 = One[One.size() - 1] + One[One.size() - 2];
    }
    else if(One.size() > 0)
    {
        res1 = One[One.size() - 1];
    }
    if(res2 > res1)
    {
        Two.pop_back();
        res += res2;
    }
    else
    {
        if(One.size() > 1)
        {
            One.pop_back();
            One.pop_back();
        }
        else
            One.pop_back();
        res += res1;
    }
    cost[i] = res;
}
for (int i = 1; i <= W; i++)
{
    if (i > 1)

```

```
    printf(" ");
    printf("%lld", cost[i]);
}
printf("\n");
return 0;
}
```

$T = O(n \log n)$

Problems to Try :

1. **Fractional Knapsack** : Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack. We can break items for maximizing the total value of knapsack.
2. **Huffman Coding** : Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. Read about this problem and implement it yourself.
3. **Maximum Circles (HackerBlocks)** : There are n circles arranged on x-y plane. All of them have their centers on x-axis. You have to remove some of them, such that no two circles are overlapping after that. Find the minimum number of circles that need to be removed.
4. **Maximum Unique Segment (Codechef)** : You are given 2 arrays $W = (W_1, W_2, \dots, W_N)$ and $C = (C_1, C_2, \dots, C_N)$ with N elements each. A range $[l, r]$ is *unique* if all the elements C_l, C_{l+1}, \dots, C_r are unique (ie. no duplicates). The *sum* of the range is $W_l + W_{l+1} + \dots + W_r$. You want to find an *unique* range with the maximum *sum* possible, and output this sum.
5. **Station Balance - UVa 410**
Read and solve the problem statement Online
<https://uva.onlinejudge.org/external/4/410.pdf>
6. **DIE HARD (Spoj)**
7. **GERGOVIA (Spoj)**
8. **SOLDIER (Spoj)**
9. **CHOCOLA (Spoj)**
10. **CMIYC (Spoj)**



Scanned by CamScanner

[REDACTED]



6

Recursion & Backtracking

Introduction :

Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. Backtracking is also known as depth-first search or branch and bound. Recursion technique is always used to solve backtracking problems. In this article, we will see the concept of recursion and backtracking.

Recursion :

When a function calls itself, it's called Recursion. It will be easier for those who have seen the movie Inception. Leonardo had a dream, in that dream he had another dream, in that dream he had yet another dream, and that goes on. So it's like there is a function called dream(), and we are just calling it in itself.

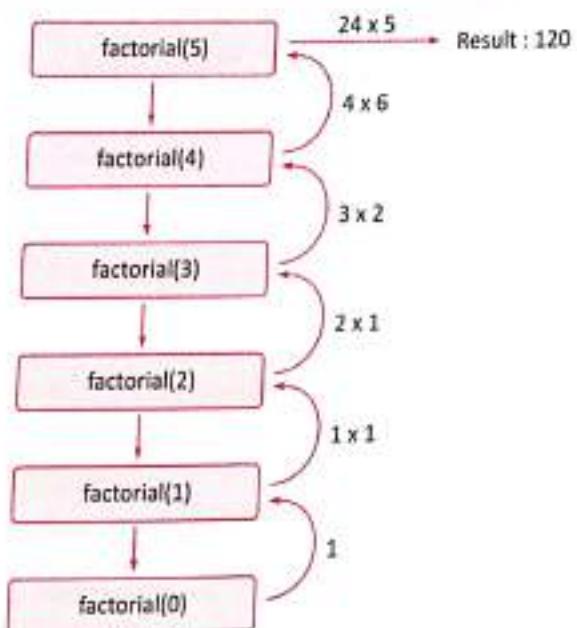
```
function dream()
{
    print "Dreaming";
    dream();
}
```

Recursion is useful in solving problems which can be broken down into smaller problems of the same kind. But when it comes to solving problems using Recursion there are several things to be taken care of. Let's take a simple example and try to understand those. Following is the pseudo code of finding factorial of a given number X using recursion.

```
function factorial(x)
{
    if (x== 0)          // base case
        return 1;
    return x*factorial(x-1); // break into smaller problem(s)
}
```



The following procedure shows how it works for factorial(5)



Base cases for Recursive Program :

Any recursive method must have a terminating condition. Terminating condition is one for which the answer is already known and we just need to return that. Ex. For the factorial problem, we know that $\text{factorial}(0) = 1$, so when x is 0 we simply return 1, otherwise we break into smaller problem i.e. find factorial of ($x-1$). If we don't include a Base Case, the function will keep calling itself, and ultimately will result in stack overflow. For example, the dream() function given above has no base case. If you write a code for it in any language, it will give a runtime error.

Limitation of Recursive Call :

There is an upper limit to the number of recursive calls that can be made. To prevent this make sure that your base case is reached before stack size limit exceeds.

So, if we want to solve a problem using recursion, then we need to make sure that:

- The problem can be broken down into smaller problems of same type.
- Problem has some base case(s).
- Base case is reached before the stack size limit exceeds.

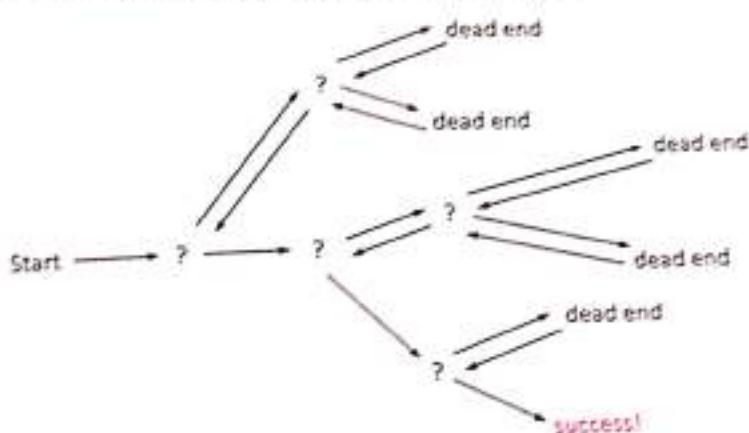
Backtracking :

When we solve a problem using recursion, we break the given problem into smaller ones. Let's say we have a problem PROB and we divided it into three smaller problems P1, P2 and P3. Now it may be the case that the solution to PROB does not depend on all the three subproblems, in fact we don't even know on which one it depends.

Let's take a situation. Suppose you are standing in front of three tunnels, one of which is having a bag of gold at its end, but you don't know which one. So you'll try all three. First go in tunnel 1, if that is not the one, then come out of it, and go into tunnel 2, and again if that is not the one, come out of it and go into tunnel 3. So basically in backtracking we attempt solving a subproblem, and if we don't reach the desired solution, then undo whatever we did for solving that subproblem, and again try solving another subproblem.

Basically **Backtracking** is an algorithmic paradigm that tries different solutions until finds a solution that "works". Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once.

Note : We can solve this by using recursion method.



Problems based on Backtracking :

1. **N-Queen Problem :** Given a chess board having $N \times N$ cells, we need to place N queens in such a way that no queen is attacked by any other queen. A queen can attack horizontally, vertically and diagonally.

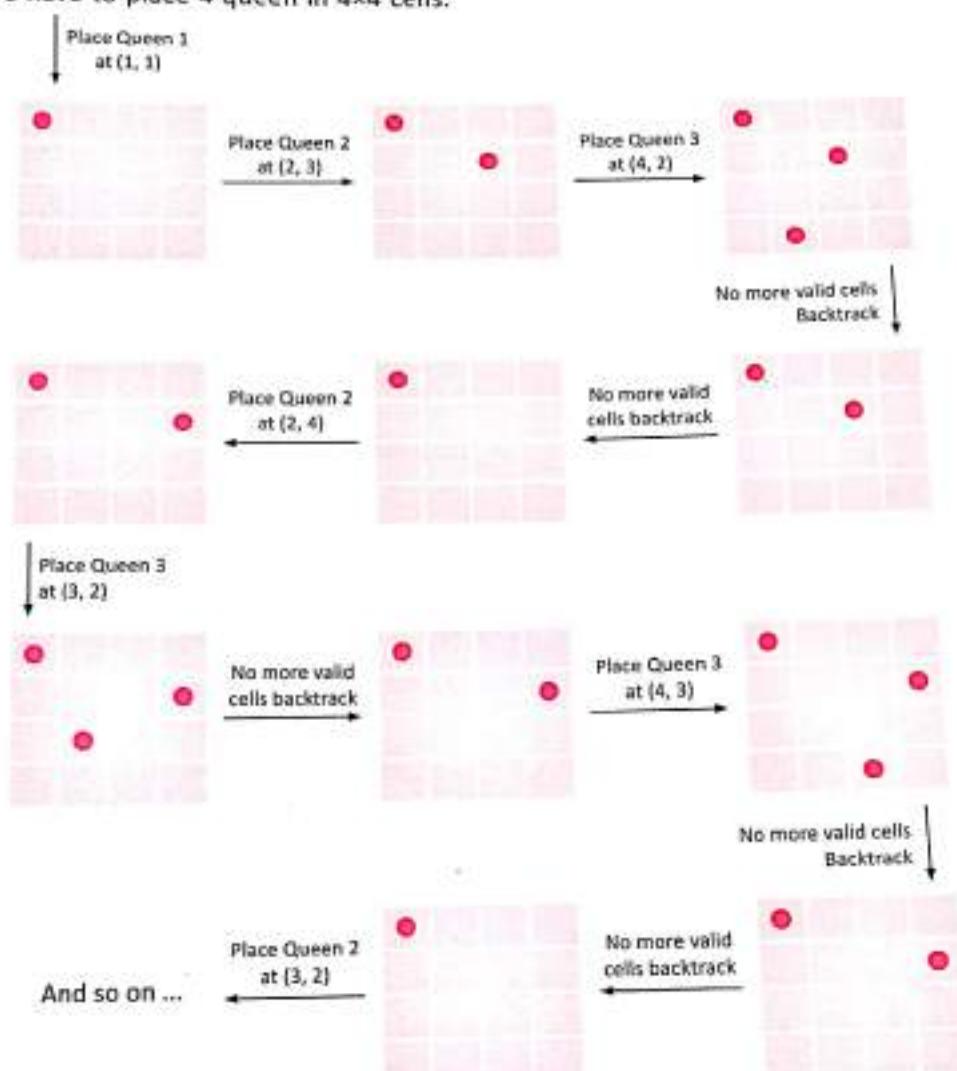
Approach towards the Solution:

We are having $N \times N$ unattacked cells where we need to place N -queens. Let's place the first queen at a cell (i, j) , so now the number of unattacked cells is reduced, and number of queens to be placed is $N - 1$. Place the next queen at some unattacked cell. This again reduces the number of unattacked cells and number of queens to be placed becomes $N - 2$. Continue doing this, as long as following conditions hold.

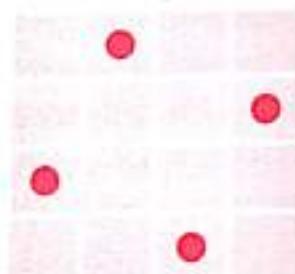
- The number of unattacked cells is not equal to 0.
- The number of queens to be placed is not equal to 0.

If the number of queens to be placed becomes 0, then it's over, we found a solution. But if the number of unattacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell.

Let's $N = 4$, We have to place 4 queen in 4×4 cells.



So, final solution of this problem is

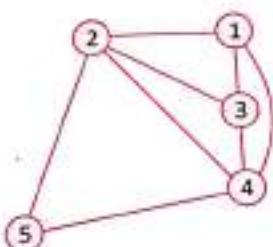


So, clearly, we tries solving a subproblem, if that does not result in the solution, it undo whatever changes were made and solve the next subproblem. If the solution does not exists (like $N=2$), then it returns false.

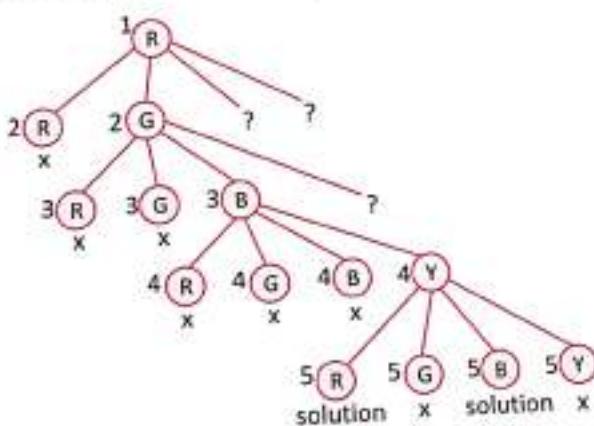
2. M Coloring Problems :

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

Suppose we have a graph G in the given figure and we need to color the vertex of this graph in such away that no two adjacent vertices have same color.



This graph coloring problem can be solved using the mechanism of Backtracking concept.
The state space tree for the above map is shown below :



Hence the above graph can be colored using four colors and four colors are Red, Green, Blue and Yellow.

3. Robots on Ice :

In this problem , we have given a grid size and a sequence of three check-in points. We need to find how many different tours are possible for to check-in all those points with equal time interval. Let's try to understand this problem with example.

Suppose the given grid size is 3×6 and that the three check-in points , in order to visitation are $(2,1)$, $(2,4)$ and $(0,4)$. Robot must start at $(0,0)$ and end at $(0,1)$. It must visit location $(2, 1)$ on step 4 ($= \#18/4\#$), location $(2, 4)$ on step 9 ($= \#18/2\#$), and location $(0, 4)$ on step 13 ($= \#3 \times 18/4\#$) and at the end it must visit all 18 squares.

Approach towards the solution :

This problem has been break into three main subproblems. In first subproblem we have to check all those path via we can reach to square (2, 1) starting from (0, 0) in 4 steps.

After this we have to check all those path via we can reach to square (2,4) starting from (2,1) in 5 (= 9-4) steps. Then we have to check all those paths via we can reach to square (0,4) starting from (2,4) in 4 (=13-9) steps. Now finally we need to rached at squares (0, 1) in 5 (=18 - 13) steps.

In above process, we have to remind that Robot must visit every square only once.

This is a backtracking problem where we have to check all possible paths to check-in all three given points.

In this given example , there are only two ways to do this.

2	3	4	7	8	9	10
1	2	5	6	15	14	11
0	1	18	17	16	13	12

2	3	4	5	6	9	10
1	2	17	16	7	8	11
0	1	18	15	14	13	12

4. Rat in a Maze :

Given a maze, NxN matrix. A rat has to find a path from source to des-ti-na-tion. maze[0][0] (left top corner)is the source and maze[N-1][N-1](right bot-ton cor-ner) is des-ti-na-tion. There are few cells which are blocked, means rat can-not enter into those cells. The rat can move only in two directions: forward and down.

Approach Towards the Solution :

Here we have to generate all paths from source to destination and one by one check if the generated path satisfies the constraints or not.

Let's the given maze is in the form of matrix whose entries are either 0 or 1. Entry 0 represent the block path from where the rat can't move. We have to print another matrix whose entries are either 0 or 1. In output matrix, entry 1 represents the the path of Rat from starting point to destination point.

Algorithm to generate all the Paths :

While there are untried paths

```

{
    Generate the next paths
    If this path has all blocks as 1
    {
        Print this path;
    }
}
  
```

Backtracking Algorithm :

```
If destination is reached
    Print the solution
```

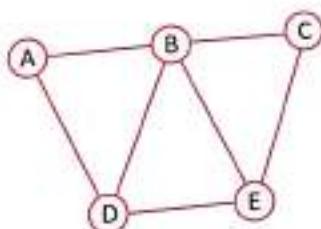
```
Else
```

- ```
{
 a) Mark current cell in solution matrix as 1.
 b) Move forward in horizontal direction and recursively check if this move leads to a solution.
 c) If the move chosen in the above step doesn't lead to a solution then move down and check if this move leads to a solution.
 d) If none of the above solutions work then unmark this cell as 0 (Backtrack) and return false.
}
```

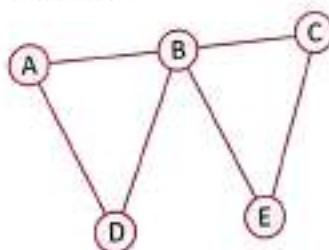
**1. Finding Hamiltonian Cycle :**

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not.

Example :



A Hamiltonian Cycle in the above graph is {A, B, C, E, D, A}. There are more Hamiltonian Cycles in the graph like {A, D, E, C, B, A}.



There is no any Hamiltonian Cycle in the above graph.

**Approach towards the solution :**

A naive algorithm is to generate all possible configurations of vertices and print a configuration that satisfies the given constraints. It will be  $n!$  ( $n$  factorial) configurations.

While there are untried configurations

```
{
```

## Recursion & Backtracking

Generate the next configuration

```
If (there are edges between two consecutive vertices of this configuration and there is an edge
from the last vertex to the first)
{
 Print this configuration;
 break;
}
}
}
```

### Backtracking Algorithm :

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.



### TIME TO TRY

- |                                  |                                  |                                        |
|----------------------------------|----------------------------------|----------------------------------------|
| 1. <a href="#">Knight's Tour</a> | 2. <a href="#">N-Queen</a>       | 3. <a href="#">Sudoku Solver</a>       |
| 4. <a href="#">Permutations</a>  | 5. <a href="#">Rat in a Maze</a> | 6. <a href="#">T-shirts (CodeChef)</a> |

Scanned by CamScanner

Rechtsanwälte - Rechtsanwälte

Rechtsanwälte - Rechtsanwälte



## 7

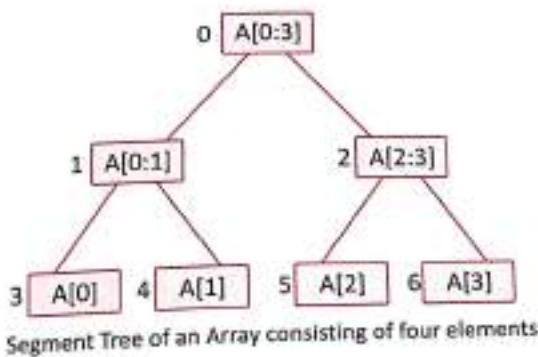
# Segment Tree

## Divide & Conquer

### What is a Segment Tree?

A Segment Tree is a full binary tree where each node represents an interval.

Generally a node would store one or more properties of an interval which can be queried later.



### Why do we need Segment Tree?

Many problem require that we give results based on query over a range or segment of available data. This can be a tedious and slow process, especially if the number of queries is large.

A segment tree let's us process such queries efficiently in logarithmic order of time.

### How do we make a Segment Tree?

Let the data be in array arr[]

1. The root of our tree will represent the entire interval of data we are interested in, i.e. arr[0...n-1].
2. Each leaf of the tree will represent a range consisting of just a single element. Thus the leaves represent arr[0], arr[1], ..., arr[n-1].
3. The internal nodes will represent the merged result of their child nodes.
4. Each of the children nodes could represent approximately half of the range represented by their parent.

Segment Tree generally contain three types of methods: BUILD, QUERY, UPDATE.

**Let's Start with an Example****Problem : Range Minimum Query**

You are given a list of  $N$  numbers and  $Q$  queries. There are two types of query:

1.  $0 \mid r$  - find the minimum element in range  $[l, r]$ .
2.  $1 \mid a$  - update the element at  $i$ th position of array to  $val$ .

The problem states that we have to find minimum element in a given range  $[i, j]$ , or update the element of the given array.

**Basic Approach**

For any range  $[i, j]$ , we can answer the minimum element in  $O(n)$ . But as we'll be having  $Q$  queries then we have to find minimum of a range,  $Q$  times.

So the overall complexity will be  $O(Q \times N)$ .

So, to solve range based problems and to bring the complexity to logarithmic time we use **Segment Tree**.

We'll use `tree[]` to store the nodes of our Segment Tree(initialized to all zeros).

- The root of the tree is at index 1, i.e, `tree[1]` is the root of our tree.
- The children of `tree[i]` are stored at `tree[i * 2]` and `tree[i * 2 + 1]`.

```
#include <iostream>
using namespace std;
int tree[400005]; //will store our segment tree structure
int arr[100005]; //input array
```

**BUILD Function :**

Here, Build Function takes three parameter, `node`, `a`, `b`.

