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 posses 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 posses 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!
      ++beginOfContaInter;
  }
   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.

1. **Algorithms :** There are inbuilt algorithms for tasks like sorting,searching etc. We will discuss these in the later part of the chapter.
2. **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**). It's **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 wit h
                         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). It's time complexity is O(1).
- **back() :** Returns the reference to the last element. It's time complexity is O(1).
- **begin() :** Returns an iterator pointing to the first element of the
- **Vector :** It's time complexity is O(1).
- **clear() :** Deletes all the elements from the vector and assign an empty vector. It's 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. ts 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. It's time complexity is O(1).
- **push_back() :** Inserts a new element at the end of the vector. It's time complexity is O(1).
- **resize() :** Resizes the vector to the new length which can be less than or greater than the current length. It's time complexity is O(N) where N is the size of the resized vector.
- **size() :** Returns the number of elements in the vector. It's 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 element s 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.It's time complexity is O(1).
- **back( ) :** It returns reference to the last element in the list.It's time complexity is O(1).
- **assign( ) :** It assigns new elements to the list by replacing its current elements and change it's size accordingly.It time complexity is O(N).
- **erase( ) :** It removes a single element or the range of element from the list. It's time complexity is O(N).

- **front( ) :** It returns reference to the first element in the list.It's time complexity is O(1).
- **push_back( ) :** It adds a new element at the end of the list, after its current last element. It's time complexity is O(1).
- **push_front( ) :** It adds a new element at the beginning of the list, before its current first element. It's 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. It's time complexity is O(1).
- **insert( ) :** It insert new elements in the list before the element on the specified position. It's time complexity is O(N).
- **reverse ( ) :** It reverses the order of elements in the list. It's time complexity is O(N).
- **size( ) :** It returns the number of elements in the list. It's time complexity is O(1).

Pair

Pair is a container that can be used to bind together a 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 look ups. 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. It's 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 be 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(logN).
- **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 if found otherwise return 0.Its time complexity is O(logN).
- **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. It's 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(logN).
- **insert( ) :** Insert a single element or the range of element in the map.Its time complexity is O(logN), 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) + rebalance	Same as search
Deletion time	log(n) + 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.

  load_factor = current_size / 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[]){
```

```
    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 :

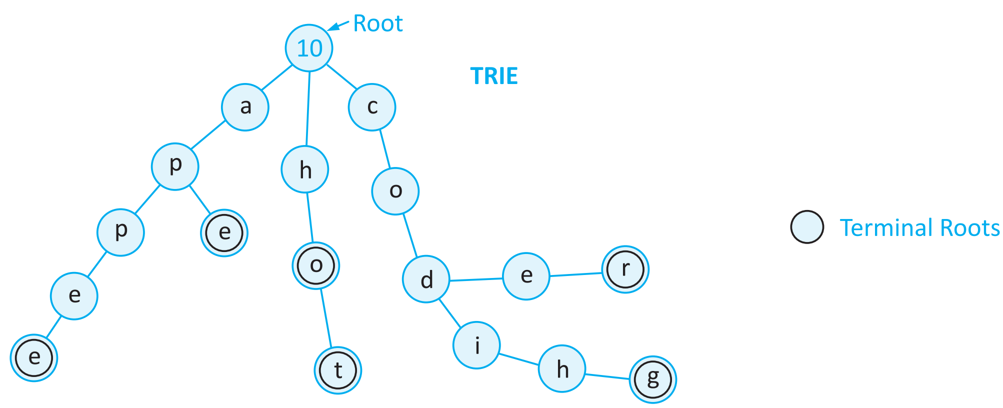
```
#include<iostream>
#include<cstdio>
#include<deque>
using namespace std;
int a[1000001];
int main(){
int n,k,i;
scanf("%d",&n);
for(i=0;i<n;i++)
  {
      scanf("%d",&a[i]);
  }
scanf("%d",&k);
deque<int> Q(k);
for(i=0;i<k;i++)
  {
   while(!Q.empty()&&a[i]>=a[Q.back()])
  Q.pop_back();
   Q.push_back(i);
  }
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("%d\n",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, the a particular string can be search 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 in marked with is Terminal = true.



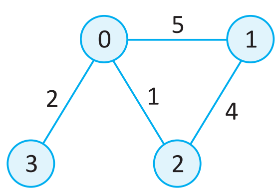

```
#include<iostream>
#include<unordered_map>
using namespace std;
#define hashmap unordered_map<char,node*>
class node{
public:
  char data;
  hashmap h;
   bool isTerminal;
  node(char d){
     data = d;
      isTerminal = false;
  }
};
class Trie{
```

```
  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.