```
void build_tree(int node, int a, int b)
{ //BUILD function
 // node represents the current node number
 // a, b represents the current node range
 // for leaf, node range will be single element
 // so 'a' will be equal to 'b' for leaf node
 if (a == b)
 { //checking if node is leaf
 //for single element, minimum will be the element itself
 tree[node] = arr[a]; //storing the answer in our tree structure
 return;
 }
}
```

```

int mid = (a + b) >> 1; //middle element of our range
build_tree(node * 2, a, mid); //recursively call for left half
build_tree(node * 2 + 1, mid + 1, b); //call for right half
//left child and right child are build
// now build the parent node using its child nodes
tree[node] = __min(tree[node*2], tree[node * 2 + 1]);
}

```

**QUERY Function**

```

int query_tree(int node, int a, int b, int i, int j)
{
// i, j represents the range to be queried
if (a > b || a > j || b < i) return INT_MAX; //out of range
if (a >= i && b <= j)
{ //current segment is totally within range[i,j]
return tree[node];
}
int mid = (a + b) >> 1;
//Query left child
int q1 = query_tree(node * 2, a, mid, i, j);
//Query right child
int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
return __min(q1, q2); // return final result
}

```

**UPDATE Function**

```

// updates the ith element to val
void update_tree(int node, int a, int b, int i, int val)
{
if (a > b || a > i || b < i) return; //out of range
if (a == b) { //leaf node
tree[node] = val;
return;
}
int mid = (a + b) >> 1;
update_tree(node * 2, a, mid, i, val); // updating left child
update_tree(node * 2 + 1, mid + 1, b, val); // updating right child
}

```

```
update_tree(node * 2 + 1, mid + 1, b, i, val); // updating right child
// updating node with min value
tree[node] = __min(tree[node * 2], tree[node * 2 + 1]);
}
```

### Complexity Analysis

#### BUILD

We visit each leaf of the Segment Tree. That makes  $n$  leaves. Also there will be  $n-1$  internal nodes. So we process about  $2 \cdot n$  nodes. This makes the Build process run in  $O(n)$  linear complexity.

#### UPDATE

The update process discards half of the range for every level of recursion to reach the appropriate leaf in the tree. This is similar to binary search and takes **logarithmic time**. After the leaf is updated, its direct ancestors at each level of the tree are updated. This takes time linear to height of the tree. So the complexity will be  $O(\lg(n))$ .

#### QUERY

The query process traverses depth-first through the tree looking for node(s) that match exactly with the queried range. At best, we query for the entire range and get our result from the root of the segment tree itself. At worst, we query for a interval/range of size 1 (which corresponds to a single element), and we end up traversing through the height of the tree. This takes time linear to height of the tree. So the complexity will be  $O(\lg(n))$ .

### Problem : Range Sum

We have an array  $\text{arr}[0, \dots, n-1]$  and we have to perform two types of queries:

1. Find sum of elements from index  $l$  to  $r$  where  $0 \leq l \leq r \leq n$
2. Increase the value of all elements from  $l$  to  $r$  by a given number.

As we can build the parent node using its two child nodes. So, we'll use the Segment Tree to answer the sum of elements from  $l$  to  $r$ .

Code :

```
int tree[400005]; //will store our segment tree structure
int arr[100005]; //input array
void build_tree(int node, int a, int b)
{ //BUILD function
if (a == b)
{ //checking if node is leaf
//for single element, sum will be the element itself
```

```

tree[node] = arr[a]; //storing the answer in our tree structure
return;
}

int mid = (a + b) >> 1; //middle element of our range
build_tree(node * 2, a, mid); //recursively call for left half
build_tree(node * 2 + 1, mid + 1, b); //call for right half
//left child and right child are build
// now build the parent node using its child nodes
tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

int query_tree(int node, int a, int b, int i, int j)
{
 // i, j represents the range to be queried
 if (a > b || a > j || b < i)
 return 0; //out of range
 if (a >= i && b <= j)
 { //current segment is totally within range[i,j]
 return tree[node];
 }
 int mid = (a + b) >> 1;
 int q1 = query_tree(node * 2, a, mid, i, j); //Query left child
 int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
 //Query right child
 return (q1 + q2); // return final result
}
void update_tree(int node, int a, int b, int i,int j, int val)
{
 if (a > b || a > j || b < i)
 return; //out of range
 if (a == b)
 { //leaf node
 tree[node] += val;
 return;
 }
 int mid = (a + b) >> 1;
}

```

```

update_tree(node * 2, a, mid, i, j, val); // updating left child
update_tree(node * 2 + 1, mid + 1, b, i, j, val); // updating right child
tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

```

The above approach will query the tree in  $O(\lg(n))$  time. But the problem will arise in the `update` function. As updating a single element takes  $O(\lg(n))$  time, now we are doing a range update. So, the overall complexity for update will be  $O(n\lg(n))$ , which will get TLE as we have to answer for multiple queries.

### **Lazy Propagation**

In the current version when we update a range, we branch its children even if the segment is covered within range. In the lazy version we only mark its child that it needs to be updated and update it when needed.

In short, we try to postpone updating descendants of a node, until the descendants themselves need to be accessed.

We use another array `lazy[]` which is the same size as our segment tree array `tree[]` to represent a lazy node. `lazy[i]` holds the amount by which the node `tree[i]` needs to be incremented, when that node is finally accessed or queried. When `lazy[i]` is zero, it means that node `tree[i]` is not lazy and has no pending updates.

### **UPDATE function**

```

void update_tree(int node, int a, int b, int i, int j, int val)
{
 if (lazy[node] != 0)
 { // lazy node
 tree[node] += (b - a + 1)*lazy[node]; //normalize current node by removing
 laziness
 if (a != b)
 { // update lazy[] for children nodes
 lazy[node * 2] += lazy[node];
 lazy[node * 2 + 1] += lazy[node];
 }
 lazy[node] = 0; // remove laziness from current node
 }
 if (a > b || a > j || b < i)
 return; //out of range
 if (a >= i && b <= j)
 { // segment is fully within update range

```

```

tree[node] += (b - a + 1)*val;
if (a != b)
{ // update lazy[] for children nodes
 lazy[node * 2] += lazy[node];
 lazy[node * 2 + 1] += lazy[node];
}
return;
}
int mid = (a + b) >> 1;
update_tree(node * 2, a, mid, i, j, val); // updating left child
update_tree(node * 2 + 1, mid + 1, b, i, j, val); // updating right child
tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

```

**QUERY function**

```

int query_tree(int node, int a, int b, int i, int j)
{
// i, j represents the range to be queried
if (a > b || a > j || b < i)
 return INT_MAX; //out of range
if (lazy[node] != 0)
{ // lazy node
 tree[node] += (b - a + 1)*lazy[node]; //normalize current node by removing
laziness
 if (a != b)
 { // update lazy[] for children nodes
 lazy[node * 2] += lazy[node];
 lazy[node * 2 + 1] += lazy[node];
 }
 lazy[node] = 0; // remove laziness from current node
}
if (a >= i && b <= j)
{ //current segment is totally within range[i,j]
 return tree[node];
}
int mid = (a + b) >> 1;
int q1 = query_tree(node * 2, a, mid, i, j); //Query left child
int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j); //Query right child
return q1 + q2;
}

```

```
int q2 = query_tree(node * 2 + 1, mid + 1, b, 1, j);
//Query right child
return (q1 + q2); // return final result
}
```

Using Lazy Propagation the range update will be done in  $O(\lg(n))$  time complexity.

### Note :

The following lines:

```
tree[node] += (b - a + 1)*lazy[node]; //normalize currentnode by removing laziness and
tree[node] += (b - a + 1)*val; // update segment and
tree[node] = tree[node * 2] + tree[node * 2 + 1]; //merge updates are specific to Range Sum Query
problem. Different problems may have different updating and merging schemes.
```

### Problem : GSS1 - Can you answer these queries

You are given a sequence  $A[1], A[2], \dots, A[N]$ . ( $|A[i]| \leq 15007, 1 \leq N \leq 50000$ ). A query is defined as follows:

$\text{Query}(x,y) = \text{Max} \{ a[i]+a[i+1]+\dots+a[j] ; x \leq i \leq j \leq y \}$ .

Given M queries, your program must output the results of these queries.

### INPUT :

```
3
-1 -2 -3
1
12
```

### OUTPUT :

```
2
http://www.spoj.com/problems/GSS1/
```

Now we need to use a Segment Tree. But what data to store in each node, such that it is easy to compute the data associated with a given node if we already know the data associated with its two child nodes.

We need to find a maximum sum subarray in a given range.



Say we have a as a parent node and p and q as its child nodes. Now we need to build data for a from and q such that node a can give the maximum sum subinterval for its range when queried.

So for this what do we need??

Maximum sum subarray in a can be equal to either:

1. Maximum sum subarray in p
2. Maximum sum subarray in q
3. Elements including from both p and q

So for each node we need to store:

- |                       |                                          |
|-----------------------|------------------------------------------|
| 1. Maximum Prefix Sum | 2. Maximum Suffix Sum                    |
| 3. Total Sum          | 4. Best Sum (Maximum Sum Subarray for a) |

Maximum Prefix Sum can be calculated for a node as

```
a.prefix = max(p.prefix,p.total + q.prefix)
```

Maximum Suffix Sum can be calculated for a node as

```
a.suffix = max(p.suffix,q.total + p.suffix)
```

Total Sum

```
a.total = p.total + q.total
```

Best Sum

```
a.best = max(p.suffix + q.prefix , max(p.best,q.best))
```

Code :

```
long long int arr[50005]; //input array
struct Tree
{
 long long int prefix, suffix, total, best;
};
Tree tree[200005];
void build_tree(long long int node, long long int a, long long int b)
{
 if (a == b)
 { //leaf node
 tree[node].prefix = arr[a]; // prefix sum
 tree[node].suffix = arr[a]; // suffix sum
 tree[node].total = arr[a]; // total sum
 tree[node].best = arr[a]; // best sum
 }
 return;
}
```

```

int mid = (a + b) >> 1;
build_tree(node * 2, a, mid);
build_tree(node * 2 + 1, mid + 1, b);
tree[node].prefix = max(tree[node * 2].prefix, tree[node*2].total + tree[node +
2 + 1].prefix); //prefix sum
tree[node].suffix = max(tree[node * 2 + 1].suffix, tree[node * 2].suffix +
tree[node * 2 + 1].total); //suffix sum
tree[node].total = tree[node * 2].total + tree[node *2 + 1].total; //total sum
tree[node].best = max(tree[node * 2].suffix + tree[node * 2 + 1].prefix,
max(tree[node * 2].best, tree[node *2 + 1].best)); //best sum
}

Tree query_tree(long long int node, long long int a, long long int b, long long int
i, long long int j)
{
Tree t;
if (a > b || a > j || b < i)
{
t.prefix = t.suffix = t.best = INT_MIN ;
t.total = 0;
return t;
}
if (a >= i && b <= j)
{
return tree[node];
}
long long int mid = (a + b) >> 1;
Tree q1 = query_tree(node * 2, a, mid, i, j);
Tree q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
t.prefix = max(q1.prefix, q1.total + q2.prefix);
t.suffix = max(q2.suffix, q1.suffix + q2.total);
t.total = q1.total + q2.total;
t.best = max(q1.suffix + q2.prefix, max(q1.best, q2.best));
return t;
}

```

Similar Problem: <http://www.spoj.com/problems/GSS3/>

The problem also includes an update operation, rest of the methods are same.

**Problem : QSET (Queries on the String)**

You have a string of N decimal digits, denoted by numbers A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>N</sub>.

Now you are given M queries, each of whom is of following two types.

1. X Y: Replace A<sub>X</sub> by Y.
2. C D: Print the number of sub-strings divisible by 3 in range [C,D]

Formally, you have to print the number of pairs (i,j) such that the string A<sub>i</sub>, A<sub>i+1</sub>...A<sub>j</sub>, (C ≤ i ≤ j ≤ D), when considered as a decimal number, is divisible by 3.

**INPUT :**

```
53
01245
213
145
235
```

**OUTPUT :**

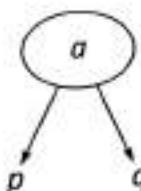
```
3
1
```

<https://www.codechef.com/problems/QSET>

A number is divisible by 3 if sum of its digits are divisible by 3.

Use segment trees. Store in node for each interval,

- the answer for that interval
- prefix[], where prefix[i] denotes number of prefixes of interval which modulo 3 give i.
- suffix[], where suffix[i] denotes number of suffixes of interval which modulo 3 give i.
- total sum of interval modulo 3.



Answer for interval denoted by a will be:

a.ans = p.ans + q.ans.

(Number of Suffixes in p which when taken with modulo 3 gives 1) + (Number of Prefixes in q which when taken with modulo 3 gives 2) → also gives the substring divisible by 3 in a.

Similarly, for all possible combinations we can calculate the answer for an interval.

```
a.ans = p.ans + q.ans;
for (int i = 0; i < 3; ++i)
{
```

```

for (int j = 0; j < 3; ++j)
{
 if ((i + j) % 3 == 0)
 {
 a.ans += p.suffix[i] * q.prefix[j];
 }
}
}

```

### How to build prefix and suffix array for a node?

There are  $p.prefix[i]$  prefixes of  $p$  which when taken modulo with 3 gives  $i$ , and there are some prefixes which are made by combining whole of  $p$  and some prefixes of  $q$ .

So, total prefixes in  $a$  which when taken modulo with 3 gives  $i$  will be:  
(Same for suffix)

```

for (int i = 0; i < 3; ++i)
{
 a.prefix[i] = p.prefix[i] + q.prefix[(3 - q1.total + i) % 3];
 a.suffix[i] = q.suffix[i] + p.suffix[(3 - q2.total + i) % 3];
}

```

Code :

```

int arr[100005]; //input array
struct Tree
{
 int ans;
 int prefix[3], suffix[3];
 int total;
};

Tree tree[400005];
void build_tree(int node, int a, int b)
{
 if (a == b)
 {
 tree[node].prefix[arr[a] % 3] = 1;
 tree[node].suffix[arr[a] % 3] = 1;
 tree[node].total = arr[a] % 3;
 }
}

```

```

tree[node].ans = (arr[a] % 3 == 0 ? 1 : 0);
return;
}
int mid = (a + b) >> 1;
build_tree(node * 2, a, mid);
build_tree(node * 2 + 1, mid + 1, b);
for (int i = 0; i < 3; ++i)
{
 tree[node].prefix[i] = tree[node * 2].prefix[i] +
 tree[node * 2 + 1].prefix[(3 - tree[node * 2].total + i)% 3];
 tree[node].suffix[i] = tree[node * 2 + 1].suffix[i] + tree[node * 2].suffix[(3 -
 tree[node * 2 + 1].total+ i) % 3];
}
tree[node].total = (tree[node * 2].total + tree[node* 2 + 1].total) % 3;
tree[node].ans = tree[node * 2].ans + tree[node * 2 +1].ans;
for (int i = 0; i < 3; ++i)

```

### PROBLEM : JTREE (JosephLand)

Nick lives in a country named JosephLand. JosephLand consists of N cities. City 1 is the capital city. There are  $N - 1$  directed roads. It's guaranteed that it is possible to reach capital city from any city, and in fact there is a unique path from any city to the capital city.

Besides, you can't cross roads for free. To pass a road, you must have a ticket. There are total M tickets. You can not have more than one ticket at a time! Each ticket is represented by three integers:  $v\ k\ w$ : you can buy a ticket with cost w in the city v. This ticket can be used at max k times. That means, after using this ticket for k roads ticket can't be used!

By the way, you can tear your ticket any time and buy a new one. But you are not allowed to buy a ticket if you are still having a ticket with you!

Nick's home is located in the capital city. He has Q friends, and he wants to invite all of them for dinner! So he is interested in knowing about how much each of his friends is going to spend in the journey!

His friends are quite smart and always choose a route to capital city that minimizes his/her spending! Nick has to prepare dinner, so he doesn't have time to figure out himself, Can you please help him? Please note that it's guaranteed that, one can reach the capital from any city using the tickets!

**INPUT :**

77  
31  
21  
76  
63  
53  
43  
723  
711  
235  
362  
424  
5310  
6120  
3  
5  
6  
7

**OUTPUT :**

10  
22  
5

<https://www.codechef.com/problems/JTREE>

The problem in simple words is:

Given a tree with edges directed towards root and nodes having some ticket information which allows you to travel k units towards the root with cost w. Answer Q queries i.e output the minimum cost for travelling from a node x to root.

Let's try to solve problem for a node X at depth H from root 1. Think of this path as a 1D vector V where we have all the H-1 nodes between root and X. The nodes are stored in vector in increasing order of depth.

Let  $dp[x]$  be the minimum cost to travel from X to the root.

So, for a given node, iterate over all the tickets present at node X and DP state will be like this:

```
for(int i=0;i<total_tickets;i++)
{
 K=current_ticket_jump_info;
 W=weight_ticket_info;
 for(int j=V.size()-1;j>= max(0,V.size()-1-k);j++) //loop 2
```

```

 {
 DP[X]=min(DP[X] , DP[V[j]] + W);
 }
}

```

We will use DFS to find vector V for every node X.

Now the code will be :

```

void dfs(int u,int p)
{
 V.push_back(u);
 for(int l=0 ; l< adj[u].size() ; l++)
 {
 if(adj[u][l]==p)
 continue;
 int x =adj[u][l];
 for(int i=0;i<total_tickets;i++) // loop 1
 {
 K=current_ticket_jump_info;
 W=weight_ticket_info;
 for(int j=V.size()-1;j>= max(0,V.size()-1-k);j++) //loop 2
 {
 DP[x]=min(DP[x] , DP[V[j]] + W);
 }
 }
 dfs(adj[u][l],u);
 }
 V.pop_back();
}

```

The complexity will be  $O(n + m * n)$ .

But If we look at the inner loop of the code which goes up to K times and we find the minimum of DP, this can be done using any data structure that supports RMQ(Range Minimum Query) + Point updates in  $O(\log n)$  time . Which will make the total complexity to be  $O(N + M \log N)$ .

So we can build a segment tree that supports two operations.

First : find the minimum element in range L,R and

## Segment Tree

Second: update an element's value to VAL.

And we need to query for the minimum element between  $[H-K, H]$  so we will build the tree over height H.

Now just print DP[ x ] for every query.

Code :

```
void dfs(long long int cur, long long int h)
{
 for (int j = 0; j < n_info[cur].size(); ++j)
 { //no. of tickets
 long long int k1 = n_info[cur][j].first;
 long long int w1 = n_info[cur][j].second;
 dp[cur] = min(dp[cur], query_tree(1, 0, N - 1, max(0LL, h - k1), h) + w1); //find the min cost from cur to root
 }
 update_tree(1, 0, N - 1, h, dp[cur]); //update the tree at posn h with its DP value
 for(int i = 0; i < g_tree[cur].size(); ++i)
 {
 long long int x = g_tree[cur][i];
 dfs(x, h + 1);
 }
 update_tree(1, 0, N - 1, h, INF_n); //update the tree at posn h with INF as we are done with its subtree
}
int main()
{
 for(int i = 0; i < 400003; ++i)
 {
 tree[i] = INF_n;
 dp[i / 4] = INF_n;
 }
 cin >> N >> M;
 for(int i = 0; i < N - 1; ++i)
 {
 cin >> a >> b;
 g_tree[b].push_back(a);
 }
 for(int i = 0; i < M; ++i)
```

```

{
 cin >> v >> k >> w;
 n_info[v].push_back(make_pair(k, w));
}
dp[1] = 0;
update_tree(1, 0, N - 1, 0, dp[1]);
dfs(1, 0);
cin >> Q;
while (Q--)
{
 cin >> a;
 cout << dp[a] << "\n";
}
return 0;
}

```

**PROBLEM : COUNZ (Counting Zeroes)**

You are given an array of N integers(Indexed at 1)

For the given array you have to answer some queries given later. The queries are of 2 types:

**1. TYPE 1 -> 1 L R (where L and R are integers)**

For this query you have to calculate the product of elements of the array in the range L to R (both inclusive) and print the number of zeros at the end of the result.

**2. TYPE 2 -> 0 L R V (where L,R,V are integers)**

For this query you have to set the value of all the elements in the array ranging from L to R (both inclusive) to V.

**INPUT :**

5  
13589  
3  
125  
01410  
115

**OUTPUT :**

1  
4  
<https://www.codechef.com/problems/COUNZ>

We'll use Segment Tree to answer the queries.

## Segment Tree

To determine the number of zeroes at the end of a number, we should know number of 2's and 5's in its prime factorization.

**Number Of Zeroes = min(No. of 2's, No. of 5's)**

**Number of Zeroes in product = min(sum of 2's in elements in product, sum of 5's in elements in product).**

**Be careful of element having value 0 in product as number of zeroes will be 1.\***

Since any array element can have value 0 at any point of time we store 3 values at a node in segment tree sum of number of twos in prime factorization of all the elements in range, sum of number of twos in prime factorization of all the elements in range and number of elements in range with value 0.

Now, using lazy propagation we can answer queries in  $O(\log_2(\text{MaxA}[i]) + \log N)$  time.

**Total Complexity :  $O(N\log_2(\text{Max A}[i]) + Q(\log_2(\text{Max A}[i]) + \log N))$**

**Code :**

```
void process_node(int node, int num)
{
 tree[node][0] = tree[node][1] = tree[node][2] = 0;
 if (num == 0)
 tree[node][0]++;
 while (num % 2 == 0 && num != 0)
 {
 num /= 2;
 tree[node][1]++;
 }
 while (num % 5 == 0 && num != 0)
 {
 num /= 5;
 tree[node][2]++;
 }
 return;
}
void build_tree(int node, int a, int b)
{
 if (a == b)
 {
 int num = arr[a];
 tree[node][0] = tree[node][1] = tree[node][2] = 0;
 if (num == 0)
 tree[node][0]++;
 while (num % 2 == 0 && num != 0)
 {
 num /= 2;
 tree[node][1]++;
 }
 while (num % 5 == 0 && num != 0)
 {
 num /= 5;
 tree[node][2]++;
 }
 }
}
```

```

process_node(node, num);
return;
}
int mid = (a + b) >> 1;
build_tree(node * 2, a, mid);
build_tree(node * 2 + 1, mid + 1, b);
for (int i = 0; i < 3; ++i) //build the parent node
tree[node][i] = tree[node * 2][i] + tree[node * 2 + 1][i];
}
void update_tree(int node, int a, int b, int i, int j, int val)
{
if (lazy[node] != -1)
{ //if lazy
process_node(node, lazy[node]);
for (int i = 0; i < 3; ++i)
tree[node][i] *= (b - a + 1);
if (a != b)
{ //not a leaf node
lazy[node * 2] = lazy[node * 2 + 1] = lazy[node]; //mark lazy
}
lazy[node] = -1; //unmark lazy
}
if (a > b || a > j || b < i) return;
if (a >= i && b <= j)
{
process_node(node, val);
for (int i = 0; i < 3; ++i)
tree[node][i] *= (b - a + 1);
if (a != b)
{
lazy[node * 2] = lazy[node * 2 + 1] = val;
}
return;
}
int mid = (a + b) >> 1;

```

```

update_tree(node * 2, a, mid, i, j, val);
update_tree(node * 2 + 1, mid + 1, b, i, j, val);
for (int i = 0; i < 3; ++i)
 tree[node][i] = tree[node * 2][i] + tree[node * 2 + 1][i];
}

void query_tree(int node, int a, int b, int i, int j)
{
if (lazy[node] != -1)
{ //if lazy
process_node(node, lazy[node]);
for (int i = 0; i < 3; ++i)
 tree[node][i] *= (b - a + 1); //multiply value by range times
if (a != b)
{ //if not leaf
lazy[node * 2] = lazy[node * 2 + 1] = lazy[node];//mark as lazy
}
lazy[node] = -1; //unmark lazy
}
if (a > b || a > j || b < i)
{
q_tree[node][0] = q_tree[node][1] = q_tree[node][2] = 0;
return;
}
if (a >= i && b <= j)
{
for (int i = 0; i < 3; ++i)
 q_tree[node][i] = tree[node][i];
return;
}
int mid = (a + b) >> 1;
query_tree(node * 2, a, mid, i, j);
query_tree(node * 2 + 1, mid + 1, b, i, j);
for (int i = 0; i < 3; ++i)
q_tree[node][i] = q_tree[node * 2][i] + q_tree[node * 2 + 1][i];
}

```

**PROBLEM : DIVMAC (Dividing Machine)**

Chef has created a special dividing machine that supports the below given operations on an array of positive integers.

There are two operations that Chef implemented on the machine.

**Type 0 Operation**

**Update(L,R):**

```
for i = L to R:
```

```
 a[i] = a[i] / LeastPrimeDivisor(a[i])
```

**Type 1 Operation**

**Get(L,R):**

```
result = 1
```

```
for i = L to R:
```

```
 result = max(result, LeastPrimeDivisor(a[i]))
```

```
return result;
```

The function **LeastPrimeDivisor(x)** finds the smallest prime divisor of a number. If the number does not have any prime divisors, then it returns 1.

Chef has provided you an array of size N, on which you have to apply M operations using the special machine. Each operation will be one of the above given two types. Your task is to implement the special dividing machine operations designed by Chef. Chef finds this task quite easy using his machine, do you too?

**INPUT :**

```
2
67
25810344
126
023
126
046
116
016
146
22
13
022
112
```

**OUTPUT :**

5 3 5 11

1

The problem is :

Given an array of numbers A, support the following two queries:

- for a given range [L, R], reduce each number in A[L...R] by its smallest prime factor
- for a given range [L, R], find the number in A[L...R] with largest smallest prime factor.

When we get the range modification entry, we modify each element in the range one by one, and hence perform  $(R - L + 1)$  single element update queries. This approach would be too slow, as it makes the complexity of range update query to  $O((R - L + 1) \lg N)$ .

Note that, if the value of an element  $A[x]$  is 1, then update query on  $A[x]$  does not have any effect on the segment tree. We call such queries as degenerate queries, and can discard them. Also note that a number M can be written as a product of at most  $O(\lg M)$  prime numbers. Hence, we can perform at most  $O(\lg M)$  non-degenerate modification queries on  $A[x] = M$  all the latter queries will be degenerate. This means that if all numbers in the array are smaller than M, then at most  $O(N \lg M)$  non-degenerate single element update queries would be performed. Each single element update query takes  $O(\lg N)$  time, and hence all update queries can be performed in  $O(N \lg M \lg N)$  time.

**Code :**

```

void modified_sieve()
{
 //linear time sieve
 lp[1] = 1;
 for (int i = 2; i < maxn; ++i)
 {
 if (lp[i] == 0)
 {
 lp[i] = i;
 prime.push_back(i);
 }
 for (int j = 0; j < prime.size() && i*prime[j] < maxn; ++j)
 {
 lp[i*prime[j]] = prime[j];
 }
 }
}

void build_tree(int node, int a, int b)
{
 if (a == b)
 {

```

```

tree[node] = lp[arr[a]];
ones[node] = (arr[a] == 1 ? 1 : 0);
return;
}
int mid = (a + b) >> 1;
build_tree(node * 2, a, mid);
build_tree(node * 2 + 1, mid + 1, b);
ones[node] = ones[node * 2] + ones[node * 2 + 1];
tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
}
void update_tree(int node, int a, int b, int i, int j)
{
if (a > b || a > j || b < i)
 return;
if (a >= i && b <= j)
{
if (ones[node] == (b - a + 1))
{
 return;
}
if (a == b)
{ //leaf node
 arr[a] /= lp[arr[a]];
 tree[node] = lp[arr[a]];
 ones[node] = (arr[a] == 1 ? 1 : 0);
 return;
}
int mid = (a + b) >> 1;
update_tree(node * 2, a, mid, i, j);
update_tree(node * 2 + 1, mid + 1, b, i, j);
ones[node] = ones[node * 2] + ones[node * 2 + 1];
tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
}
int query_tree(int node, int a, int b, int i, int j)
{
if (a > b || a > j || b < i)
{
 return INT_MIN;
}
}

```

## Segment Tree

```
if (a >= i && b <= j)
{
 return tree[node];
}
int mid = (a + b) >> 1;
int q1 = query_tree(node * 2, a, mid, i, j);
int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
return q1 > q2 ? q1 : q2;
}
```





## 8

# Binary Index Tree

Binary Indexed Tree also called Fenwick Tree provides a way to represent an array of numbers in an array, allowing prefix sums to be calculated efficiently.

Calculating prefix sums efficiently is useful in various scenarios. Let's start with a simple problem:

We are given an array  $a[]$ , and we want to be able to perform two types of operations on it.

1. Change the value stored at an index  $i$ . (This is called a point update operation)
2. Find the sum of a prefix of length  $k$ . (This is called a range sum query)

A straightforward implementation of the above would look like this.

```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v) //assigns value v to a[i]
{
 a[i] = v;
}
int prefixsum(int k) //calculate the sum of all a[i] s
uch that 0 <= i < k
{
 int sum = 0;
 for(int i = 0; i < k; i++)
 sum += a[i];
 return sum;
}
```

This is a perfect solution, but unfortunately the time required to calculate a prefix sum is proportional to the length of the array, so this will usually time out when large number of such operations are performed.

## Can We Do Better Than This?

One efficient solution is to use segment tree that can perform both operation in  $O(\log N)$  time.

Using **Binary Indexed Tree** also, we can perform both the tasks in  $O(\log N)$  time. But then why learn another data structure when segment tree can do the work for us. It's because binary indexed trees require less space and are very easy to implement during programming contests (the total code is not more than 8-10 lines).

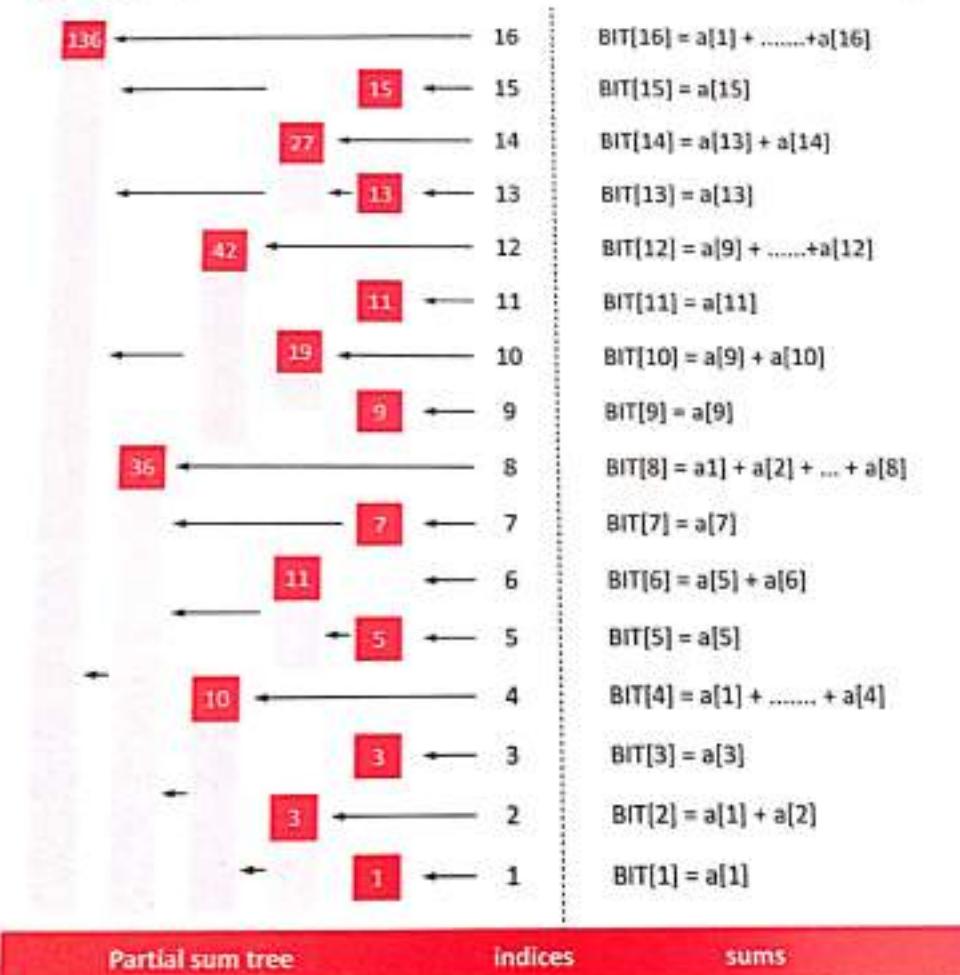
**Basic Idea of Binary Indexed Tree :**

We know the fact that each integer can be represented as sum of powers of two. Similarly, for a given array of size  $N$ , we can maintain an array  $\text{BIT}[]$  such that, at any index we can store sum of some numbers of the given array. This can also be called a partial sum tree.

Let's use an example to understand how  $\text{BIT}[]$  stores partial sums.

//for ease, we make sure our given array is 1-based indexed

```
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
```



The value in the enclosed box represents  $\text{BIT}[\text{index}]$

The above picture shows the binary indexed tree, each enclosed box of which denotes the value  $\text{BIT}[\text{index}]$  and each  $\text{BIT}[\text{index}]$  stores partial sum of some numbers.

Every index  $i$  in the  $\text{BIT}[]$  array stores the cumulative sum from the index  $i$  to  $i - (1 \ll r) + 1$  (both inclusive), where  $r$  represents the last set bit in the index  $i$ .

**Sum of first 12 numbers in array a[] = BIT[12] + BIT[8] = (a[12] + ... + a[9]) + (a[8] + ... + a[1])**

Similarly, **sum of first 6 elements = BIT[6] + BIT[4] = (a[6] + a[5]) + (a[4] + ... + a[1])**

**Sum of first 8 elements = BIT[8] = a[8] + ... + a[1]**

### Construct The BIT :

Let's see how to construct this tree and then we will come back to querying the tree for prefix sums.

BIT[] is an array of size = 1 + the size of the given array a[] on which we need to perform operations. Initially all values in BIT[] are equal to 0. Then we call update() operation for each element of given array to construct the Binary Indexed Tree. The update() operation is discussed below.

```
void update(int index,int val) {
 while (index <= maxn) {
 tree[index] += val;
 index += (index & (-index));
 }
}
```

Suppose we call **update(13, 2)**.

Here we see from the above figure that indices 13, 14, 16 cover index 13 and thus we need to add 2 to them also.

Initially x is 13, we update BIT[13]

BIT[13] += 2;

Now isolate the last set bit of x = 13(1101) and add that to x , i.e. **x += x&(-x)**

Last bit is of x = 13(1101) is 1 which we add to x, then  $x = 13+1 = 14$ , we update BIT[14]

BIT[14] += 2;

Now 14 is 1110, isolate last bit and add to 14, x becomes  $14+2 = 16$ (10000), we update BIT[16]

BIT[16] += 2;

In this way, when an update() operation is performed on index x we update all the indices of BIT[] which cover index x and maintain the BIT[].

If we look at the for loop in update() operation, we can see that the loop runs at most the number of bits in index x which is restricted to be less or equal to n (the size of the given array), so we can say that the update operation takes at most **O(log<sub>2</sub>(n))** time.

### Query in BIT :

```
int query(int index) {
 int sum = 0;
 while (index > 0) {
 sum += tree[index];
 index -= (index & (-index));
 }
 return sum;
}
```

The above function `query()` returns the sum of first  $x$  elements in given array. Let's see how it works.

Suppose we call `query(14)`, initially  $sum = 0$

$x$  is 14(1110) we add  $BIT[14]$  to our  $sum$  variable, thus  $sum = BIT[14] = \{a[14] + a[13]\}$

Now we isolate the last set bit from  $x = 14(1110)$  and subtract it from  $x$  last set bit in 14(1110) is 2(10), thus

$$x = 14 - 2 = 12$$

We add  $BIT[12]$  to our  $sum$  variable, thus  $sum = BIT[14] + BIT[12] = \{a[14] + a[13]\} + \{a[12] + \dots + a[9]\}$

again we isolate last set bit from  $x = 12(1100)$  and subtract it from  $x$  last set bit in 12(1100) is 4(100), thus  
 $x = 12 - 4 = 8$  we add  $BIT[8]$  to our  $sum$  variable, thus

$sum = BIT[14] + BIT[12] + BIT[8] = \{a[14] + a[13]\} + \{a[12] + \dots + a[9]\} + \{a[8] + \dots + a[1]\}$

Once again we isolate last set bit from  $x = 8(1000)$  and subtract it from  $x$  last set bit in 8(1000) is 8(1000), thus  $x = 8 - 8 = 0$  since  $x = 0$ , the for loop breaks and we return the prefix sum.

Talking about complexity, again we can see that the loop iterates at most the number of bits in  $x$  which will be at most  $n$ (the size of the given array). Thus the `query` operation takes  $O(\log_2(n))$  time.

### When To use BIT ?

Before going for Binary Indexed tree to perform operations over range, one must confirm that the operation or the function is:

**Associative.** i.e  $f(f(a, b), c) = f(a, f(b, c))$  this is true even for seg-tree

**Has an inverse.** eg:

- Addition has inverse subtraction (this example we have discussed)
- Multiplication has inverse division
- $\gcd()$  has no inverse, so we can't use BIT to calculate range gcd's

### PROBLEM : INVCNT (Inversion Count)

Let  $A[0...n - 1]$  be an array of  $n$  distinct positive integers. If  $i < j$  and  $A[i] > A[j]$  then the pair  $\{i, j\}$  is called an **inversion of A**. Given  $n$  and an array  $A$  your task is to find the number of inversions of  $A$ .

For example, the array `a={2,3,1,5,4}` has three inversions: (1,3), (2,3), (4,5), for the pairs of entries (2,1), (3,1), (5,4).

We'll use BIT to solve the problem of Inversion Count.

#### Replacing the Values of the Array with the Indexes :

Usually when we are implementing a BIT it is necessary to map the original values of the array to a new range with values between [1,N], where N is the size of the array. This is due to the following reasons:

- The values in one or more `A[i]` entry are too high or too low.

(e.g.  $10 - 12$  or  $10^4 - 12$ ).

For example imagine that we are given an array of 3 integers: {1,  $10^{12}$ , 5}

This means that if we want to construct a frequency table for our BIT data structure, we are going to need at least an array of  $10^{12}$  elements.

- The values in one or more `A[i]` entry are negative.

Because we are using arrays it's not possible to handle in our BIT frequency of negative values (e.g. we are not able to do `freq[-12]`).

A simple way to deal with this issues is to replace the original values of the target array for indexes that maintain its relative order.

For example, given the following array:

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| A | 9 | 1 | 0 | 5 | 4 |
|---|---|---|---|---|---|

The first step is to make a copy of the original array A let's call it B.

Then we proceed to sort B in non-descending order as follow:

|   |                            |  |  |  |  |
|---|----------------------------|--|--|--|--|
|   | sorted copy of the array A |  |  |  |  |
| B | 0 1 4 5 9                  |  |  |  |  |

Using binary search over the array B we are going to seek for every element in the array A, and stored the resulting position indexes (1-based) in the new array A.

`binary_search(B,9)=4` found at position 4 of the array B

`binary_search(B,1)=1` found at position 1 of the array B

`binary_search(B,0)=0` found at position 0 of the array B

`binary_search(B,5)=3` found at position 3 of the array B

`binary_search(B,4)=2` found at position 2 of the array B

The resulting array after increment each position by one is the following:

|   |             |   |   |   |   |
|---|-------------|---|---|---|---|
|   | new array A |   |   |   |   |
| A | 5           | 2 | 1 | 4 | 3 |

## Binary Indexed Tree

The following C++ code fragment illustrate the ideas previously explained:

```
for(int i = 0; i < N; ++i)
 B[i] = A[i]; // copy the content of array A to array B

sort(B, B + N); // sort array B

for(int i = 0; i < N; ++i) {
 int ind = int(lower_bound(B, B + N, A[i]) - B);
 A[i] = ind + 1;
}
```

### Counting inversions with the accumulate frequency :

Initially the cumulative frequency table is empty, we start the process with the element 3, the last one in our array.

|                            |   |   |   |   |   |
|----------------------------|---|---|---|---|---|
| A                          | 5 | 2 | 1 | 4 | 3 |
| cumulative frequency array |   |   |   |   |   |
|                            | 0 | 0 | 0 | 0 |   |

how many numbers less than 3 have we seen so far

x=read(3"1)=0

inv\_counter=inv\_counter+x

update the count of 3's so far

update(3,+1)

inv\_counter=0

|                            |   |   |   |   |   |
|----------------------------|---|---|---|---|---|
| A                          | 5 | 2 | 1 | 4 | 3 |
| cumulative frequency array |   |   |   |   |   |
|                            | 0 | 0 | 1 | 1 | 1 |

The cumulative frequency of value 3 was increased in the previous step, this is why the read(4"1) count the inversion (4,3).

how many numbers less than 4 have we seen so far

x=read(4"1)=1

inv\_counter=inv\_counter+x

update the count of 4's so far

```
update(4,+1)
inv_counter=1
```

The term 1 is the lowest in our array, this is why there is no inversions beginning at 1.

|                            |   |   |   |   |   |
|----------------------------|---|---|---|---|---|
| A                          | 5 | 2 | 1 | 4 | 3 |
| cumulative frequency array |   |   |   |   |   |
|                            | 0 | 0 | 1 | 2 | 2 |

how many numbers less than 1 have we seen so far

```
x=read(1"1)=0
```

```
inv_counter=inv_counter+x
```

update the count of 1's so far

```
update(1,+1)
```

```
inv_counter=1
```

And so on for the other elements...

### PROBLEM : BAADSHAH

You are provided an array consisting of 'n' integers and asked to do following operations:

1. Update the position at "x" of array to "y" ie  $A[x]=y$ .
2. Find if there is any prefix in the array whose sum equals to "S". If yes, then output "Found" and the last index of prefix, else output "Not Found".

#### INPUT :

```
44
1234
26
14 10
216
25
```

#### OUTPUT :

Found 3

Found 4

Not Found

<https://www.codechef.com/problems/BAADSHAH>

Construct a BIT(Binary Indexed Tree) from the given array, where internal nodes represents range sum.

Use Point updates for query 1, and for query 2, using Binary search on range-query sums, where starting range is 1 to n, check if the prefix sum exists or not.

Code :

```
ll read(ll index) {
 ll sum = 0;
 while (index > 0) {
 sum += BIT[index];
 index -= (index & (-index));
 }
 return sum;
}

void update(ll index, ll val) {
 while (index < maxn) {
 BIT[index] += val;
 index += (index & (-index));
 }
}

int main() {
 cin >> N >> M;
 for (int i = 1; i <= N; ++i) {
 cin >> arr[i];
 update(i, arr[i]);
 }

 while (M--) {
 ll idx;
 cin >> type;
 if (type == 1) {
 cin >> a >> b;
 update(a, b - arr[a]);
 arr[a] = b;
 }
 else {
 }
```

```

bool flag = false;
cin >> a;
ll lo = 1, hi = N;
while (lo <= hi) {
 int mid = (lo + hi) >> 1;
 ll ans = read(mid);
 if (ans < a) {
 lo = mid + 1;
 }
 else if (ans > a) {
 hi = mid - 1;
 }
 else {
 flag = true;
 idx = mid;
 break;
 }
}
if (flag) cout << "Found " << idx << "\n";
else cout << "Not Found\n";
}
return 0;
}

```

### PROBLEM : CTRICK (CARD TRICK)

The magician shuffles a small pack of cards, holds it face down and performs the following procedure:

The top card is moved to the bottom of the pack. The new top card is dealt face up onto the table.  
It is the Ace of Spades.

Two cards are moved one at a time from the top to the bottom. The next card is dealt face up onto the table. It is the Two of Spades.

Three cards are moved one at a time...

This goes on until the nth and last card turns out to be the n of Spades.

## Binary Indexed Tree

This impressive trick works if the magician knows how to arrange the cards beforehand (and knows how to give a false shuffle). Your program has to determine the initial order of the cards for a given number of cards,  $1 \leq n \leq 20000$ .

**INPUT :**

2

4

5

**OUTPUT :**

2 1 4 3

3 1 4 5 2

We are asked to place 1 in 2nd free cell of our answer, then to place 2 in 3rd free cell of our answer while starting counting from position where we had placed 1 (and starting from the beginning if we reached end of array), then to place 3 in 4th free cell, and so on.

Now add one more optimization. At every moment we know how many free cells are in our array now. Assume we are at position X now, need to move 20 free cells forward, and you know that at given moment

there are 2 free cells before X and 4 free cells after X. It means that we need to find 22nd free cell starting from beginning of array, which is same as going full circle 3 times and then taking 4th free cell. As we can see, we moved to a problem "find i-th zero in array".

**Code :**

```
int read(int ind) {
 int sum = 0;
 while (ind > 0) {
 sum += tree[ind];
 ind -= (ind & -ind);
 }
 return sum;
}

void update(int ind, int val) {
 while (ind < 20002) {
 tree[ind] += val;
 ind += (ind & -ind);
 }
}
```

```

int ans[20005];
int b_search(int val) {

 int l = 1, r = n;
 while (l < r) {

 int mid = (l + r) >> 1;
 int tt = read(mid);

 if (tt > val) {
 r = mid;
 }
 else if (tt < val) {
 l = mid + 1;
 }
 else {
 r = mid;
 if (ans[mid] == 0) return mid;
 }
 }
 return l;
}

int main() {

 ios_base::sync_with_stdio(false);
 cin.tie(NULL);

 cin >> t;
 while (t--) {

 cin >> n;
 memset(tree, 0, sizeof(tree));
 for (int i = 1; i <= n; i++) {
 update(i, 1);
 }
 }
}

```

```

int fcells = n;
int cur = 1;
for (int i = 1; i < n + 1; ++i) {
 int free = i + 1;
 int freeb = read(cur - 1);
 int freef = read(n) - freeb;
 int yy = free + freeb;
 int cell = yy % fcells;
 if (cell == 0) cell = fcells;
 int ret_ind = b_search(cell);
 ans[ret_ind] = i;
 fcells--;
 update(ret_ind, -1);
 cur = ret_ind;
}

for (int i = 1; i < n + 1; ++i) {
 cout << ans[i] << " ";
}
cout << "\n";
}

return 0;
}

```



### TIME TO TRY

#### DQUERY

Given a sequence of  $n$  numbers  $a_1, a_2, \dots, a_n$  and a number of d-queries. A d-query is a pair  $(i, j)$  ( $1 \leq i = j \leq n$ ). For each d-query  $(i, j)$ , you have to return the number of distinct elements in the subsequence  $a_i, a_{i+1}, \dots, a_j$ .

<http://www.spoj.com/problems/DQUERY/>





## 9

# Dynamic Programming

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

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

**Q. Where do we need Dynamic Programming?**

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

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

**Example**

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

"What's that equal to?"

Counting "Eight!"

Writes down another "1+" on the left.

"What about that?"

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

"You just added one more!"

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

**Dynamic Programming and Recursion:**

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

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

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

**Fibonacci (n) = 1; if n = 0**

**Fibonacci (n) = 1; if n = 1**

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

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

A code for it using pure recursion:

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

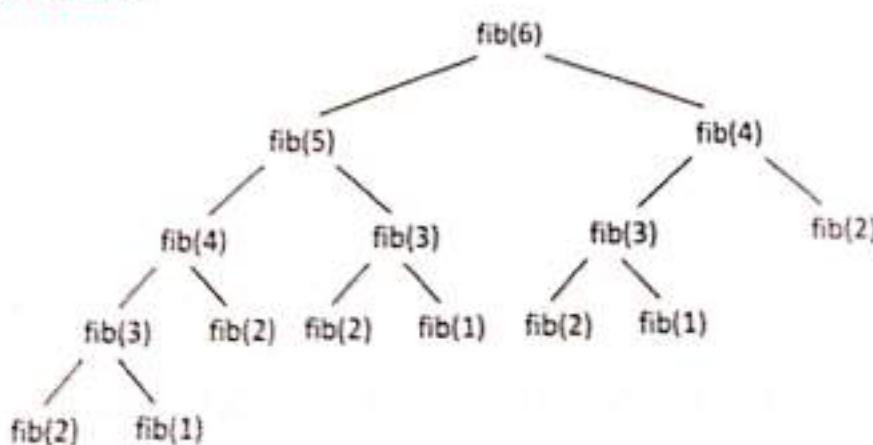
Using Dynamic Programming approach with memoization:

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

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

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

Let's Visualize :



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

**Optimization Problems :**

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

**□ MINIMUM STEPS TO ONE**

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

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

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

Cannot apply Greedy! (Why?)

**Recursive Solution :**

We will define the recurrence relation as

`solve(n):`

```
if n = 1
ans = 0
if n > 1
ans = min{1 + solve(n-1), 1 + solve(n/2), 1 + solve(n/3)}
```

**Code :**

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

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

Can we do better?

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

### Dynamic Programming Solution :

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

*Example :*

I will be an amazing coder. How?  
I will work hard like crazy. How?  
I'll practice more and try to improve. How?  
I'll start taking part in contests. How?  
I'll practicing. How?  
I'm going to learn programming.

*Code :*

```
int minStep(int n){
 //if already calculated, return this answer
 if(dp[n] != -1)
 return dp[n];

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

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

*Example :*

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

*Code :*

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

## □ MINIMUM COIN CHANGE

Given a value N, if we want to make change for N cents, and we have infinite supply of each of C = {C<sub>1</sub>, C<sub>2</sub>, ..., C<sub>M</sub>} valued coins, what is the minimum number of coins to make the change?