$\Longleftrightarrow$

List <pair<int,int>>

$$\begin{bmatrix} 0 & \rightarrow & (1, 5)\ (2, 1)\ (3, 0) \\ 1 & \rightarrow & (0, 5)\ (2, 4) \\ 2 & \rightarrow & (0, 1)\ (1, 4) \\ 3 & \rightarrow & (0, 2) \end{bmatrix}$$

Adj List

```
#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;
```

```
    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

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 ?

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 – (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++;
```

```
            cout<<"Probability is "<<p<<" and people are "<<people<<endl;
      }
      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 $S = f1; 2; 3; 5; 8; 13; 21; 34$ 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 Big Integer Class

In Java, the Big Integer 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");
```


CODING BLOCKS
Code Your Way To Success

```
        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 he 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 could'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) I 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 $^{n}C_{k}$ denotes the number of ways of selecting k items from n items.

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

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 it its left half and right-half. Do it for every value of i (1<=i<=n) and sum it up.

The formula generating after adding the series is the nth Catalan Number !!

It is defined using binomial coefficient notation ${}^{n}C_{k}$ as :

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

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

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

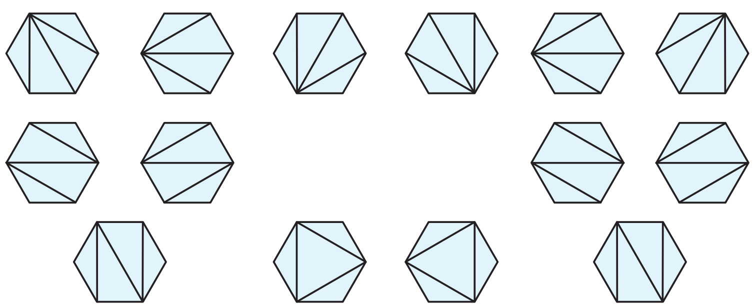
1, 1, 2, 5, 14, 42, 132, 429, 1430

Another Recursive Formula is :

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

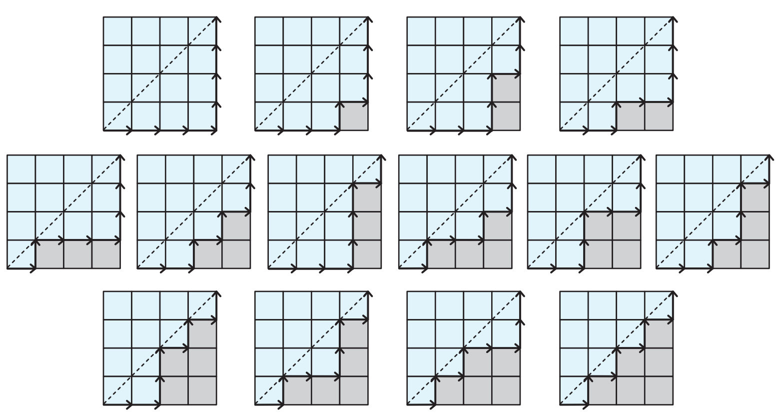
Applications of Catalan Numbers :

- 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.



- Number of ways to form a "mountain ranges" with n upstrokes and n down-strokes that all stay above the original line. The mountain range interpretation is that the mountains will never go below the horizon

n = 0	*	1 way
n = 1	/\	1 way
n = 2	/\ /\/\, / \	2 ways
n = 3	/\ /\ /\ /\/\ / \ /\/\/\, /\/ \, / \/\, / \, / \	5 ways

Mountain Ranges

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 F1 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 Fi as a K x 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 F1 that has values f(1) through f(K):

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

$TF_i = F_i + 1$

Suppose,

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

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

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 F(n)

$$F_2 = TF_1$$
$$F_3 = TF_2 = T^2F_1$$
$$F_n = T^{n-1}F_1$$

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

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

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:

$a_i = b_i$ **(for i <= k)**

$a_i = c_1a_{i-1} + c_2a_{i-2} + \ldots + c_ka_{i-k}$ **(for i > k)**

where b_j and c_j are given natural numbers for 1 <= j <= 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) x 1 now :

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

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

$TF_i = F_{i+1}$

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

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

1. Generate the Transformation matrix for the given sequence.

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

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

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

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

3. **Fibonacci Number (HackerBlocks) :**

   Write an efficient code to compute nth Fibonacci Number where N<=10^18.

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

   Read the problem statement at Spoj.

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

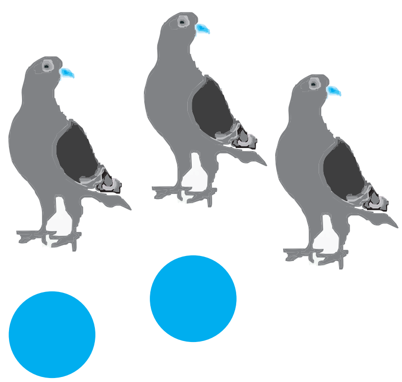
5. Fast Ladders (HackerBlocks) :

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.

6. Fast Tiling Problem (HackerBlocks) :

Given n, and grid of size 4 x n, you have to find out the number of ways of filling the grid using 1 x 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 $a_1 + a_2 + \ldots + a_k$ by $\mathbf{b_k}$. So, we obtain:

$b_0 = 0$

$b_1 = a_1$

$b_2 = a_1 + a_2$

.

. . .

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

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

Therefore, if there are two values with equal residues modulo N among b_0, b_1, …, 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 bi 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. $\{X_1, X_2\}$, $\{X_3, X_4\}$, $\{X_5, X_6\}$ … $\{X_{129}, X_{130}\}$. 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 of pair of these four numbers, we will get 0.

Thus, by pigeonhole principle for all n >= 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] ^ A[j] ^ 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]){
```

```
                        count[node] += dfsHelper(v,visited,count,ans);
                        ans += 2*min(count[v],V-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 Ai, the

EXAMPLES :

1. Prime Looking Numbers - HackerBlocks :