*Example :*

Input: coins[] = {25, 10, 5}, N = 30

Output: Minimum 2 coins required

We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, N = 13

Output: Minimum 3 coins required

We can use one coin of 6 + 6 + 1 cents coins.

**Recursive Solution :**

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

If N == 0, then 0 coins required.

If N > 0

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

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

**Code :**

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

**Dynamic Programming Solution :**

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

**1. Top Down DP**

**Code :**

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

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

Code :

```

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

```

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

## □ WINE AND MAXIMUM PRICE

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

Because the wines get better every year, supposing today is the year 1, on year  $y$  the price of the  $i$ th wine will be  $y \cdot p_i$ , i.e.  $y$ -times the value that current year.

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

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

So, for example, if the prices of the wines are (in the order as they are placed on the shelf, from left to right):  $p_1=1, p_2=4, p_3=2, p_4=3$ . The optimal solution would be to sell the wines in the order  $p_1, p_4, p_3, p_2$  for a total profit  $1 \cdot 1 + 3 \cdot 2 + 2 \cdot 3 + 4 \cdot 4 = 29$ .

### Wrong Solution!

#### Greedy Approach :

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

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

At  $t = 1$  year: {2,3,5,1,4} sell  $p_1 = 2$  to get cost = 2

At  $t = 2$  years: {3,5,1,4} sell  $p_2 = 3$  to get cost =  $2 + 2 \cdot 3 = 8$

At  $t = 3$  years: {5,1,4} sell  $p_5 = 4$  to get cost =  $8 + 4 \cdot 3 = 20$

At  $t = 4$  years: {5,1} sell  $p_4 = 1$  to get cost =  $20 + 1 \cdot 4 = 24$

At  $t = 5$  years: {5} sell  $p_3 = 5$  to get cost =  $24 + 5 \cdot 5 = 49$

Greedy approach gives an optimal answer of 49, but if we sell in the order of  $p_1, p_5, p_4, p_2, p_3$  for a total profit  $2 \cdot 1 + 4 \cdot 2 + 1 \cdot 3 + 3 \cdot 4 + 5 \cdot 5 = 50$ , greedy fails.

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

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

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

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

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

**Code :**

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

### Can we do better?

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

**Top Down DP Code :**

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

```

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

```

Solution: <http://pastebin.com/Yx1FG3ys>

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

According to the definition of states, we define:

```

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

```

#### Bottom Up DP Code :

```

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

```

```

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

```

**Solution:** <http://pastebin.com/idsyg4aw>

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

## □ PROBLEMS INVOLVING GRIDS

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

### Finding Minimum-Cost Path in a 2-D Matrix

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

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

**Code :**

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

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

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

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

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

```

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

```

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

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

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

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

$\text{numWays}(i,j) = \text{numWays}(i-1,j) + \text{numWays}(i,j-1)$

Code :

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

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

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

<https://www.codechef.com/problems/CDLIT4>

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

```

int numWays(int Ro, int Col)
{
 //if the initial block is blocked, we cannot move further
 if(dp[0][0] == -1)
 {
 return 0;
 }

 //Number of ways for the first row
 for(int j = 0;j<Col;j++)
 {
 //If any cell is blocked in the first row, we can not visit any
 //cell that are to the right of this blocked cell
 if(dp[0][j] == -1)
 break;

 //There is only one way to reach this cell i.e. from left cell
 dp[0][j] = 1;
 }

 //Number of ways for first column
 for(int i = 0;i<Ro;i++)
 {
 //Similar to first row
 if(dp[i][0] == -1)
 break;

 dp[i][0] = 1;
 }

 //This bottom-up approach ensures that all the sub-problems needed
 //have already been calculated.
 for(int i = 1;i<Ro;i++)
 {
 for(int j = 1;j<Col;j++)
 {
 //If we encounter a blocked cell, do nothing

```

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

Solution : <http://pastebin.com/LeAhedeQ>

#### Another Variant :

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

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

Let us analyse this problem in steps:

The boy can meet the girl in only one cell.

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

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

$(i,j-1) \rightarrow (i,j) \rightarrow (i,j+1)$

$(i,j-1) \rightarrow (i,j) \rightarrow (i+1,j)$

$(i-1,j) \rightarrow (i,j) \rightarrow (i,j+1)$

$(i-1,j) \rightarrow (i,j) \rightarrow (i+1,j)$

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

$(i,j-1) \rightarrow (i,j) \rightarrow (i,j+1)$

$(i,j-1) \rightarrow (i,j) \rightarrow (i-1,j)$

$(i+1,j) \rightarrow (i,j) \rightarrow (i,j+1)$

$(i+1,j) \rightarrow (i,j) \rightarrow (i-1,j)$

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

Boy:  $(i,j-1) \rightarrow (i,j) \rightarrow (i,j+1)$  and Girl:  $(i+1,j) \rightarrow (i,j) \rightarrow (i-1,j)$

or

Boy:  $(i-1,j) \rightarrow (i,j) \rightarrow (i+1,j)$  and Girl:  $(i,j-1) \rightarrow (i,j) \rightarrow (i,j+1)$

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

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

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

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

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

The meeting cell can range from  $2 \leq i \leq n-1$  and  $2 \leq j \leq m-1$

See the code below for more details:

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

```

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

```

```

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

```

**Solution:** <http://pastebin.com/cnBqeJEP>

## □ BUILDING BRIDGES

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

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

## □ LONGEST INCREASING SUBSEQUENCE

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 9, 3, 5, 4, 11, 7, 8} is 4 and LIS is {3, 4, 7, 8}.

**Recurrence relation :**

$L(i) = 1 + \max\{ L(j) \}$  where  $0 < j < i$  and  $\text{arr}[j] < \text{arr}[i]$ ;

or

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

return  $\max(L(i))$  where  $0 < i < n$

**Code :**

```

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

```

```

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

```

$T = O(n^2)$

We can also print the Longest Increasing Subsequence as:

```

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

```

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

Voila! You have a solution :)

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

Code :

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

Solution:<http://pastebin.com/UdHtgasV>

### □ LONGEST COMMON SUBSEQUENCE

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

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

**Example :**

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

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

**Recurrence Relation :**

$\text{LCS}(\text{str1}, \text{str2}, m, n) = 0$ , if  $m = 0$  or  $n = 0$  //Base Case

$\text{LCS}(\text{str1}, \text{str2}, m, n) = 1 + \text{LCS}(\text{str1}, \text{str2}, m-1, n-1)$ , if  $\text{str1}[m] = \text{str2}[n]$

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

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

**Code :**

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

```

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

```

$T = O(mn)$

□ SHORTEST COMMON SUPERSEQUENCE OF TWO STRINGS

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

```
str1 = "apple", str2 = "les"
"apples"

str1 = "AGGTAB", str2 = "GXTXAYB"
"AGGXTXAYB"

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

**Approach:**

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

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

 int i = m, j = n, index = LCS[m][n];
 //array containing the final LCS string lcs_arr;
 while (i > 0 && j > 0)
 {
 // If current character in str1[] and str2[] are same, then
 // current character is part of LCS
```

```

if (str1[i-1] == str2[j-1])
{
 // Put current character in result
 lcs_arr += str1[i-1];
 // reduce values of i, j and index
 i--;
 j--;
 index--;
}

// If not same, then find the larger of two and
// go in the direction of larger value
else if (LCS[i-1][j] > LCS[i][j-1])
 i--;
else
 j--;

}

//Print LCS
//cout<<lcs_arr<<endl;
//Use LCS to get the final answer
i = m-1, j = n-1;
//Use to make the final array
int l = 0, k = 0;
//contains the length of longest common subsequence of the two strings
index = LCS[m][n];
//string containing the final answer
string ans = "";
while(i>=0 || j>=0)
{
 //If we have not covered the full LCS array
 if(l < index)
 {
 //Find the non-LCS characters of A and put them in a array
}

```

```

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

```

Solution: <http://pastebin.com/EB9U8PNu>

EDIT DISTANCE

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

- Insert       Remove       Replace

All the above operations are of equal cost.

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

*Example :*

str1 = "cat"

str2 = "cut"

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

str1 = "sunday"

str2 = "saturday"

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

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

**Recurrence Relation :**

```

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

```

**Transform "Sunday" to "Saturday" :**

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

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

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

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

(S,Sa) → (S,S) //Insert 'a', cost = 1

(S,S) → ("") //Ignore 'S', cost = 0

("") → return 0

```

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

```

```

 dp[i][j-1]), //Insert
 dp[i-1][j-1] //Replace
);
}
}
//Return the most optimal solution
return dp[m][n];
}

```

**Solution:** <http://pastebin.com/ZVqfmHHJ>

## □ MIXTURES

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

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

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

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

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

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

**Code :**

```

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

```

```

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

```

## □ 0-1 KNAPSACK PROBLEM

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

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

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

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

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

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

#### Recurrence Relation :

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

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

Let us define the dp solution with states  $i$  and  $j$  as

$dp[i,j]$   $\rightarrow$  max value that can be obtained with objects up to index  $i$  and knapsack capacity of  $j$ .

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

#### Code :

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

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

Time Complexity = O(NW)

Space Complexity = O(NW)

### Can we do better?

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

**Space-Optimized DP Code :**

```

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

```

**Time Complexity:**  $O(N \cdot W)$ **Space Complexity:**  $O(W)$

**ROD CUTTING PROBLEM**

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

**Example :**

Let the length of the rod is 6.

Price of size is given below

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

**Sol.** Possible number cut on given rod :

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

Max Revenue : 16 (3,3)

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

**Optimal Substructure Property :**

Let the maximum revenue be  $r(n)$ .

then,  $r(n) = \max [p(n), p(1) + r(n - 1), p(2)+r(n-2), \dots, p(n-1)+r(1)]$

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

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

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

**Code :**

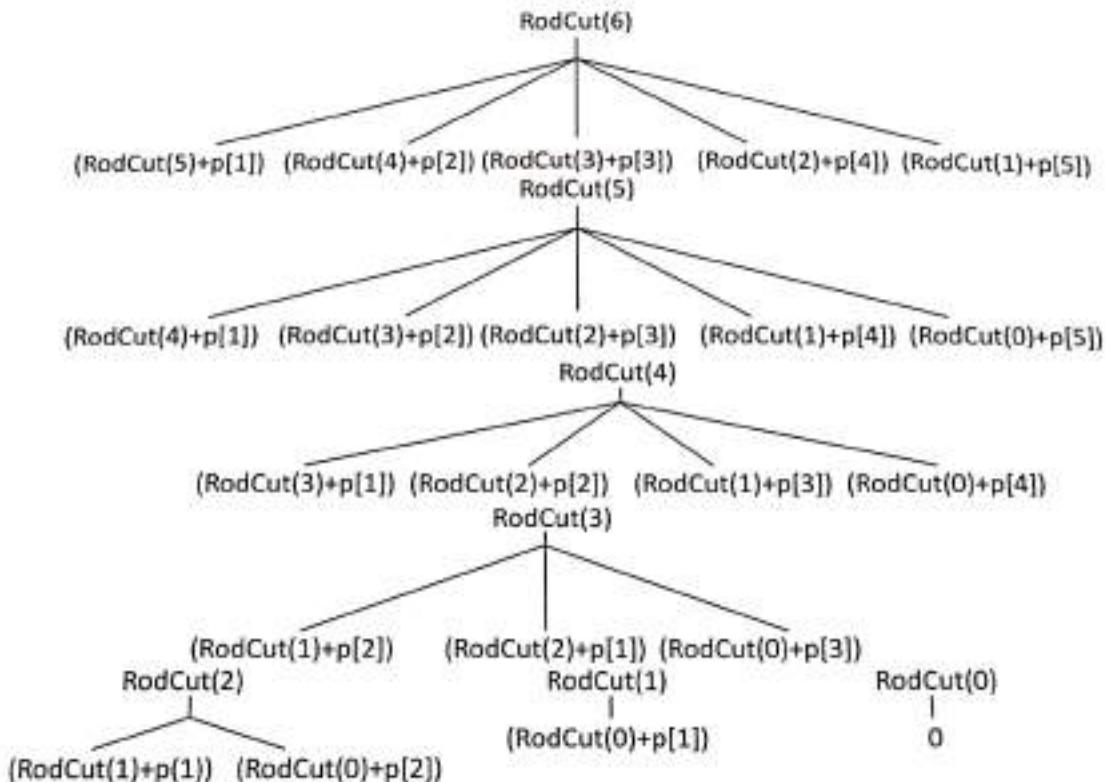
```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b)
{
 return (a>b)?a:b;
}
```

```

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

```

## Overlapping Subproblems :



**Time Complexity :** Exponential Time Complexity

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

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

when we sum up over all possibilities for  $0 \leq k \leq n - 1$  then

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

## Top Down Method :

```

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

```

```

int max(int x, int y)
{
 return (x>y)?x:y;
}
int RodCut(int p[], int n)
{
 int Revenue[n+1];
 Revenue[0]=0;
 for(int i=1;i<=n;i++)
 {
 int best_price = INT_MIN;
 for(int j=1;j<=i;j++)
 {
 best_price = max(best_price, p[j]+Revenue[i-j]);
 }
 Revenue[i]=best_price;
 }
 return Revenue[n];
}
int main()
{
 /* code */
 int t;
 cin>>t;
 for(int i=0;i<t;i++)
 {
 int n;
 cin>>n;
 int p[n+1];
 p[0]=0;
 for(int i=1;i<=n;i++)
 {
 cin>>p[i];
 }
 cout<<"Maximum Revenue of given rod of length "<< n << " =
 <<RodCut(p,n)<<"\n";
 }
 return 0;
}
int Revenue[MAX];
int max(int a, int b)

```

```

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

```

```

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

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

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

```

```

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

```

#### □ LONGEST PALINDROME SUBSEQUENCE

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

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

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

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

#### Optimal substructure Property :

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

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

**Recursive Relation :**

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

**Base Cases :**

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

Start the code with  $i = 0$  and  $j = n - 1$ ;

**Code :**

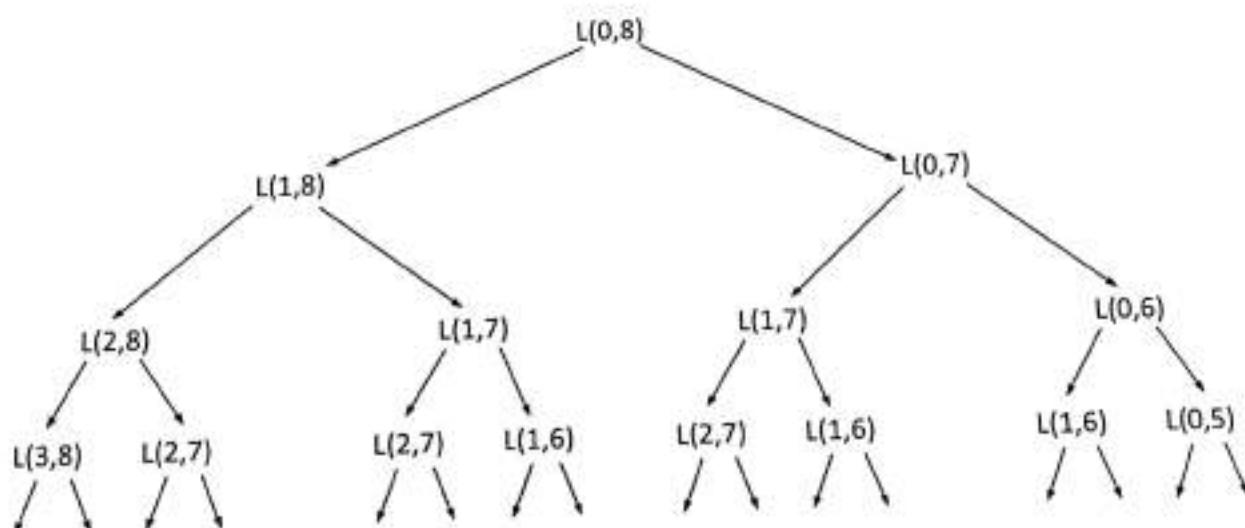
```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int SubSeq(char seq[], int i, int j)
{
 int length_SubSeq = 0;
 if(i==j)
 return 1;
 else if(i==(j-1) && seq[i]==seq[j])
 {
 return 2;
 }
 else if(seq[i]==seq[j])
 length_SubSeq = SubSeq(seq,i+1,j-1)+2;
 else
 length_SubSeq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

```

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

```

## Recursion Tree (Overlapping Subproblems) :



## Top Down (Memoization) :

```

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

```

```

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

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

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

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

```

## Bottom Up (Tabulation):

```

#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int SubSeq(char seq[], int n)
{
 int length[n][n];
 for(int i=0;i<n;i++)
 length[0][i]=1;
 for(int k=2;k<=n;k++)
 {
 for(int i=0;i<=(n-k);i++)
 {
 int j = k+i-1;
 if(k==2 && seq[i]==seq[j])
 {
 length[i][j]=2;
 }
 else
 {
 if(seq[i]==seq[j])
 {
 length[i][j] = 2+length[i+1][j-1];
 }
 else
 {
 length[i][j]=max(length[i][j-1],length[i+1][j]);
 }
 }
 }
 }
 return length[0][n-1];
}
int main()
{
 int t;
 cin>>t;
 for(int i=0;i<t;i++)
 {
 int n;
 cout<<"Enter sequence: ";
 string str;
 cin>>str;
 cout<<"Length of longest palindromic subsequence is: ";
 cout<<SubSeq(str.c_str(), n);
 cout<<endl;
 }
}

```

```

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

```

### MATRIX CHAIN MULTIPLICATION

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

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

There are multiple ways to find this.

- |                                                |                                               |                                              |
|------------------------------------------------|-----------------------------------------------|----------------------------------------------|
| 1. $((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$   | 2. $((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4))$ | 3. $(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$ |
| 4. $((((A_1 \cdot A_2) \cdot A_3) \cdot A_4))$ | 5. $(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$  |                                              |

#### Multiplication Cost :

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

Then multiplication cost for 1st multiplication would be :

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

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

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

Total Cost =  $(40 \times 20 \times 30) + (30 \times 10 \times 30) + (40 \times 30 \times 30) = 70200$  Size of Resultant Matrix =  $40 \times 30$

Similarly cost for other multiplication would be given below :

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

**Approach :**

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

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

$$\text{size of } A_i = p[i-1] * p[i], \forall 1 \leq i \leq n$$

Let  $m[i,j]$  be the minimum number scalar multiplication needed to compute the multiplication of matrix  $A_i$  to  $A_j$  (i.e.  $A_1 \cdot A_{1+1} \cdot A_{1+2} \cdots \cdots \cdot A_j$ ).

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

**A recursive Solution :**

We can split the product the product  $A_1 \cdot A_{1+1} \cdot A_{1+2} \cdots \cdots \cdot A_j$  between  $A_k$  and  $A_{k+1}$  where  $i \leq k < j$  then  $m[i,j]$  equals the minimum cost for computing the subproducts  $A_{i,\dots,k}$  and  $A_{k+1,\dots,j}$  plus the cost of multiplying these two matrices together.

- Size of  $A_{i,\dots,k} = p[i-1]*p[k]$  and size of  $A_{k+1,\dots,j} = p[k]*p[j]$
  - then multiplication cost of  $A_{i,\dots,k} * A_{k+1,\dots,j} = p[i-1] * p[k] * p[j]$
  - $A_{i,\dots,k}$  = minimum cost to multiply the matrices  $(A_1 \cdot A_{1+1} \cdot A_{1+2} \cdots \cdots \cdot A_k) = m[i, k]$
  - $A_{k+1,\dots,j}$  = minimum cost to multiply the matrices =  $m[k+1, j]$
  - $m[i, j] = m[i, k] + m[k+1, j] + p[i-1] * p[k] * p[j]$
- $$m[i, j] = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]$$

Possible value of  $k$  is  $j - i$  i.e.  $k = i, i+1, i+2, \dots, j-1$

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

If  $i == j$

$$m[i, j] = 0;$$

else

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

**Code :**

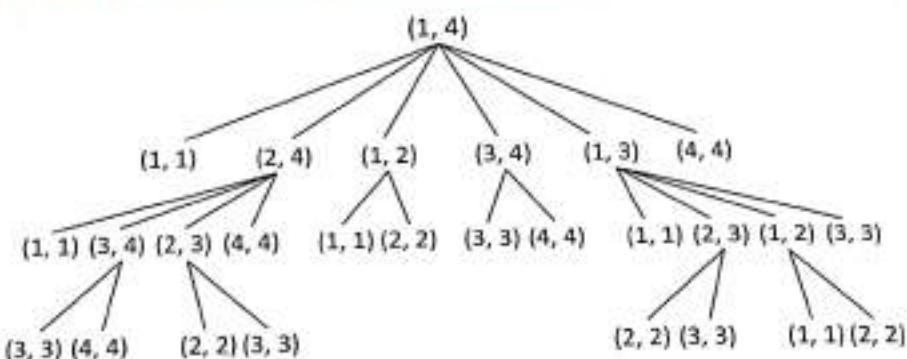
```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int min(int x, int y)
{
 return (x>y)?y:x;
}
```

```

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

```

## Recursive Tree Diagram (Overlapping Subproblems) :



## Top Down (Memoization) :

Code :

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

```

```

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

```

```

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

```

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

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



### DO IT YOURSELVES

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

### SELF STUDY NOTES





**10**

## MO's Algorithm

### Query SQRT Decomposition

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

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

**Let's Start with a Simple Problem :**

Given an array of size N. All elements of array  $\leq N$ . You need to answer M queries. Each query is of the form L, R. You need to answer the count of values in range [L, R] which are repeated at least 3 times. Example: Let the array be {1, 2, 3, 1, 1, 2, 1, 2, 3, 1} (zero indexed)

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

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

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

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

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

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

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

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

```

if count[array[position]] == 2:
 answer = ...

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

```

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

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

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

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

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

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

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

### How does the final order look like?

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

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

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

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

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

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

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

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

### Proof for complexity of above algorithm – $O(\text{Sqrt}(N) * N)$ :

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

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

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

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

### Let us talk about the right pointer first.

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

Let us see how the left pointer moves.

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

So, total complexity is  $O((N + M) * \sqrt{N}) = O(N * \sqrt{N})$ .

This Algorithm is Offline :

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

### DQUERY (Spoj)

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

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

How'll we sort query according to above defined way:
bool compare(query x, query y) {
 if (x.l / BLOCK != y.l / BLOCK) {
 // different blocks, so sort by block.
 return x.l / BLOCK < y.l / BLOCK;
 }
 // same block, so sort by R value
 return x.r < y.r;
}
```

Add and Remove :

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

```

}

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

Main :
int main() {

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

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

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

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

```

```

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

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

```

### □ POWERFUL ARRAY (CODE FORCES, 86-D)

An array of positive integers  $a_1, a_2, \dots, a_n$  is given. Let us consider its arbitrary subarray  $a_l, a_{l+1}, \dots, a_r$ , where  $1 \leq l \leq r \leq n$ . For every positive integer  $s$  denote by  $K_s$  the number of occurrences of  $s$  into the subarray. We call the power of the subarray the sum of products  $K_s \cdot K_s \cdot s$  for every positive integer  $s$ . The sum contains only finite number of nonzero summands as the number of different values in the array is indeed finite.

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

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

Second line contains  $n$  positive integers  $a_i$  ( $1 \leq a_i \leq 10^6$ ) — the elements of the array.

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

**Code :**

```

11 arr[200005];
11 ans[200005];
11 cnt[1000005];

11 tans;

struct query {

```

```

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

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

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

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

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

 fastRead_int(n), fastRead_int(t);

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

```

```

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

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

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

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

return 0;
}

```



## TIME TO TRY

## Tree and Queries :

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

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

## INPUT :

```
85
12 2 3 3 2 3 3
12
15
23
24
56
57
58
12
13
14
23
53
```

## OUTPUT :

```
2
2
1
0
1
```





## 11

## Graph Algorithms

**Introduction :**

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

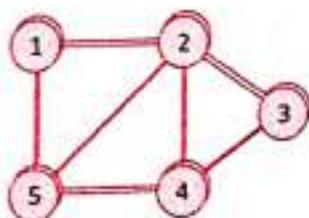
- Nodes** : These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- Edges** : Edges are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

**Some Real Life Applications:**

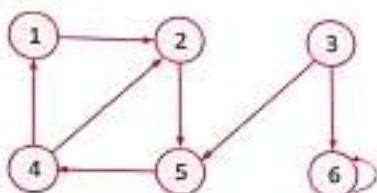
- Google Maps** : To find a route based on shortest route/Time.
- Social Networks** : Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
- Web Search** : Google, to search for webpages, where pages on the internet are linked to each other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

**Types of Graphs :**

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



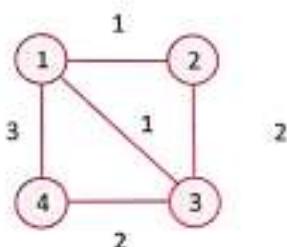
- Directed** : A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



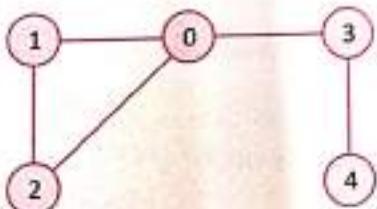
- Weighted:** In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:

➤ 1 → 2 → 3    ➤ 1 → 3    ➤ 1 → 4 → 3

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



- Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that does not contain any cycles.

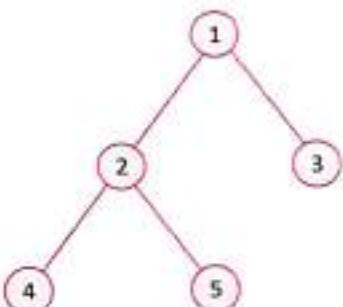


Cycle presents in the above graph (0->1->2)

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

**Note:** A root node has no parent.

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



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

### Graph Representation

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

#### 1. Adjacency Matrix

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

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

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

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

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

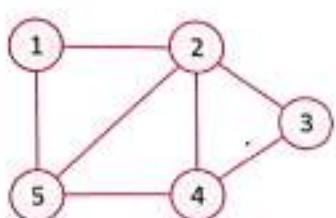
Adjacency matrix provides constant time access ( $O(1)$ ) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is  $O(V^2)$ .

#### 2. Adjacency List

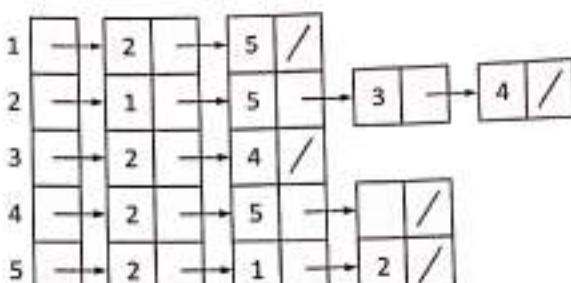
The other way to represent a graph is by using an adjacency list. An adjacency list is an array  $A$  of separate lists. Each element of the array  $A_i$  is a list, which contains all the vertices adjacent to vertex  $i$ .

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

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



(a)



(b)

|   | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

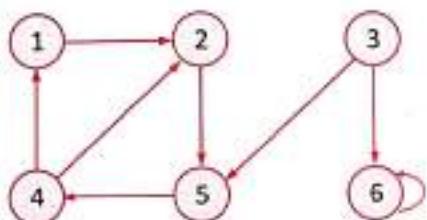
(c)

Two representations of an undirected graph.

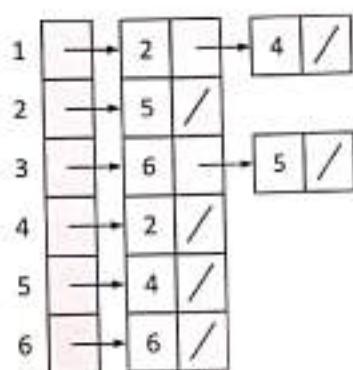
(a) An undirected graph G having five vertices and seven edges

(b) An adjacency-list representation of G

(c) The adjacency-matrix representation of G



(a)



(b)

|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 1 | 0 | 1 |

Code for Adjacency list representation of a graph :

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

```

 l[u].push_back(v);
 if(bidir){
 l[v].push_back(u);
 }
 }
 void printAdjList(){
 for(int i=0;i<V;i++){
 cout<<i<<"->";
 //l[i] is a linked list
 for(int vertex: l[i]){
 cout<<vertex<<",";
 }
 cout<<endl;
 }
 }
};

int main(){
 // Graph has 5 vertices number from 0 to 4
 Graph g(5);
 g.addEdge(0,1);
 g.addEdge(0,4);
 g.addEdge(4,3);
 g.addEdge(1,4);
 g.addEdge(1,2);
 g.addEdge(2,3);
 g.addEdge(1,3);
 g.printAdjList();
 return 0;
}

```

### Graph Traversal :

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

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

While using  
once. The order  
question that

**Breadth First Search (BFS) :**

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

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

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

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

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

Breadth-first search constructs a breadth-first tree, initially containing only its root,  $v = s$ , the source vertex  $s$ . Whenever a white vertex  $v$  is discovered in the course of scanning the adjacency list of an already discovered vertex  $u$ , the vertex  $v$  and the edge  $\{u, v\}$  are added to the tree, so that  $u$  is the *predecessor* or *parent* of  $v$  in the breadth-first tree.

**Algorithm :**

```

BFS(G, s)
 for each vertex $u \in V[G] - \{s\}$
 do color[u] \leftarrow WHITE
 d[u] $\leftarrow \infty$
 $\pi[u] \leftarrow \text{NIL}$
 color[s] \leftarrow GRAY
 d[s] $\leftarrow 0$
 $\pi[s] \leftarrow \text{NIL}$
 Q $\leftarrow \emptyset$
 ENQUEUE(Q, s)
 while Q $\neq \emptyset$
 do $u \leftarrow \text{DEQUEUE}(Q)$
 for each $v \in \text{Adj}[u]$
 do if color[v] = WHITE
 then color[v] \leftarrow GRAY
 $\pi[v] \leftarrow u$
 d[v] $\leftarrow d[u] + 1$
 ENQUEUE(Q, v)

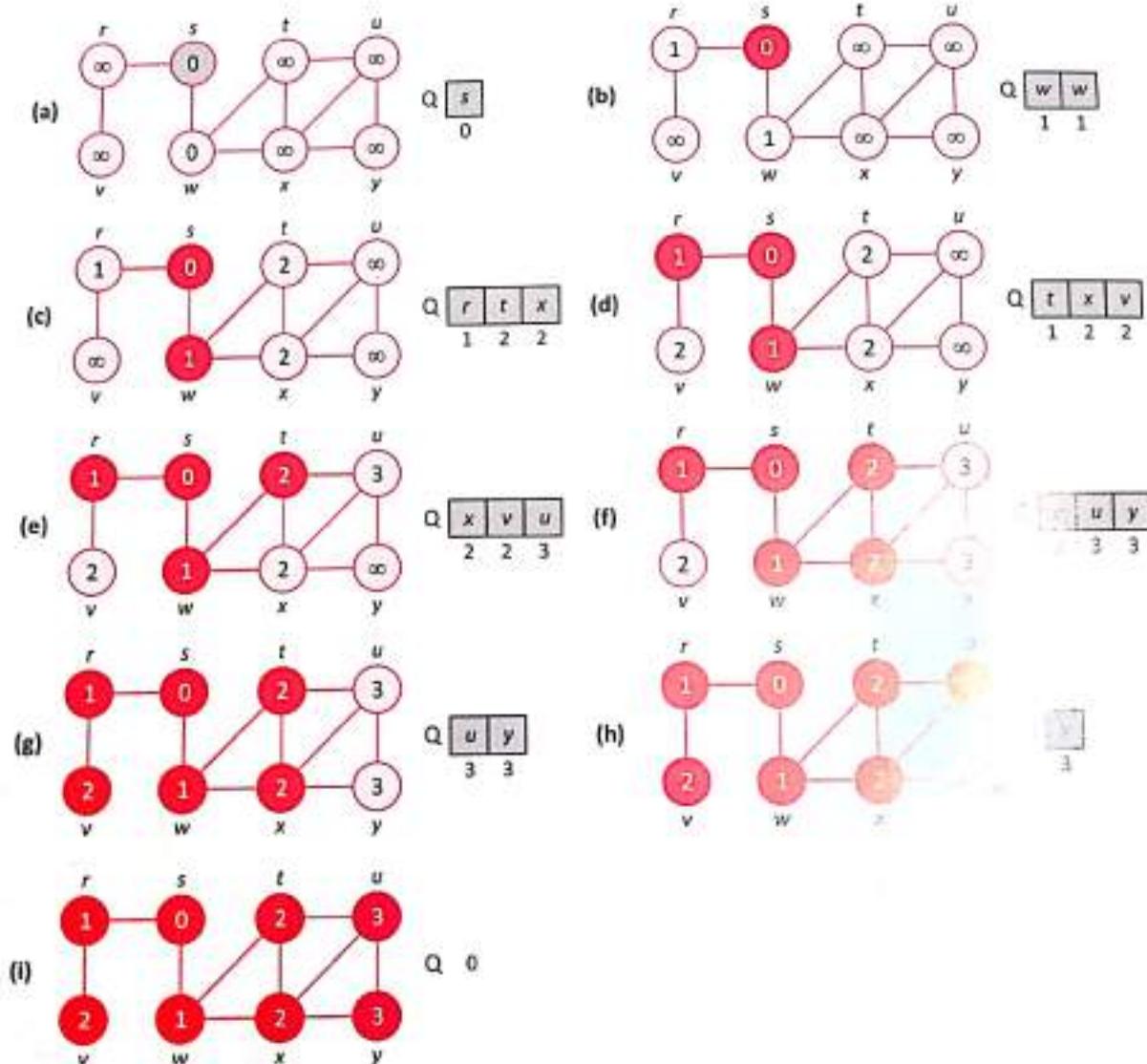
```

```

 $d[v] \leftarrow d[u] + 1$
 $\pi[v] \leftarrow u$
ENQUEUE(Q, v)

```

color[u]  $\leftarrow$  BLACK



The operation of BFS on an undirected graph. Tree edges are shown shaded grey. The distance from the source vertex vertex u is shown  $d[u]$ . The queue Q is shown at the beginning of each step. Vertex distances are shown next to vertices in the queue.

duced by BFS. Within each iteration of the while loop of lines 10-18,

```

Code :

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

template<typename T>
class Graph{

 map<T,list<T> > adjList;

public:
 Graph(){
 }

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

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

 void print(){
}

```

**Time Complexity :**

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

**Application of BFS****1. Shortest Path and Minimum Spanning Tree for unweighted graph**

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

**Code for finding shortest path using BFS :**

```

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

```

```
#include<queue>
using namespace std;
template<typename T>
class Graph{
 map<T,list<T> > adjList;
public:
 Graph(){
 }
 void addEdge(T u, T v,bool bidir=true){
 adjList[u].push_back(v);
 if(bidir){
 adjList[v].push_back(u);
 }
 }
 void print(){
 //Iterate over the map
 for(auto i:adjList){
 cout<<i.first<<"->";
 //i.second is LL
 for(T entry:i.second){
 cout<<entry<<",";
 }
 cout<<endl;
 }
 }
 void bfs(T src){
 queue<T> q;
 map<T,int> dist;
 map<T,T> parent;
 for(auto i:adjList){
 dist[i.first] = INT_MAX;
 }
 q.push(src);
 }
}
```

```

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

```

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

### Depth First Search (DFS)

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

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

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

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

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

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

#### Algorithm :

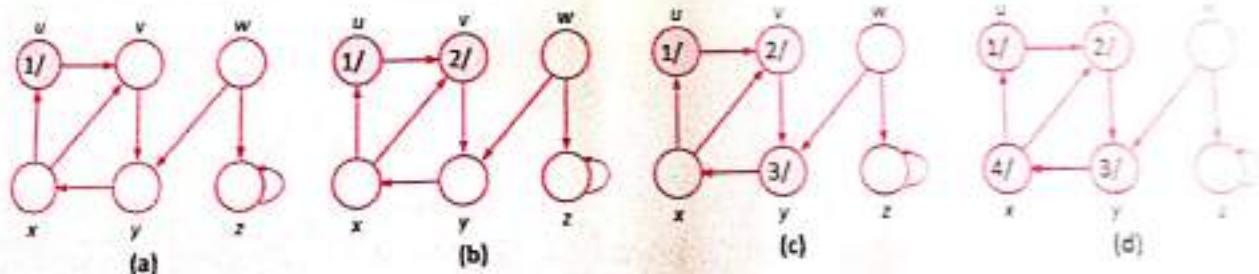
```

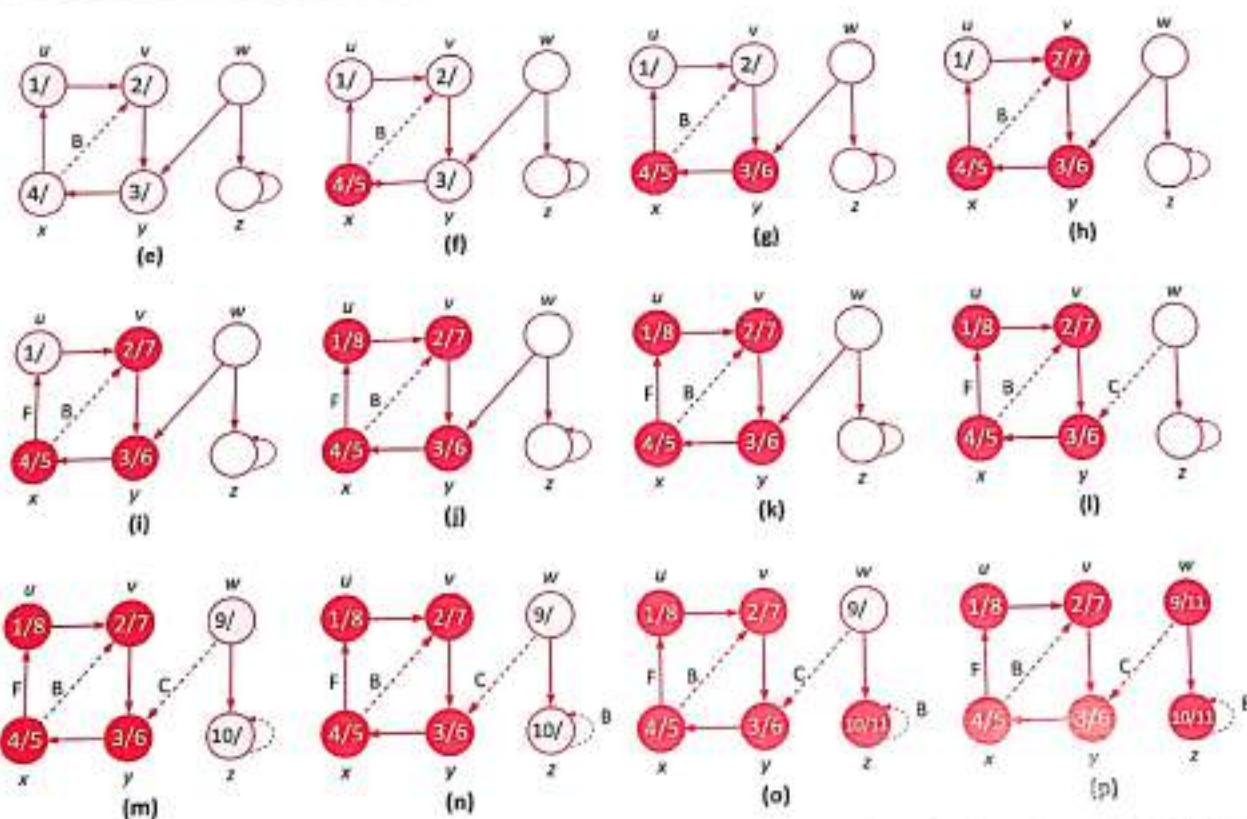
DFS(G)
for each vertex $u \in V[G]$
 do color[u] \leftarrow WHITE
 $p[u] \leftarrow$ NIL
 time $\leftarrow 0$
for each vertex $u \in V[G]$
 do if color[u] = WHITE
 then DFS-VISIT(u)

DFS-VISIT(u)
color[u] \leftarrow GRAY White vertex u has just been discovered
time \leftarrow time + 1
 $d[u] \leftarrow$ time
for each $v \in \text{Adj}[u]$ Explore edge (u, v)
 do if color[v] = WHITE
 then $\pi[v] \leftarrow u$
 DFS-VISIT(v)

color[u] \leftarrow BLACK Blacken u ; it is finished
 $f[u] \leftarrow$ time \leftarrow time + 1

```





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

**Code :**

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

template<typename T>
class Graph{
 map<T,list<T> > adjList;

public:
 Graph(){
 }

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

```

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

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

 //i.second is LL
 for(T entry:i.second){
 cout<<entry<<",";
 }
 cout<<endl;
 }
 }

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

 void dfs(T src){
 map<T,bool> visited;
 dfsHelper(src,visited);
 }
};

int main(){

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

```

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

 return 0;
}
```

### Time Complexity :

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

### Application of DFS :

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

### Code for detect a cycle in a graph :

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

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

**3. Path Finding :** We can specialize the DFS algorithm to find a path between two given vertices  $u$  and  $z$ .

- (i) Call  $\text{DFS}(G, u)$  with  $u$  as the start vertex.
- (ii) Use a stack  $S$  to keep track of the path between the start vertex and the current vertex.
- (iii) As soon as destination vertex  $z$  is encountered, return the path as the contents of the stack.

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

**Code for printing Topological sorting of DAG :**

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

```

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

```

```

 }

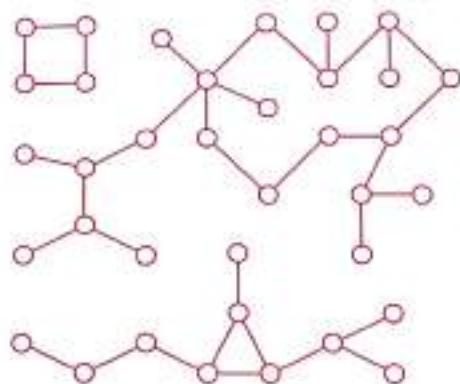
};

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

 return 0;
}

```

- 5. To test if a graph is bipartite :** We can augment either BFS or DFS where we start from a vertex, color it opposite its parents, and for each other edge, check if both vertices of the same color. The first vertex in any connected component can be red or blue.
- 6. Finding Strongly Connected Components of a graph :** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
- 7. Solving puzzles with only one solution,** such as mazes. (DFS can be adapted to not visit a node in a maze by only including nodes on the current path in the visited set.)
- 8. For finding Connected Components :** In graph theory, a connected component (or cluster) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

*Example :*

In the above picture, there are three connected components.