How many number 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 floor((N-1)/m).

Solution :

divisible by 2 = floor(999/2) = 449 ; divisible by 3 = floor(999/3) = 333

divisible by 5 = floor(999/5) = 199 ; divisible by 2.3 = floor(999/6) = 166

divisible by 2.5 = floor(999/10) = 99

divisible by 3.5 = floor(999/15) = 66

divisible by 2.3.5 = floor(999/30) = 33

$|2 \cup 3 \cup 5| = |2| + |3| + |5| - |2 \cap 3| - |2 \cap 5| - |3 \cap 5| + |2 \cap 3 \cap 5|$

$= 499 + 333 + 199 - 166 - 99 - 66 + 33 = 733$

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 of numbers which are divisible by 2,3 and 5
        int a[] = {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],…,A[N] such that A[i] >=1 and A[i] <= M and LCM(A[1], A[2], …, A[N]) is divisible by D. We have to find the sum of the answers with D = L, L+1, …,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, ans = (possible array with m 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 = |not(P) U not(Q) U not(R) U not(S)|

Applying Principle of Inclusion-Exclusion Principle :

A = power(m - m/P,n) + power(m - m/Q,n) + power(m - m/R,n) + power(m - m/S,n);

B = power(m - m/P - m/Q + m/(P*Q), n) + power(m - m/Q - m/R + m/(Q*R), n) . . .

C = power(m- m/P -m/Q - m/R + m/(P*Q) + m/(Q*R) + m/(P*R) - m/(P*Q*R),n) . . .

D = 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

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 $\sum x_iP(x_i)$ the summation runs over all the distinct values xi 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.

For a continuous variable X with probability density function P(x) , the expected value E(X) is given by $\int xP(x)dx$.

- Mathematical expectation is some sort of average value of your 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 exepected 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) + ... + (1/(2^k))(x+k) + .. + (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 number 0.16 has been replaced by p. Observe that 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 the final answer does not depend on K. The answer is thus 1/(1/6) i.e. 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 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 * 2/3 * 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 * 2/3 * 3/4 * ... * (k-1)/k * 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 probabilty of each candidate depends on it's 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 :

x = 1/(1*2) + 2/(2*3) + 3/(3*4) + 4/(4*5) + ... + k/(k*(k+1)) + ...

= 1/2 + 1/3 + 1/4 + ...

This is a well-known divergent series, which means that sum doesnot converge, and hence the expectation doesnot exist.

8. A random permutation P of [1...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 = $\{a_1, a_2, \ldots, a_n\}$ 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 ai = if the i'th experiment gave head then 1 else 0.

Hence we have: Number of heads in z = $a_1 + a_2 + \ldots + a_n$

Hence E[number of heads in z] = $E[a_1 + a_2 + \ldots + a_n] = E[a_1] + E[a_2] + \ldots + E[a_n]$

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

10. (Bernaulli 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 **bernaulli trials**. An experiment is called a bernaulli 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 bernaulli 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 bernaulli 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 bernaulli trials - If the probability of a success in a bernaulli trial is p then the expected number of successes in n trials is n*p. The proof is simple :

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

The expected value of each bracket is 1*p + 0*(1-p) = p. Thus the expected number of successes in n trials is n*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*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 * 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,..., 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 bernaulli 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 bernaulli 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 Xi be the **number of boxes it takes for the coupon collector to collect the i-th new coupon after the i-1th 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 Xi boxes, the coupon collector would have collected i distinct coupons, but with only Xi-1 boxes, the coupon

collector would have only collected i-1distinct 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.

This is a bernaulli trial with probability of success p and failure (1-p). In bernaulli trial, the expected number of trials for i-th success is 1/p i.e. 1/(success of the i-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:

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

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

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

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) + ... + F(M)) mod 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 $a_1, a_2, \ldots, a_n$, and a number m.

Check if it is possible to choose a non-empty subsequence a_{ij} 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

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 gcd(a,b), is defined as the largest positive integer d such that d | a and d | b where x | y implies that x divides y.

Example of GCD: gcd(4, 8) = 4, gcd(10, 5) = 5, gcd(20,12) = 4.