**Code for finding connected components :**

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

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

```

 for(T entry:i.second){
 cout<<entry<<",";
 }
 cout<<endl;
 }
}

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

void dfs(T src){
 map<T,bool> visited;
 int component = 1;
 dfsHelper(src,visited);
 cout<<endl;
 for(auto i:adjList){
 T city = i.first;
 if(!visited[city]){
 dfsHelper(city,visited);
 component++;
 }
 }
 cout<<endl;
 cout<<"The current graph had "<<component<<" components";
}
};

int main(){
 Graph<string> g;

```

```

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

```

### Minimum Spanning Tree

#### What is a Spanning Tree?

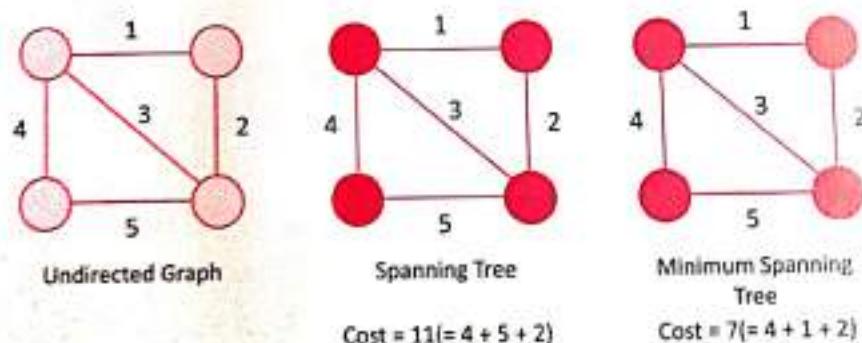
Given an undirected and connected graph  $G=(V,E)$ , a spanning tree of the graph  $G$  is a tree that spans  $G$  (that is, it includes every vertex of  $G$ ) and is a subgraph of  $G$  (every edge in the tree belongs to  $G$ )

#### What is a Minimum Spanning Tree?

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

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

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



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

### Kruskal's Algorithm

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

#### Algorithm Steps

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

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

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

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

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

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

#### Algorithm :

MST-KRUSKAL( $G, w$ )

$A \leftarrow \emptyset$

For each vertex  $v \in V[G]$

Do MAKE-SET( $v$ )

sort the edges of  $E$  into nondecreasing order by weight  $w$

for each edge  $(u, v) \in E$ , taken in nondecreasing order by weight

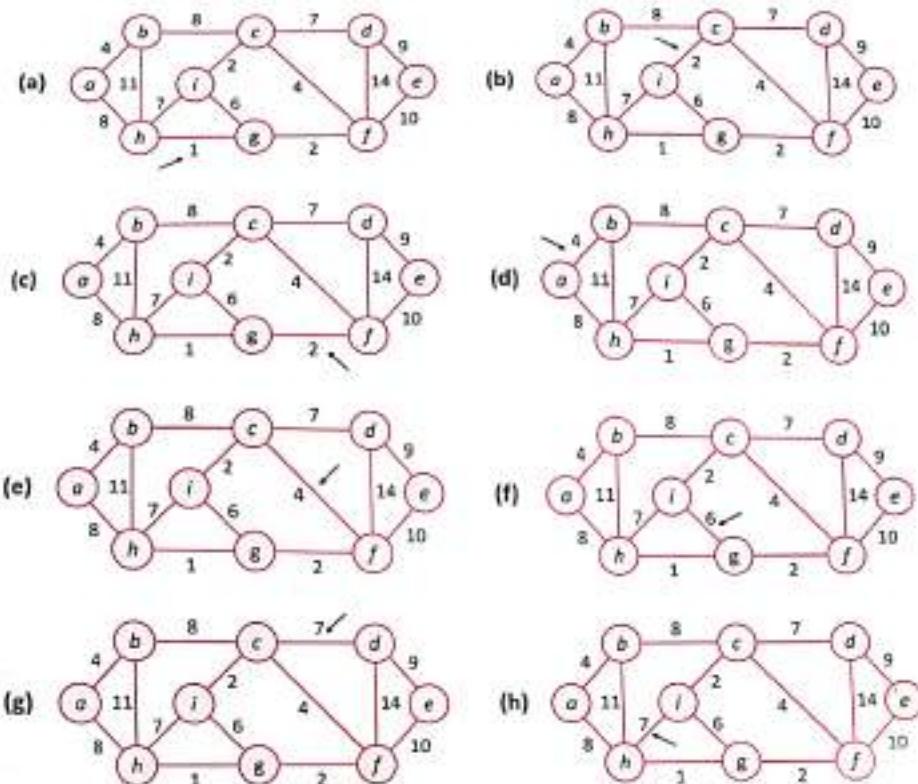
do if FIND-SET( $u$ ) ≠ FIND-SET( $v$ )

then  $A \leftarrow A \cup \{(u, v)\}$

UNION( $u, v$ )

return  $A$

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



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

**Code :**

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

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

void initialize()
{
 for(int i = 0; i < MAX; ++i)
 id[i] = i;
}
```

```

 id[i] = i;
 }

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

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

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

```

```

 }
 return minimumCost;
}

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

```

**Time Complexity :**

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

**Prim's Algorithm :**

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

**Algorithm Steps :**

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

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

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

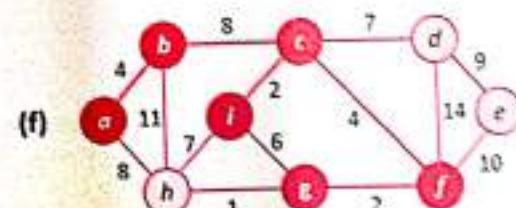
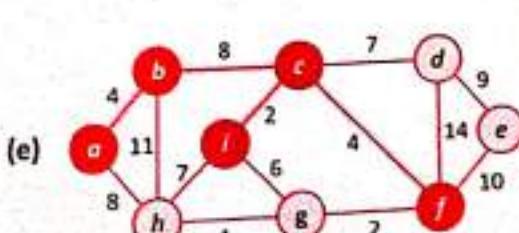
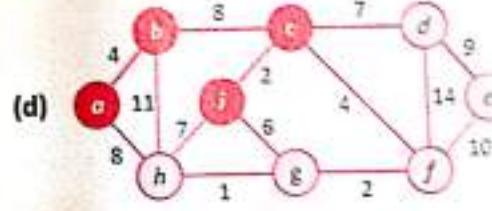
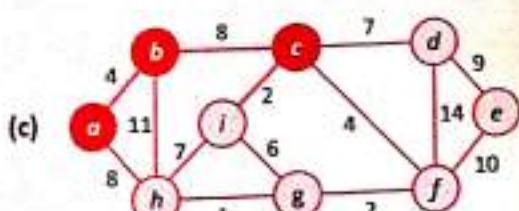
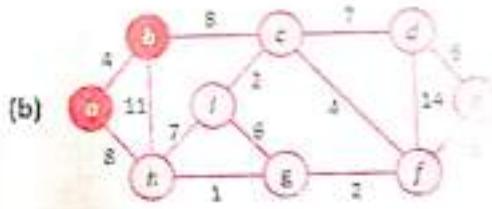
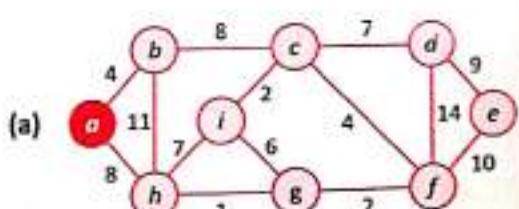
**MST-PRISM( $G, w, r$ )**

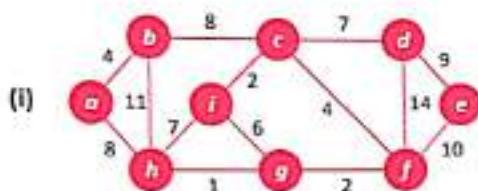
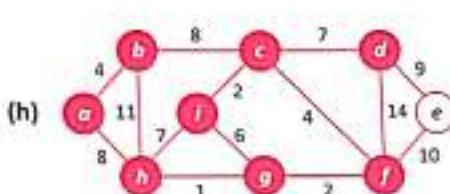
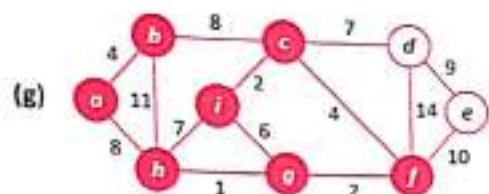
```

for each $u \in V[G]$
 do $key[u] \leftarrow \infty$
 $p[u] \leftarrow \text{NIL}$
 $key[r] \leftarrow 0$
 $Q \leftarrow V[G]$
while $Q \neq \emptyset$
 do $u \leftarrow \text{EXTRACT-MIN}(Q)$
 for each $v \in \text{Adj}[u]$
 do if $v \in Q$ and $w(u, v) < key[v]$
 then $\pi[v] \leftarrow u$
 $key[v] \leftarrow w(u, v)$

```

The prims algorithm works as shown in below graph.





Code :

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

long long prim(int x)
{
 priority_queue<PII, vector<PII>, greater<PII> > Q;
 int y;
 long long minimumCost = 0;
 PII p;
 Q.push(make_pair(0, x));
 while(!Q.empty())
 {
 // Select the edge with minimum weight
 p = Q.top();
 Q.pop();
 x = p.second;
 // Checking for cycle
 if(marked[x])
 continue;
 marked[x] = true;
 minimumCost += p.first;
 for(int i = 0; i < adj[x].size(); i++)
 {
 if(!marked[adj[x][i].second])
 Q.push(adj[x][i]);
 }
 }
 return minimumCost;
}
```

```

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

```

### Shortest Path Algorithms

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

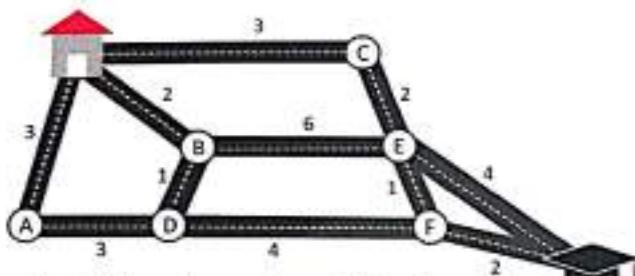
This problem could be solved easily using (BFS) if all edge weights were (1), but here weights can take any value. There are some algorithm discussed below which work to find Shortest path between two vertices.

### 1. Single Source Shortest Path Algorithm :

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

**Example :**

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



A weighted graph representing roads from home to school

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

Home → B → D → F → School

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

#### A. Dijkstra's Algorithm

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

**Algorithm Steps:**

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

**Pseudo Code :**

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

```

do $u \leftarrow \text{EXTRACT-MIN}(Q)$
 $S \leftarrow S \cup \{u\}$
for each vertex $v \in \text{Adj}[u]$
 do $\text{RELAX}(u, v, w)$

```

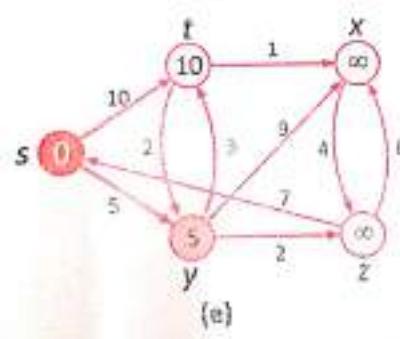
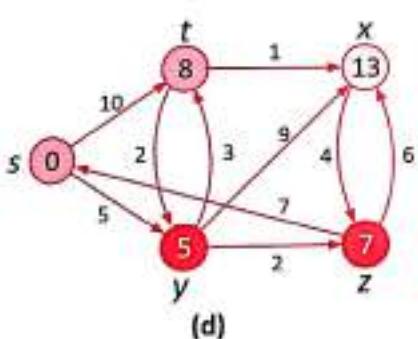
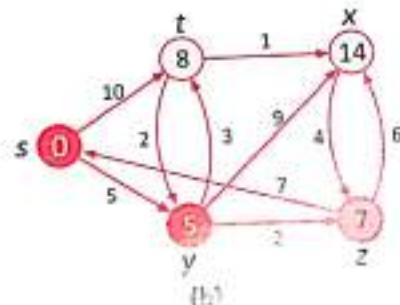
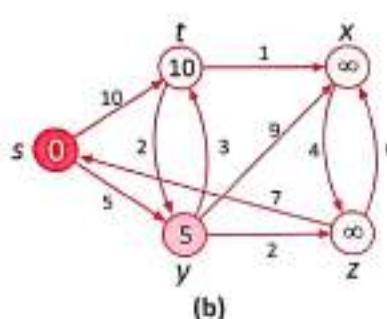
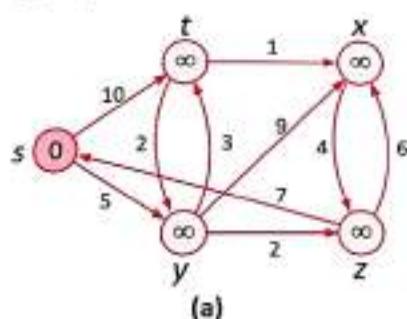
For relaxing an edge of given graph, Algorithm works like this :

```

 $\text{RELAX}(u, v, w)$
if $d[v] > d[u] + w(u, v)$
then $d[v] \leftarrow d[u] + w(u, v)$
 $\pi[v] \leftarrow u$

```

**Example :**



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

**Code :**

```

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

```

```

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

```

```

//Iterate over neighbours/children of the current node
for(auto childPair: m[node]){
 if(nodeDist + childPair.second < dist[childPair.first]){
 //In the set updation of a particular is not possible
 // we have to remove the old pair, and insert the new pair
 to simulation updation
 T dest = childPair.first;
 auto f = s.find(make_pair(dist[dest],dest));
 if(f!=s.end()){
 s.erase(f);
 }
 //Insert the new pair
 dist[dest] = nodeDist + childPair.second;
 s.insert(make_pair(dist[dest],dest));
 }
}
}

//Lets print distance to all other node from src
for(auto d:dist){
 cout<<d.first<<" is located at distance of "<<d.second;
}
}

};

int main(){
 Graph<int> g;
 g.addEdge(1,2,1);
 g.addEdge(1,3,4);
 g.addEdge(2,3,1);
 g.addEdge(3,4,2);
 g.addEdge(1,4,7);
 //g.printAdj();
 // g.dijkstraSSSP(1);
 Graph<string> india;
 india.addEdge("Amritsar","Delhi",1);
 india.addEdge("Amritsar","Jaipur",4);
 india.addEdge("Jaipur","Delhi",2);
}

```

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

return 0;
}
```

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

### B. Bellman Ford's Algorithm

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

So why shortest path shouldn't have a cycle ?

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

Algorithm Steps:

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

This algorithm depends on the relaxation principle where the shortest distance to a vertex is gradually replaced by more accurate values until eventually reaching the minimum. At the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex is zero. Then update all the connected vertices with the new distances (source vertex to its neighbors), then apply the same concept for the new vertices with new distances and so on.

Pseudo Code :

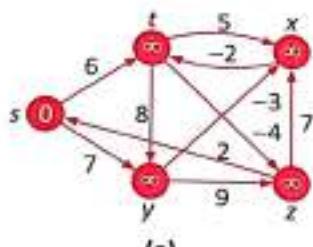
```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| - 1
 do for each edge (u, v) ∈ E[G]
 do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
```

```

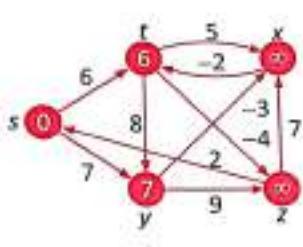
do if $d[v] > d[u] + w(u, v)$
 then return FALSE
return TRUE

```

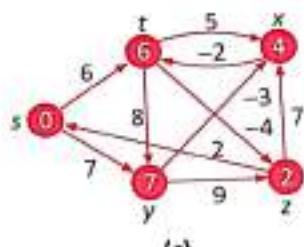
**Example :**



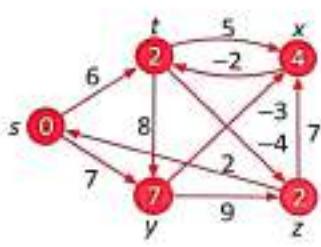
(a)



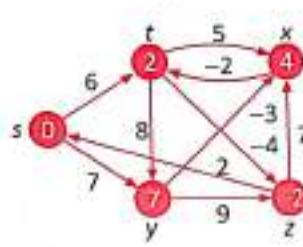
(b)



(c)



(d)



(e)

### All-Pairs Shortest Paths :

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

#### 1. Floyd-Warshall's Algorithm

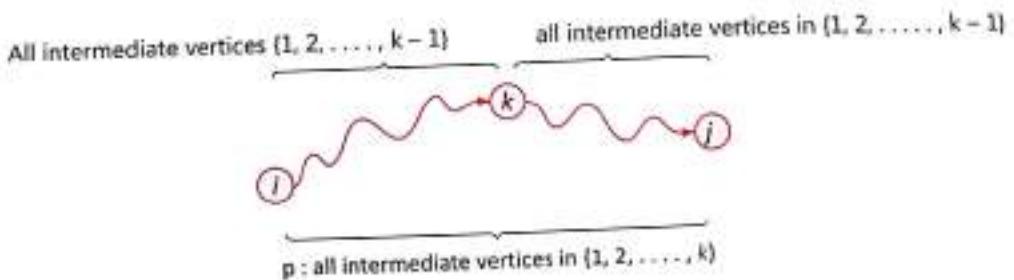
#### 2. Johnson's Algorithm

### Floyd-Warshall's Algorithm

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

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

- If  $k$  is not an intermediate vertex of path  $p$ , then all intermediate vertices of path  $p$  are in the set  $\{1, 2, \dots, k-1\}$ . Thus, a shortest path from vertex  $i$  to vertex  $j$  with all intermediate vertices in the set  $\{1, 2, \dots, k-1\}$  is also a shortest path from  $i$  to  $j$  with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ .
- If  $k$  is an intermediate vertex of path  $p$ , then we break  $p$  down into  $i \rightarrow k \rightarrow j$  as  $p_1$  is a shortest path from  $i$  to  $k$  and  $p_2$  is a shortest path from  $k$  to  $j$ . Since  $p_1$  is a shortest path from  $i$  to  $k$  with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ . Because vertex  $k$  is not an intermediate vertex of path  $p_1$ , we see that  $p_1$  is a shortest path from  $i$  to  $k$  with all intermediate vertices in the set  $\{1, 2, \dots, k-1\}$ . Similarly,  $p_2$  is a shortest path from vertex  $k$  to vertex  $j$  with all intermediate vertices in the set  $\{1, 2, \dots, k-1\}$ .



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

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

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

### The Algorithm Steps:

For a graph with  $V$  vertices:

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

$$\text{Min}(\text{dist}[i][k] + \text{dist}[k][j], \text{dist}[i][j])$$

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

test paths that use

air using the first K

**Constructing a shortest path**

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

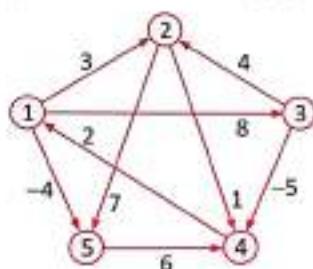
For  $k = 0$ :

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

And For  $1 \leq k \leq V$

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

Consider the below graph and find distance (D) and parents ( $\pi$ ) matrix for this graph



Computed distance (D) and parents ( $\pi$ ) matrix for the above graph :

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

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

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

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

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

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

**Implemented Code :**

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

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

```

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

}

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

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

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

```

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

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

```

 cout<<p[i][j]<<" ";
 else
 {
 cout<<"NIL"<<" ";
 //cout<<p[i][j]<<" ";
 }
}
cout<<endl;
}
}

//All pair shortest path matrix i.e D
void printSolution(int dist[][V])
{
 cout<<"Following matrix shows the shortest distances"
 " between every pair of vertices"<<"\n";
 for (int i = 0; i < V; i++)
 {
 for (int j = 0; j < V; j++)
 {
 if (dist[i][j] == -1)
 cout<<"INF ";
 else
 cout<<dist[i][j]<<" ";
 }
 cout<<endl;
 }
}

//Print the shortest path , distance and all intermediate vertex
void print_path(int p[][V], int d[][V])
{
 // cout<<"Hello"<<"\n";
 for(int i=0;i<V;i++)
 {
 for(int j=0;j<V;j++)
 {
 if (d[i][j] != -1)
 cout<<"Path from vertex "<<i<<" to vertex "<<j<<" is ";
 cout<<p[i][j]<<" ";
 }
 }
}

```

```

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

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

```

```

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

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

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

 printSolution(dist);
 cout<<"\n\n";
 printParents(parent2);
 cout<<"\n\n";
 cout<<"All pair shortest path is given below :"\<<"\n";
 print_path(parent2, dist);
 }
};

//Main Function
int main()

```

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

}
```



## DO IT YOURSELVES

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

Stress - Hyperactive

High - Stress - Hyperactive

Stress - Hyperactive



Stress - Hyperactive



## 12

## Combinatorial Game Theory

**Introduction :**

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

**Let's see an Example :**

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

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

If  $N=1$  then?

If  $N=2$  then?

If  $N=3$  then?

If  $N=4$  then?

If  $N=5$  then?

If  $N=6$  then?

Once you start playing this game, you will realise that it isn't much fun. You will need to come up with a strategy which will let you win for certain values of  $N$ . Eventually you will find that if both players play smartly, the winner of the game is decided entirely by the initial value of  $N$ .

be able to devise  
out that if both  
of  $N$ .

**Strategy :**

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

$N$  is a multiple of 3.

#### **Finders Keepers game :**

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

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

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

PROBLEMS

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

**Sol:** If A picks 2, B will be left with 72 and no matter what number B picks, A can always pick  $\{8 - x\}$  and wrap up.

2. In a game of keepers-losers B started the game when there were N coins on the table. If B is confident of winning the game and  $a = 3$ ,  $b = 5$ , which of the following cannot be a value of N?  
 (a) 94                    (b) 92                    (c) 76                    (d) 66

**Sol: (d)** In keeper-losers, the motto is to give opponent  $8k + 1$  coins to win for sure. Whatever B does, he cannot give 65 coins to A.

3. A and B play Finders-winners with 56 coins, where A plays first and  $a = 1$ ,  $b = 1$ . Should B pick immediately after A?

(a) 2 (b) -  
The number of coins A picks depends on what B picks. But the number of coins B picks depends on what A picks.

4. In a game of Keepers-losers played with 126 coins where A plays first and a winner? who is the

**Sol:** In order to win, A should leave  $9k+1$ ,  $9k+2$  or  $9k+3$  coins on the table, since B can take 1, 2 or 3 coins and hence can win the game.

5. In an interesting version of game B gets to choose the number of coins on the board. A gets to choose the format of the game it will be as well as pick coins first. If B chooses 18 coins, what format should A choose in order to win?

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

### Properties of the above Games :

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

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

### Game of NIM

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

#### SOLUTION :

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

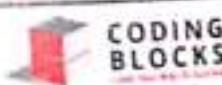
Nim-Sum : The cumulative XOR value of the number of coins/stones in each of the piles at any point of the game is called Nim-Sum at that point.

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

#### Why does it work?

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

- if  $\oplus$  of the sizes of the piles is 0 then it will be changed after our move (one 1 will be changed to 0, so in that column will be odd number of 1s).
- From the winning positions it is possible to move to at least one losing position.
- if  $\oplus$  of the sizes of the piles is not 0 we can change it to 0 by finding a column where the number of 1s is odd, changing one of them to 0 and then by changing the rest of it to gain even number of 1s in every column.



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

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

### Sprague-Grundy Theorem

A game composed of K solvable subgames with Grundy numbers  $G_1, G_2 \dots G_K$  is winnable iff the Nim game composed of Nim piles with sizes  $G_1, G_2 \dots G_K$  is winnable.

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

### What is Mex Function ?

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

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

$\text{Mex}(S) = 0$

$S_1 = \{0, 1, 3, 4, 5\}$

$\text{Mex}(S_1) = 2$

### Calculating Grundy Numbers

The game starts with a pile of n stones. Calculate the Grundy Numbers for all possible states of the game?

$\text{Grundy}(0) = ?$

$\text{Grundy}(1) = ?$

$\text{Grundy}(n) = \text{Mex } (\{0, 1, 2, \dots, n-1\}) = ?$

int calculateMex(set<int> Set)

{

    int Mex = 0;

    while (Set.find(Mex) != Set.end())

        Mex++;

    return (Mex);

}

// A function to Compute Grundy Number of 'n'

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

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

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

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

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

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

```

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

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

return (calculateMex(Set));
}

```

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

```

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

```

#### How to apply Sprague Grundy theorem ?

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

## PROBLEMS

## QCJ3

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

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

**Hint :**

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

Then we use the property of xor operator

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

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

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

```

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

return 0;
}

```

**Try it yourself !**

#### M&M Game

<http://www.spoj.com/problems/MMMGAME/>

**Hint :** This is a Miser Game.

#### Game of Stones

<https://www.hackerrank.com/challenges/game-of-stones-1/problem>

#### Piles Again

(Variation of Nim, HackerBlocks)

#### Game Theory-1

(Grundy Numbers, HackerBlocks)

#### Game Theory-2

(Sprague Grundy Thm, HackerBlocks)

#### Game Theory-3

(Sprague Grundy Thm, HackerBlocks)

#### MORE QUESTIONS FOR PRACTICE

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

### Combinatorial Game Theory

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

### SELF STUDY NOTES

International Game Show

WORLD'S FAIR



## 13

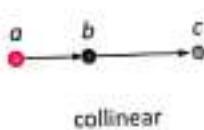
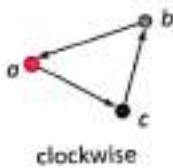
## Geometric Algorithms

**How to check if two line segments intersect?**

Given two line segments  $(p_1, q_1)$  and  $(p_2, q_2)$ , find if the given line segments intersect with each other.

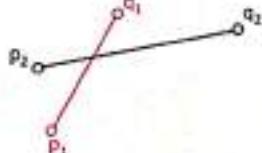
Orientation of an ordered triplet of points in the plane can be

- counterclockwise
- clockwise
- collinear

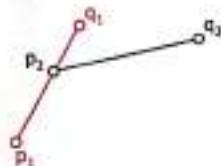


Two segments  $(p_1, q_1)$  and  $(p_2, q_2)$  intersect if and only if one of the following two conditions is verified:

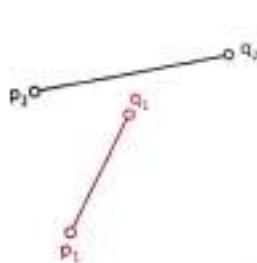
1.  $(p_1, q_1, p_2)$  and  $(p_1, q_1, q_2)$  have different orientations and  $(p_2, q_2, p_1)$  and  $(p_2, q_2, q_1)$  have different orientations.



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



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



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



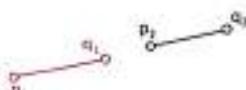
**Example 4:** Orientations of  $(p_1, q_1, p_2)$  and  $(p_1, q_1, q_2)$  are different. Orientations of  $(p_2, q_2, p_1)$  and  $(p_2, q_2, q_1)$  are also different

2.  $(p_1, q_1, p_2)$ ,  $(p_1, q_1, q_2)$ ,  $(p_1, q_2, p_2)$ , and  $(p_1, q_2, q_1)$  are all collinear and

- the x-projections of  $(p_1, q_1)$  and  $(p_2, q_1)$  intersect
- the y-projections of  $(p_1, q_1)$  and  $(p_2, q_1)$  intersect



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



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

### How to compute Orientation?

Use Cross Product :)

Code:

```
//cross product of OA and OB, this function will tell the orientation of OAB
ll cross(Point o, Point a, Point b) {
 int val = ((a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x));
 if(val == 0) return 0;
 return (val > 0 ? 2 : 1); //1 = clockwise 2 = anticlockwise
}

struct Point
{
 int x;
 int y;
};

// Given three colinear points p, q, r
// point q lies on line segment pr
bool onSegment(Point p, Point q, Point r)
{
 if (q.x <= max(p.x, r.x) && q.x >=
 q.y <= max(p.y, r.y) && q.y >=
 return true;

 return false;
}

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

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

// Case 1
if (o1 != o2 && o3 != o4)
 return true;

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

// p1, q1 and q2 are colinear and q2 lies on segment p1q1
if (o2 == 0 && onSegment(p1, q2, q1)) return true;

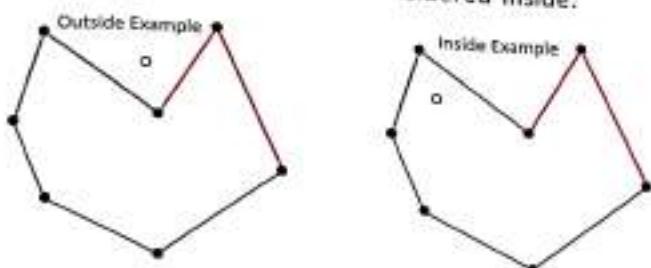
// p2, q2 and p1 are colinear and p1 lies on segment p2q2
if (o3 == 0 && onSegment(p2, p1, q2)) return true;

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

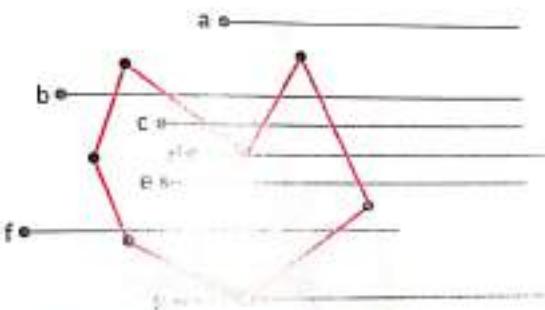
return false;
}
```

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

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



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



```

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

```

```

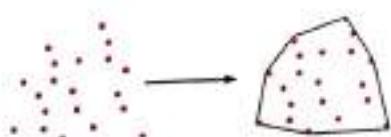
 // then check if it lies on segment. If it lies, return true,
 // otherwise false
 if (orientation(polygon[i], p, polygon[next]) == 0)
 return onSegment(polygon[i], p, polygon[next]);

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

```

**Convex Hull :**

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

**Jarvis Algorithm :**

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

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

```

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

```

```

for (int i = 1; i < n; i++)
 if (points[i].x < points[1].x)
 l = i;
// Start from leftmost point, keep moving counterclockwise
// until reach the start point again.
int p = l, q;
do
{
 // Add current point to result
 hull.push_back(points[p]);
 // Search for a point 'q' such that orientation(p, x, q)
 // is counterclockwise for all points 'x'. The idea
 // is to keep track of last visited most counterclock-
 // wise point in q. If any point 'i' is more counterclock-
 // wise than q, then update q.
 q = (p+1)%n;
 for (int i = 0; i < n; i++)
 {
 // If i is more counterclockwise than current q, then
 // update
 if (orient(points[p], points[i], points[q]) > 0)
 q = i;
 }
 // Now q is the most counterclockwise point
 // Set p as q for next iteration
 // result 'hull'
 p = q;
} while (p != l); // While we
}

```

**Graham Scan :**

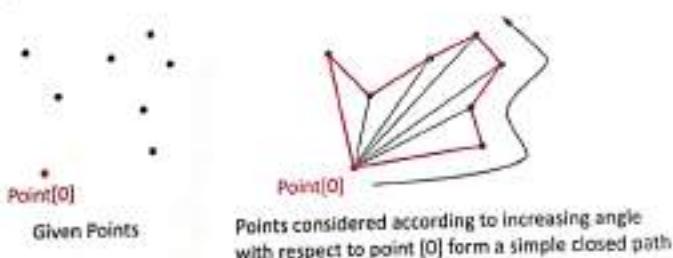
Let  $\text{points}[0 \dots n-1]$  be the input array.

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

2. Consider the remaining  $n-1$  points and sort them by polar angle in counterclockwise order around  $\text{points}[0]$ . If polar angle of two points is same, then put the nearest point first.
3. After sorting, check if two or more points have same angle. If two more points have same angle, then remove all same angle points except the point farthest from  $P_0$ . Let the size of new array be  $m$ .
4. If  $m$  is less than 3, return (Convex Hull not possible)
5. Create an empty stack 'S' and push  $\text{points}[0]$ ,  $\text{points}[1]$  and  $\text{points}[2]$  to  $S$ .
6. Process remaining  $m-3$  points one by one. Do following for every point ' $\text{points}[i]$ '  
 6.1 Keep removing points from stack while orientation of following 3 points is not counterclockwise (or they don't make a left turn).  
 (a) Point next to top in stack      (b) Point at the top of stack  
 (c)  $\text{points}[i]$
- 4.2 Push  $\text{points}[i]$  to  $S$
7. Print contents of  $S$ .

The above algorithm can be divided in two phases :

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



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

```
void convexHull(Point points[], int n)
{
 // Find the bottommost point
 int ymin = points[0].y, min = 0;
 for (int i = 1; i < n; i++)
 if (points[i].y < points[min].y)
 min = i;
 if (min != 0)
 swap(points[0], points[min]);
 int curr = 0, prev = 0, next = 1;
 while (next < n) {
 if ((curr + 1) % n == next)
 curr = next;
 else if (orient(points[curr], points[next], points[(curr + 1) % n]) != -1)
 curr = next;
 else
 next++;
 }
}
```

```

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

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

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

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

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

```

```

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

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

// Process remaining n-3 points
for (int i = 3; i < m; i++)
{
 // Keep removing top while the angle formed by
 // points next-to-top, top, and points[i] makes
 // a non-left turn
 while (orientation(nextToTop(S), S.top(), points[i]) == -1)
 S.pop();
 S.push(points[i]);
}
// Now stack has the output points, print contents of S
}

```

### **Andrew's monotone chain convex hull algorithm :**

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

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

**Input:** a list P of points in the plane.

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

Initialize U and L as empty lists.

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

```

for i = 1, 2, ..., n:
 while L contains at least two points and the sequence of last two points of L and the point P[i] does
 not make a counter-clockwise turn:
 remove the last point from L
 append P[i] to L
 for i = n, n-1, ..., 1:
 while U contains at least two points and the sequence of last two points of U and the point P[i] does
 not make a counter-clockwise turn:
 remove the last point from U
 append P[i] to U
 Remove the last point of each list (it's the same as the first point of the other list).
 Concatenate L and U to obtain the convex hull of P.
 Points in the result will be listed in counter-clockwise order.

```

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

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

### Problem - KTHCON

A 1-concave polygon is a simple polygon which has at least one concave interior angle.

There are N points on a plane. You have to find the area of the largest 1-concave polygon formed by those points. Compute 2S. Note that S is the area of the polygon.

#### Constraints:

$$1 \leq T \leq 100$$

$$3 \leq N \leq 105$$

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

No three points are collinear.

The sum of N over all test cases won't exceed 5·10<sup>4</sup>.

$$|x|, |y| \leq 10^9$$

#### Input:

2

5

22

-2 -2

2 -2

```
-2 2
```

```
0 1
```

```
3
```

```
0 0
```

```
1 0
```

```
0 1
```

**Output**

```
28
```

```
-1
```

**Code :**

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

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

```

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

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

```

```
}

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

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

}

return 0;
}
```

**14****Fast Fourier Transformation****Step-1 FFT**

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

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

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

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

Finally we can write original polynomial as combination of  $A_0$  and  $A_1$ :

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

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

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

$$\begin{aligned} y_{k+n/2} &= A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) = A_0(w_n^{2k} w_n^n) A_1(w_n^{2k} w_n^n) \\ &= A(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^{k+n/2-1} y_k^1 \end{aligned}$$

Finally our point form will be:

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

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

**FFT()**

```

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

```

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

**Time Complexity:** O(nlogn)

**Step 2: Convolution**

**Multiply()**

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

```

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

```

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

### Step 3: Inverse FFT

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

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

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

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

Thus we get the formula:

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

Comparing it with the formula for  $y_k$ :

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

**Inverse\_FFT()**

```

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

 return y;
}

```

**Time Complexity:**  $O(n \log n)$ **Ques:** Very Fast Multiplication<http://www.spoj.com/problems/VFMUL/>

```

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

using namespace std;

```

```

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

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

```

```

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

```

```

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

```



Fast Fourier Transformation

SELF STUDY NOTES



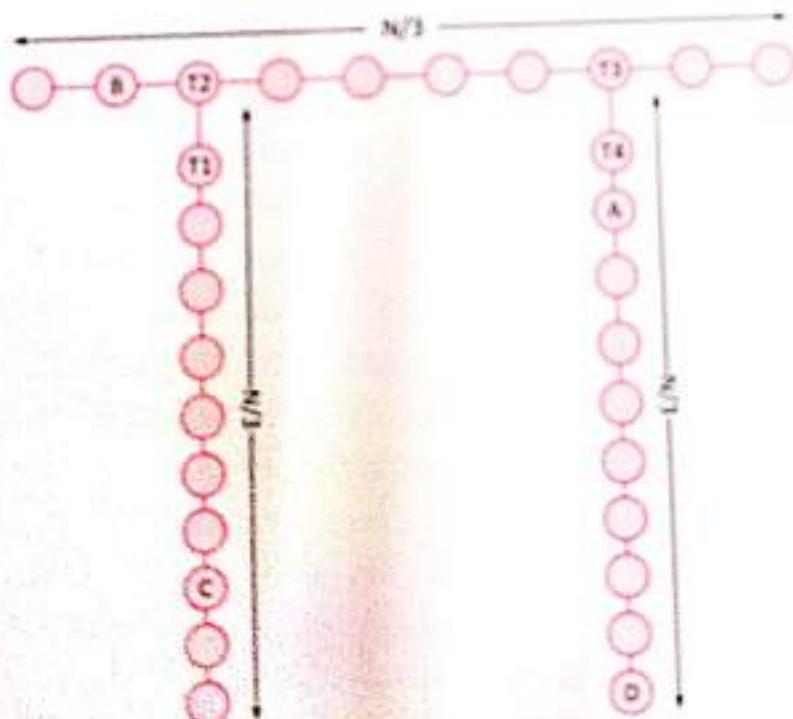
## 15

## Heavy Light Decomposition

## Heavy Light Decomposition

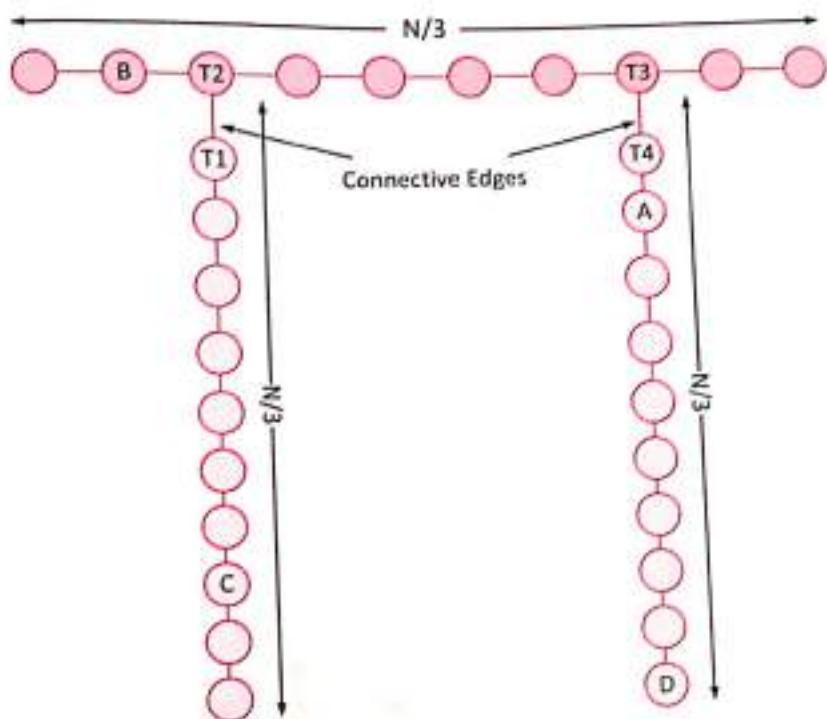
- **Balanced Binary Tree** is good, need to visit at most  $2\log N$  nodes to reach from any node to any other node in the tree.
- If a balanced binary tree with  $N$  nodes is given, then many queries can be done with  $O(\log N)$  complexity. **Distance of a path, Maximum/Minimum in a path, Maximum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

## Unbalanced Trees :



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

What if we broke the tree in to 3 chains?



### Basic Idea

We will divide the tree into **vertex-disjoint chains** ( Meaning no two chains share a common vertex). In such a way that to move from any node in the tree to the root node, we have to change at most  $\log N$  chains. To put it in another words, the path from any node to the root node will pass through such that the each piece belongs to only one chain, then we will have at most  $\log N$  pieces.

**So what?**

Now the path from any node A to any node B can be broken into  $\text{LCA}(A, B)$  and B to  $\text{LCA}(A, B)$ .

### Time Complexity

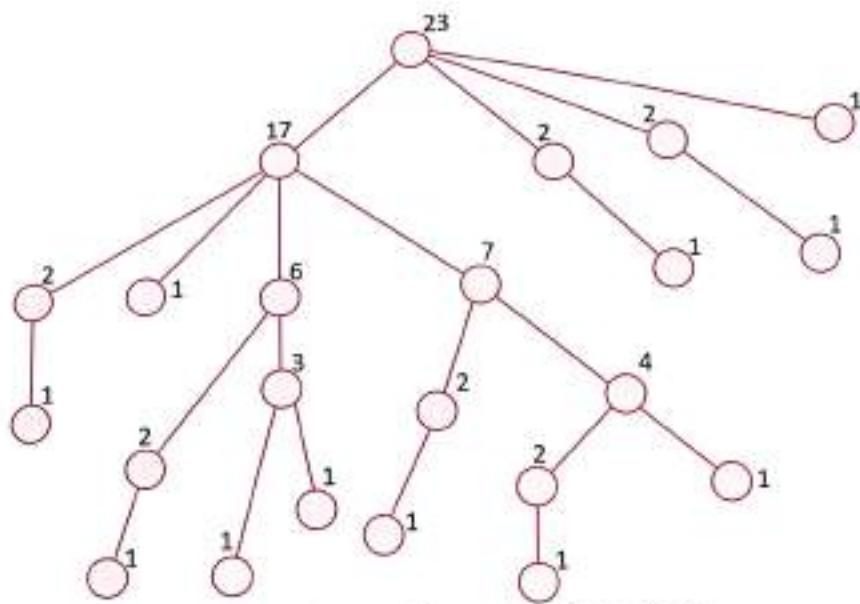
We already know that queries in each chain can be answered in constant time. Since there are at most  $\log N$  chains we need to consider per path, we have  $O(\log^2 N)$  complexity solution.

**Special Child** : Among all child nodes of a node, the one with maximum number of children is called Special child. Each non leaf node has exactly one Special child.

**Special Edge** : For each non-leaf node, the edge connecting the node with its Special child is considered as Special Edge.

Time complexity and therefore we have  $O(\log^2 N)$

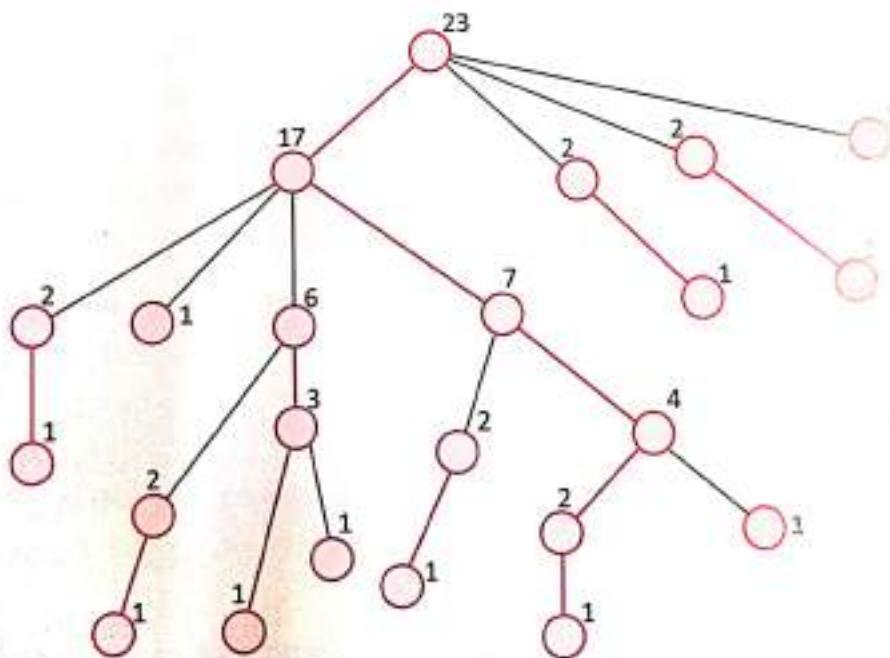
Up-tree size is considered as



Each node has its sub-tree size written on top.

Each non-leaf node has exactly one special child whose sub-tree size is colored red.

Colored child is the one with maximum sub-tree size.



Each chain is represented with different color.

This black lines represent the connecting edges. They connect 2 chains.

**Algorithm**

```

HLD(curNode, Chain):
 Add curNode to curchain
 If curNode is LeafNode: return
 //Nothing left to do
 sc := child node with maximum sub-tree size
 //sc is the special child
 HLD(sc, Chain)
 //Extend current chain to special child
 for each child node cn of curNode:
 //For normal child
 if cn != sc: HLD(cn, newChain) //As told above, for each normal child,
 a new chain starts

```

**dfs()**

```

void dfs(int cur, int prev, int level = 0){
 //pa[cur][0] will be its immediate parent
 pa[cur][0] = prev;
 //setting the depth of the current node
 depth[cur] = level;
 //Initially set the subtree size of every node as 1
 //Add the subtree size of every child node of cur node
 subsz[cur] = 1;
 for (int i = 0; i < adj[cur].size(); i++){
 if (adj[cur][i] != prev){
 otherEnd[ind[cur][i]] = adj[cur][i];
 dfs(adj[cur][i], cur, level + 1);
 subsz[cur] += subsz[adj[cur][i]];
 }
 }
}

```

**LCA()**

```

int LCA(int p, int q){
 if (depth[q] > depth[p]) swap(p, q);
 int diff = depth[p] - depth[q];
 for (int i = 0; i < LN; i++){
 if ((diff >> i) & 1)
 p = pa[p][i];
 }
 //Now p and q are at same level
 if (p == q) return p;
 for (int i = LN - 1; i >= 0; i--){
 if (pa[p][i] != -1 && pa[p][i] != pa[q][i]){
 p = pa[p][i];
 q = pa[q][i];
 }
 }
 return pa[p][0];
}

```

**HLD()**

ChainNo → Chain number of current chain.

chainHead[i] → Chain head of the chain in which i-th node is present.

chainPos[i] → Position of i-th node in its chain.

chainInd[i] → Index of chain in which i-th node is present.

chainSize[chainNo] → Size of chain with chain number.

baseArray[] → array on which segment tree will operate.

all the nodes in the same chain will be adjacent to each other.

posInBase[i] → position of i-th node in base array.