```
Gcd(A,B) = Gcd(B,A%B) // recurrence for gcd
Gcd(A,0) = A // base case
```

Proof : If a = bq + r.

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

Let e be any common divisor of b and r $\Rightarrow$ e|b , e|r $\Rightarrow$ e|(bq + r) $\Rightarrow$ e|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 lcm(a,b), is defined as the smallest positive integer l such that a |l and b | l. Example of LCM : lcm(4,8) = 8, lcm(10,5) = 10, lcm(20,12) = 60.

gcd(a,b) * 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. gcd(a,b,c) is equal to gcd(a,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. Ax + By = 1 (we will find solutions of this equation let them be x' and y' given that gcd(a,b) is 1 then the solutions of equation Ax + By = k where k is a multiple of gcd(A,B) are given by k*x' and k*y'.

Ax + By = 1 . . . .(1)

Bx' + (A%B)y' = 1 . . . .(2) // using euclid's algo gcd(a,b) = 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 :

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

Let's see :

$$x \equiv a^{-1} (\text{mod } m)$$

$$a \,.\, a^{-1} \equiv a \,.\, x (\text{mod } m)$$

$$x \,.\, a \equiv 1 (\text{mod } m)$$

$\Rightarrow$ $ax - 1 = qm$

$\Rightarrow$ $ax - qm = 1$

This equation has solutions only if a and m are co-prime that is 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 0.5sqrt(n) steps*. That means it takes 50,000 steps to check that 10,000,000,000 is a prime.

Time Complexiety :

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

Can we do better?

Ofcourse! we can use a sieve of numbers upto N. For all prime numbers <= sqrt(N), we can make their multiple non-prime i.e. if p is prime, 2p, 3p, ..., floor(n/p)*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 10000000
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) //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<=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;
```

```
   }
 }
//  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) = (xo + b/d *t, yo — a/d *t)

We can use *Extended Euclidean Method* above to find the xo,yo.

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:

$$x \equiv 1 \ (\text{mod } 2), \quad x \equiv 2 \text{ mod}(3), \quad x \equiv 5 \text{ mod } (7).$$

x % num[0] = rem[0],

x % num[1] = rem[1],

....

x % num[k-1] = 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

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

```
rem[i] is given array of remainders
prod is product of all given numbers
prod = num[0] * num[1] * ... * num[k-1]

pp[i] is product of all but num[i]
pp[i] = prod / 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 , φ(8) = 4, φ(9) = 6

The value φ(n) can be obtained by Euler's formula :

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

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

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 $d_1, d_2, \ldots d_k$ be all divisors of n (including n). Then $\varphi(d1) + \varphi(d2) + \ldots + \varphi(dk) = n$

   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 $n = p_1^{a_1} * p_2^{a_2} * \ldots\ldots * p_n^{a_n}$:

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

5. Sum of divisors:

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

Wilson's Theorem

If p is a prime, then (p — 1)! = –1 (mod p)

Problem : DCEPC11B (SPOJ)

Hint :

This can be solved using Wisdom theorem

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

   $$(p-1)! \equiv -1 (\text{mod } p)$$

   $1*2*3* \ldots\ldots *(n-1)*(n)* \ldots\ldots *(p-1) \equiv -1 \pmod p$

   $n!\,(n+1)* \ldots\ldots *(p-1) \equiv -1 \pmod p$

   $n! \equiv -1*[(n+1)* \ldots\ldots (p-2)*(p-1)\text{^} -1 \pmod p$

Lucas Theorem

In number theory, Lucas's theorem expresses the remainder of division of the binomial coefficient ${}^{m}C_{n}$ 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 $m = m_k p^k + m_{k-1} p^{k-1} + \ldots + m_1 p + m_0$ and $n = n_k p^k + n_{k-1} p^{k-1} + \ldots + n_1 p + n_0$

Problem : Compute ${}^{n}C_{r}$ % p.