**int** chainNo=0,chainHead[N],chainPos[N],chainInd[N],posInBase[N];

**void** hld(**int** cur,, **int** prev) {

//If we have encountered this chain for the first time

if(chainHead[chainNo] == -1) chainHead[chainNo] = cur;

//This node will be present in current chain

chainInd[cur] = chainNo;

//Position of current node in chain will be present size of chain

chainPos[cur] = chainSize[chainNo];

```

//Increment size of chain
chainSize[chainNo]++;
// Position of this node in baseArray which we will use in Segtree
//Initially ptr was 0
posInBase[curNode] = ptr;
//Insert the cost/value of node in the base array
baseArray[ptr++] = cost;

//Find the maximum sized subtree and its index
int ind = -1, mx_sz = -1;
for(int i = 0; i < adj[cur].sz; i++) {
 //make sure we are not calling HLD for the parent node
 if(adj[cur][i] != prev && subsize[adj[cur][i]] > mx_sz) {
 mx_sz = subsize[adj[cur][i]]; ind = i;
 }
}
//continue the HLD for present chain with specific child as new member
//of current chain
if(ind >= 0) hld(adj[cur][ind], cur);

for(int i = 0; i < adj[cur].sz; i++) {
 //If the child is not special, start a new chain
 if(i != ind) {
 chainNo++;
 hld(adj[cur][i], cur);
 }
}
}
}

query_up()
//v is ancestor of u
//query the chain in which 'u' is present till head of that chain, then move to
next chain up
//we do that till u and v are in the same chain and then we query from u to v
in the same chain :D
int query_up(int u, int v){

```

## Heavy Light Decomposition

```
if (u == v) return 0;
int uchain, vchain = chainInd[v], ans = -1;
while (1){
 //If both u and v are in same chain, query that chain and we are done
 uchain = chainInd[u];
 if (uchain == vchain){
 if (u == v) break;
 ans = max(ans,query_tree(1, 0, sz, posInBase[v] + 1, posInBase[u]));
 break;
 }
 //else, query the u-chain from u to its head and then change the chain
 //by calling query from parent[u] to v
 ans = max(ans,query_tree(1, 0, sz, posInBase[chainHead[uchain]], posInBase[u]));
 u = chainHead[uchain];
 u = pa[u][0];//move to parent of u, we are moving a chain up

 //keep on doing this untill u and v are in same chain
}
return ans;
}

query()
int query(int p, int q){
 int lca = LCA(p, q);
 //path from p to q is divided into path from p to lca
 // (p,q) and q to lca(p,q)
 int ans1 = query_up(p, lca);
 int ans2 = query_up(q, lca);

 //If max query
 int ans = max(ans1,ans2);

 //If sum query
 int ans = ans1 + ans2 - cost[LCA(p,q)];
 return ans;
}
```

**Time Complexity**Build Time  $\rightarrow O(n)$ Query Time  $\rightarrow O(\log n * \log n)$ **Spoj QTREE**

```

http://www.spoj.com/problems/QTREE/
#include<bits/stdc++.h>
typedef long long ll;
#define fast std::ios::sync_with_stdio(false);std::cin.tie(false)
#define endl "\n"
//#define abs(a) a >= 0 ? a : -a
#define ll long long int
#define mod 1000000007
#define Endl endl
using namespace std;

const int N = 10000 + 10;
const int MAX = 1e6 + 10;
const int LN = 14;
vector<int> adj[N], costs[N];
int n, sz, arr[N], tree[N], d[N],
[N], subsz[N];
int chainNo, chainHead[N], chainD[N];
int st[4 * N];

void build(int node, int l, int r){
 if (l == r){
 st[node] = arr[l];
 return;
 }
 int mid = (l + r) >> 1;
 build(2 * node, l, mid);
 build(2 * node + 1, mid + 1, r);
 st[node] = (st[2 * node] > st[2 * node + 1]) ? st[2 * node] : st[2 * node + 1];
}

```

## Heavy Light Decomposition

```
void update(int node, int l, int r, int i, int val){
 if (l > r || l > i || r < i) return;
 if (l == i && r == i){
 st[node] = val;
 return;
 }
 int mid = (l + r) >> 1;
 update(2 * node, l, mid, i, val);
 update(2 * node + 1, mid + 1, r, i, val);
 st[node] = (st[2 * node] > st[2 * node + 1]) ? st[2 * node] : st[2 * node + 1];
}

int query_tree(int node, int l, int r, int i, int j){
 if (l > r || l > j || r < i){
 return -1;
 }
 if (l >= i && r <= j){
 return st[node];
 }
 int mid = (l + r) >> 1;
 return max(query_tree(2 * node, l, mid, i, j), query_tree(2 * node + 1, mid + 1, r, i, j));
}

//v is ancestor of u
//query the chain in which 'u' is present till head of
at chain, then move to next chain up
//we do that till u and v are in the same chain and then
we query from u to v in the same chain :D
int query_up(int u, int v){
 if (u == v) return 0;
 int uchain, vchain = chainInd[v], ans = -1;
 while (1){
 uchain = chainInd[u];
 if (uchain == vchain){
 ans = max(ans, st[u]);
 break;
 }
 u = parent[u];
 }
 return ans;
}
```

```

 if (u == v)break;
 ans = max(ans,query_tree(1, 0, sz, posInBase[v] + 1, posInBase[u]));break;
 }
 ans = max(ans,query_tree(1, 0, sz, posInBase[chainHead[uchain]], posInBase[u]));
 u = chainHead[uchain];
 u = pa[u][0];//move to parent of u, we are moving a chain up
}
return ans;
}

int LCA(int p, int q){
 if (depth[q] > depth[p])swap(p, q);
 int diff = depth[p] - depth[q];
 for (int i = 0; i < LN; i++){
 if ((diff >> i) & 1)
 p = pa[p][i];
 }
 //Now p and q are at same level
 if (p == q) return p;
 for (int i = LN - 1; i >= 0; i--){
 if (pa[p][i] != -1 && pa[p][i] == pa[q][i])
 p = pa[p][i];
 q = pa[q][i];
 }
}
return pa[p][0];
}

int query(int p, int q){
 int lca = LCA(p, q);
 //path from p to q is divided into path from p to lca (p,q) and q to lca (q,lca)
 int ans = query_up(p, lca);
 int temp = query_up(q, lca);
 if (temp > ans)ans = temp;
 return ans;
}

```

## Heavy Light Decomposition

```
void change(int i, int val){
 int p = otherEnd[i];
 update(i, 0, sz, posInBase[p], val);
}

void HLD(int cur, int cost, int prev){
 if (chainHead[chainNo] == -1){
 chainHead[chainNo] = cur;
 }
 chainInd[cur] = chainNo;
 posInBase[cur] = sz;
 arr[sz++] = cost;

 int sc = -1, sc_cost;
 for (int i = 0; i < adj[cur].size(); i++){
 if (adj[cur][i] != prev){
 if (sc == -1 || subsz[sc] < subsz[adj[cur][i]]){
 sc = adj[cur][i];
 sc_cost = costs[cur][i];
 }
 }
 }
 if (sc != -1)HLD(sc, sc_cost, cur);

 for (int i = 0; i < adj[cur].size(); i++){
 if (adj[cur][i] != prev && sc != adj[cur][i]){
 //new chain at each normal node :)
 chainNo++;
 HLD(adj[cur][i], costs[cur][i], cur);
 }
 }
}

void dfs(int cur, int prev, int level = 0){
 pa[cur][0] = prev;
 depth[cur] = level;
```

```

subsz[cur] = 1;
for (int i = 0; i < adj[cur].size(); i++){
 if (adj[cur][i] != prev){
 otherEnd[ind[cur][i]] = adj[cur][i];
 dfs(adj[cur][i], cur, level + 1);
 subsz[cur] += subsz[adj[cur][i]];
 }
}
}

int main(){
 int t;
 scanf("%d", &t);
 while (t--){
 //printf("\n");
 sz = 0;
 scanf("%d", &n);
 for (int i = 0; i < n; i++){
 adj[i].clear();
 costs[i].clear();
 ind[i].clear();
 chainHead[i] = -1;
 for (int j = 0; j < LN; j++) pa[i][j] = -1;
 }
 for (int i = 1; i < n; i++){
 int a, b, c;
 scanf("%d %d %d", &a, &b, &c);
 -a;-b;
 adj[a].push_back(b);
 adj[b].push_back(a);
 ind[a].push_back(i - 1);
 ind[b].push_back(i - 1);
 costs[a].push_back(c);
 costs[b].push_back(c);
 }
 }
}

```

## Heavy Light Decomposition

---

```
chainNo = 0;
dfs(0, -1); //setup subtree size, depth and parent for each node;
//cout << "dfs over !!!" << endl;
HLD(0, -1, -1); //decompose the tree and create the baseArray
//cout << "HLD over" << endl;
build(1, 0, sz);
//cout << "Build over" << endl;
//for (int i = 0; i < n; i++) cout << "edge : " << i << " " << otherEnd[i]
//<< " " << posInBase[i] << endl;

//bottom up DP code for LCA:
for (int j = 1; j < LN; j++){
 for (int i = 0; i < n; i++){
 if (pa[i][j - 1] != -1) pa[i][j] = pa[pa[i][j - 1]][j - 1];
 }
}
char ch[20];
while (1){
 scanf("%s", ch);
 if (ch[0] == 'D') break;
 int a, b;
 scanf("%d %d", &a, &b);
 //cout << ch << " " << a << " " << b << endl;
 if (ch[0] == 'Q')
 printf("%d\n", query(a - 1, b - 1));
 else
 change(a - 1, b);
}
return 0;
}
```

**Spoj QTREE2**

```

http://www.spoj.com/problems/QTREE2/
#include<bits/stdc++.h>
typedef long long ll;
#define fast std::ios::sync_with_stdio(false);std::cin.tie(false)
#define endl "\n"
//#define abs(a) a >= 0 ? a : -a
#define ll long long int
#define mod 1000000007
#define Endl endl
using namespace std;
const int N = 200000 + 10;
const int MAX = 1e6 + 10;
const int LN = 15;
vector<int> adj[N], costs[N], ind[N];
int n, sz, arr[N], tree[N], depth[N], pa[N][LN], subsz[N];
int chainNo, chainHead[N], chainInd[N], chainSz[N];
int st[4 * N];

void build(int node, int l, int r){
 if (l == r){
 st[node] = arr[l];
 return;
 }
 int mid = (l + r) >> 1;
 build(2 * node, l, mid);
 build(2 * node + 1, mid + 1, r);
 st[node] = st[2 * node] + st[2 * node + 1];
}
int query_tree(int node, int l, int r, int i, int j){
 if (l > r || l > j || r < i){
 return 0;
 }
 if (l >= i && r <= j){
 return st[node];
 }
 int mid = (l + r) >> 1;
 return query_tree(2 * node, l, mid, i, j) +
 query_tree(2 * node + 1, mid + 1, r, i, j);
}

int main(){
 int T;
 cin >> T;
 while (T--){
 int n, m;
 cin >> n >> m;
 for (int i = 1; i <= n; i++){
 adj[i].clear();
 ind[i] = 0;
 arr[i] = 0;
 tree[i] = 0;
 depth[i] = 0;
 pa[i][0] = -1;
 for (int j = 1; j < LN; j++)
 pa[i][j] = -1;
 chainNo = -1;
 chainHead[i] = -1;
 chainInd[i] = -1;
 chainSz[i] = -1;
 }
 for (int i = 1; i < m; i++){
 int u, v;
 cin >> u >> v;
 adj[u].push_back(v);
 adj[v].push_back(u);
 ind[u]++;
 ind[v]++;
 }
 int root = 1;
 queue<int> q;
 q.push(root);
 while (!q.empty()){
 int cur = q.front();
 q.pop();
 for (int i = 0; i < ind[cur]; i++){
 int next = adj[cur][i];
 if (pa[next][0] == -1){
 pa[next][0] = cur;
 chainNo++;
 chainHead[chainNo] = next;
 chainInd[chainNo] = 1;
 chainSz[chainNo] = 1;
 } else {
 int prevChainNo = pa[next][0];
 if (chainHead[prevChainNo] != next){
 chainHead[prevChainNo] = next;
 chainInd[prevChainNo] = chainSz[prevChainNo] + 1;
 chainSz[prevChainNo]++;
 }
 }
 pa[next][1] = chainNo;
 pa[next][2] = chainInd[chainNo];
 pa[next][3] = chainSz[chainNo];
 q.push(next);
 }
 }
 for (int i = 1; i <= n; i++)
 cout << query_tree(1, 1, N, i, i) << endl;
 }
}

```

```

 }

 int mid = (l + r) >> 1;
 return (query_tree(2 * node, l, mid, i, j) + query_tree(2 * node + 1, mid + 1,
 r, i, j));
}

int query_up(int u, int v){
 if (u == v) return 0;
 int uchain, vchain = chainInd[v], ans = 0; //uchain and vchain contains the
 chain number
 while (1){
 uchain = chainInd[u];
 if (uchain == vchain){
 if (u == v) break;
 ans += query_tree(1, 0, sz, posInBase[v] + 1,
posInBase[u]); //please note that we are doing +1 because arr[v] contains the
edge length

 //between node 'v-1' and 'v'. So by doing
arr[posInBase[v] + 1] we get the
//first edge between v and v + 1 and so forth and so on upto
 break;
 }
 ans += query_tree(1, 0, sz, posInBase[chainHead[uchain]] - posInBase[u]);
 u = chainHead[uchain];
 u = pa[u][0];
 }
 return ans;
}

int LCA(int u, int v){
 if (depth[u] < depth[v]) swap(u, v);
 int diff = depth[u] - depth[v];
 for (int i = 0; i < LN; i++){
 if ((diff >> i) & 1){
 u = pa[u][i];
 }
 }
}

```

```

 }
}

if (u == v) return u;
//Now u and v are at the same level
for (int i = LN - 1; i >= 0; i--){
 if (pa[u][i] != pa[v][i]){
 u = pa[u][i];
 v = pa[v][i];
 }
}
return pa[u][0];
}

int query(int u, int v){
 int lca = LCA(u, v);
 int ans = query_up(u, lca);
 ans += query_up(v, lca);
 //cout << "query: " << u << " " << v << " " << lca << endl;
 return ans;
}

void HLD(int node, int prev, int cost){
 //cout << "head : " << chainNo << endl;
 chainHead[chainNo] = node;
 endl;
 if (chainHead[chainNo] == -1){
 chainHead[chainNo] = node;
 }
 chainInd[node] = chainNo;
 posInBase[node] = sz;
 arr[sz++] = cost;

 int sc = -1, ncost;
 for (int i = 0; i < adj[node].size(); i++){
 if (adj[node][i] != prev){
 if (sc == -1 || subsz[sc] < subsz[adj[node][i]]){
 sc = adj[node][i];
 ncost = costs[node][i];
 }
 }
 }
}

```

```

)
 }
}

if (sc != -1){
 HLD(sc, node, ncost);
}

for (int i = 0; i < adj[node].size(); i++){
 if (adj[node][i] != prev && adj[node][i] != sc){chainNo++;
 HLD(adj[node][i], node, costs[node][i]);
 }
}
return;
}

void dfs(int node, int prev, int level){
 pa[node][0] = prev;
 depth[node] = level;
 subsz[node] = 1;
 for (int i = 0; i < adj[node].size(); i++){
 if (adj[node][i] != prev){
 otherEnd[ind[node][i]] = adj[node][i];
 dfs(adj[node][i], node, level + 1);
 subsz[node] += subsz[adj[node][i]];
 }
 }
}

int getkth(int p, int q, int k){
 int lca = LCA(p, q), d;
 if (lca == p){
 d = depth[q] - depth[p] + 1;
 swap(p, q); //we want p to be at higher depth...
 so swap p and q if p is at lower depth i.e. it will now become total
 k = d - k + 1; //decide 'k' according to which was originally q)
 distance minus k as we have now changed
 }
}

```

```

else if (q == lca); //do nothing if q is lca
//case when neither p and q are lca
else{
 d = depth[p] + depth[q] - 2 * depth[lca] + 1;
 /*
 d denotes the total dist between the nodes p and q. it will be =
 dist(p,lca) + dist(lca,q) - 1
 = (depth[p] - depth[lca] + 1) + (depth[q] - depth [lca] + 1) - 1
 = depth[p] + depth[q] - 2 * depth[lca] + 1
 */

 if (k > depth[p] - depth[lca] + 1){//case when 'k' will be between lca
 and q i.e. dist b/w lca and 'p' is less than k
 k = d - k + 1;//change 'k' accordingly
 swap(p, q);//swap p and q as we want to calculate the dist from 'p'
 only.
 }
}

//Now we have set starting node as 'p' and 'q' accordingly such that the
//kth node between 'p'
//and 'q' will always lie between 'p' and 'q' at dist 'k' from p
//Also dist(p,lca) > k
//cout << "p : " << p << " q : " << q << " lca : " << lca
//<< endl;
k--;//decrement k as k = 1 will indicate
for (int i = LN - 1; i >= 0; i--){
 if ((1 << i) <= k){//if k is greater than or equal to 2^i than we can
 move up by that much nodes
 p = pa[p][i];//p will become 2^i th ancestor
 k -= (1 << i);//we will move 2^i nodes up and k will be decreased
 by that amount
 }
}
return p;
}

```

## Heavy Light Decomposition

---

```
int main(){
 int t, a, b, c, x, y;
 scanf("%d", &t);
 while (t--){
 scanf("%d", &n);
 for (int i = 0; i < n; i++){
 ind[i].clear();
 adj[i].clear();
 costs[i].clear();
 chainHead[i] = -1;
 for (int j = 0; j < LN; j++) pa[i][j] = -1;
 }
 for (int i = 1; i < n; i++){
 scanf("%d %d %d", &a, &b, &c);
 -a; -b;
 adj[a].push_back(b);
 adj[b].push_back(a);
 costs[a].push_back(c);
 costs[b].push_back(c);
 ind[a].push_back(i - 1);
 ind[b].push_back(i - 1);
 }
 //cout << "chainHead : "; for (int i = 0; i < n; i+) chainHead[i]
 << " "; cout << endl;
 chainNo = 0;
 dfs(0, -1, 0);
 HLD(0, -1, -1);
 //cout << "arr : "; for (int i = 0; i <= sz; i++) cout << arr[i] << " "; cout
 << endl;
 build(1, 0, sz);
 for (int j = 1; j < LN; j++){
 for (int i = 0; i < n; i++){
 if (pa[i][j - 1] != -1) pa[i][j] = pa[pa[i][j - 1]][j - 1];
 }
 }
 }
}
```

```

//cout << "sz : " << sz << endl;
//for (int i = 0; i <= chainNo; i++) cout << "chain : " << i << " " <<
chainHead[i] << endl;
char ch[20];
while (1){
 scanf("%s", ch);
 if (ch[1] == '0') break;
 if (ch[0] == 'D'){
 scanf("%d %d", &a, &b);
 printf("%d\n", query(a-1, b-1));
 }
 else if (ch[0] == 'K'){
 scanf("%d %d %d", &a, &b, &c);
 printf("%d\n", getkth(a - 1, b - 1, c) + 1);
 }
}
return 0;
}

```

**Important Resources**

I would suggest all the readers to go through anudeep's blog post on HLD which can be found [here](http://blog.anudeep2011.com/heavy-light-decomposition/). I have not taken any screenshots from there:

<https://blog.anudeep2011.com/heavy-light-decomposition/>

**Problems to try**

- SPOJ – QTREE – <http://www.spoj.com/problems/QTREE/>
- SPOJ – QTREE2 – <http://www.spoj.com/problems/QTREE2/>
- SPOJ – QTREE3 – <http://www.spoj.com/problems/QTREE3/>
- SPOJ – QTREE4 – <http://www.spoj.com/problems/QTREE4/>
- SPOJ – QTREE5 – <http://www.spoj.com/problems/QTREE5/>
- SPOJ – COT – <http://www.spoj.com/problems/COT/>
- SPOJ – COT2 – <http://www.spoj.com/problems/COT2/>
- SPOJ – COT3 – <http://www.spoj.com/problems/COT3/>
- SPOJ – GOT – <http://www.spoj.com/problems/GOT/>
- SPOJ – GRASSPLA – <http://www.spoj.com/problems/GRASSPLA/>

## Heavy Light Decomposition

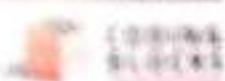
SPOJ – GSS7 – <http://www.spoj.com/problems/GSS7/>  
CODECHEF – RRTREE – <http://www.codechef.com/problems/RRTREE>  
CODECHEF – QUERY – <http://www.codechef.com/problems/QUERY>  
CODECHEF – QTREE – <http://www.codechef.com/problems/QTREE>  
CODECHEF – DGCD – <http://www.codechef.com/problems/DGCD>  
CODECHEF – MONOPLOY –  
<http://www.codechef.com/problems/MONOPLOY>

SELF STUDY NOTES

Während Sie Ihre Übungen machen

SELBSTDICHTUNG

100



## Profiles of Contributors

### Prateek Narang

Fondly referred to as *Prateek Bhaiya*, Prateek Narang is one of the founding members of Coding Blocks. Having completed his Computer Engineering from DTU with exceptional grades, he is now pursuing a Masters in Machine Learning from IIT Delhi. He is the lead mentor for Competitive Programming and Data Structures-Algorithms at Coding Blocks - for both classroom and online courses. His interactive CV ([www.prateeknarang.com](http://www.prateeknarang.com)) is extremely popular in the tech world internationally.

### Rajkishor Ranjan

An IIT-Patna graduate in Computer Science, Rajkishor is a Mathematics wizard and an avid competitive coder. He loves to create and solve challenging problems, and has an immense passion for teaching. Dynamic Programming, recursion, backtracking and graph algorithms are some of his favorite topics.

### Vishal Panwar

Vishal is a DTU graduate who is currently working as a Software Engineer at Microsoft. An experienced competitive programmer, he has a strong hold on various dynamic programming algorithms. He has mentored in multiple batches of Competitive Programming at Coding Blocks.

### Aditya Agrawal

Aditya is a CS undergrad from DTU, who loves competitive coding and has won many awards in various topics. He is an exceptional competitive coder and his team finished eighth at the ACM-ICPC Gwalior Regionals and fifteenth at the ACM-ICPC Gwalior Regionals. He also boasts of a 6-star rating on Codechef and DIV1 on Codeforces.

### Mayank Ladia

Mayank is a skilled competitive programmer and an enthusiastic software developer at IIIT. He has a comprehensive knowledge of advanced topics like graph algorithms and dynamic programming, which led to his team finishing twenty fifth at the ACM-ICPC. He has researched and developed various optimization algorithms which are at par with international giants such as Routific. He has mentored in multiple batches of Competitive Programming at Coding Blocks.

### Saransh Gupta

A graduate from IIIT-Delhi, Saransh is currently working as a software developer at Amazon. His joy lies in coding and building great usable products. Owing to his expertise in the topics, he has taught Number Theory and Game Theory in multiple Competitive Programming Bootcamps by Coding Blocks.

### **Shubham Rawat**

A DTU graduate, Shubham is an expert programmer and an exceptional mentor, currently working as a software developer at Works Applications in Tokyo, Japan. He has contributed to topics like segment trees, BIT and dynamic programming in this module. He has mentored in multiple batches of Competitive Programming at Coding Blocks.

### **Hardik Agrawal**

Hardik is an ace competitive programmer, currently pursuing his engineering in Information Technology from DTU. He has a vast experience in competitive coding and qualified for ACM-ICPC Amritapuri in 2016 and ACM-ICPC Chennai in 2017. He has solved more than 300 problems on various platforms and has an expert level on Codeforces. In his final year at the moment, he has already received job offers from Microsoft and Flipkart.

We are also thankful to **Mr. Priyanshu Agrawal**, one of the founding members of Coding Blocks, for helping the team with the entire process. Our heartfelt also to **Mr. Siddharth Jain**, a designer at Coding Blocks, for the front and back cover design as well as to **Mr. Sheel Kaushik** for layout design.

We hope you enjoy gaining expertise in competitive programming through this module. We have tried our best to keep it error free, but some errors might have crept in. In case you find any such errors, please report them at <https://cb.lk/Error-CP> and we shall make the necessary changes in the next iteration of this module. If you have any other suggestions or just want to reach out to us, feel free to drop us an email at [info@codingblocks.com](mailto:info@codingblocks.com).