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

Using Lucas Theorem ${}^{n}C_{r}$ % p

Lucas theorem basically suggests that the value of ${}^{n}C_{r}$ can be computed by multiplying results of ${}^{n_i}C_{r_i}$ 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 ${}^{n_i}C_{r_i}$ 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 " 2 = 7 × 18 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

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

For example, if a = 2 and p = 7 then 2^6 = 64 and 64 " 1 = 63 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 power B) % (10^9 + 7) where A, B<=10^100000

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

$A = [a * (mod) + b]$ (where mod = $10^9 + 7$, a and b are integers). This implies that

$(A \uparrow n) \% mod = [(a * (mod) + b) \uparrow n]\%mod = \{[(a * (mod) + b) \uparrow \% mod] * [(a * (mod) + b) \uparrow \%mod] *$

$\ldots\ldots\ldots$ n times $\% mod = b \uparrow n$ (where b is nothing but A % mod)

Using modulo properties :

(a∗ b) %m = ((a%m)% (a + b)%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 it's 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

$x^{p-1}\%p = 1$ (where p 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))

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 found in the similar way as A % (10^9 + 7)

Finally

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(logn).

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 probabily prime

write n – 1 as 2^r.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]

$x \leftarrow a^d$ mod n

if x = 1 or y = n – 1 then

continue WitnessLoop

repeat r – 1 times:

$x \leftarrow x^2 \bmod n$

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*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 = pow(a, d) % n
   x = 43 % 13 = 12
3. Since x = (n – 1), return prime.

IInd Iteration:

1. Pick a random number 'a' in range [2 + n – 2]
   Suppose a = 5
2. Compute: x = pow(a, d) % n
   x = 53 % 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, 1d ≤ a,b,nd ≤ 10^{5}

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

Dealing with f(n) :

The function f complicates the expression, but we can 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 $a^{\varphi(m)} = 1(\text{mod } m)$, where φ(m) is Euler's totient function. This is useful because $a^y \equiv a^{(y \text{ mod } \varphi(m))}(\text{mod } m)$.

The repeated φ(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 $a^{y \text{ mod } 100000006} \text{mod} 1000000007$.

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 $b^{(2n,\ n)}$mod 1000000006.

Finding $b^{(2n,\ n)}$mod 1000000006 when b is odd

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

$$b^{(2n,\,n)} \equiv b^{(2n,\,n) \text{mod} \varphi\,(1000000006)} \text{mod} 1000000006$$

$$\varphi(1000000006) = \varphi(2) \times (500000003) = (2-1) \times (500000003-1) = 500000002)$$

$$500000002 = 2 \times 41^2 \times 148721500000002 = 2 \times 41^2 \times 148721$$

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

(2n, n)mod 2

(2n, n)mod 41^2

(2n, n)mod 148721

and use CRT to get the result modulo 500000002.

Finding $b^{(2n,\ n)}$mod 1000000006 $b^{(2n,\ n)}$ 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 and 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 φ(500000003)= φ(1000000006), so this part is equal to the case when b is odd. The only difference is using CRT.

Adding everything together

After finding y = $b^{(2n, n)}$ mod 1000000006, we can calculate a y mod 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 md = 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[i] %= (tmd);
  }
 }
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 < 200003; ++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 <=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)) = 1 (mod m)
a^(-1) = a^(m-2) (mod 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 :

1. **Find sum of divisors of all the numbers from 1 to n . n <= 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 <= 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 p1^k1 * p2^k2...

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

Code :

```
/*input
3
2
10
10000000000000000
*/
#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 3.141592653589793238462643383279502 88
#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*... 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 = p1*p2 where p1 = 186583 p2 = 587117

Let a = 1! * 2! * 3*... 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 3.141592653589793238462643383279502884
#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
        ll ans_p1 = 1ll , ans_p2=1ll , curfactorial_p1 = 1ll , curfactorial_p2 = 1ll;
        F(i,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]
    ll inv_zero = power(p2,p1-2ll,p1); // fermats
    ll inv_first = power(p1,p2-2ll,p2); // fermats
    ll 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 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

SELF STUDY NOTES