STL Function & Descriptions:

STL Functions :

push_back(value) [add value to the end]
push_fron(value) [add value to the first]
emplace_back(val) [add value to the end]
push(val) [add val to container]
pop() [delete a value from cont]
pop_back() [delete last value]
Pop_front () [delete first value]
empty() [if empty then true]
size() [number of total value stored]
clear() [clear the container]
resize(n) [resize the container with size n]

front() [return last value / top value]
back() [returns last value]
count(val) [returns the number of matches to element 'val']

at(key) [function is used to return the reference to the element associated with the key key]

insert(value) 1. [insert the value]
insert(iterator , val) 2. [insert value to the iterator position]

num = erase(val) 1. [erase all data matches with val, and return no of values erased]
erase(iterator) 2. [erase only data of the iterator]
erase(it1 , it2) 3. [erase all data inclusive the iterator [it1 , it2]]

find(value) [returns the iterator of first occurrence of value if not found then equals to end()][if type is pair then search base on the first element]

assign(size,val) 1. [assign 'val' upto size index]
assign(arr, arr+size) 2. [assign size no of value arr to a container]

cn1.swap(cn2) [swap two container O(1)]
reserve(start,end) [reverse start to end iterator]
sort(start,end) [sort start to end iterator]

STL Iterrators :

container_name<.....> ::iterator it [Declaration]
cout << *it << "\n"; [printing is like pointer]

begin() [pointing to the first element]
end() [pointing to end of container | after last val]
rbegin() [returns a reverse iterator pointing to the last element in container (reverse beginning). It moves from last to first element]
rend() [returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)]

auto it = s.begin(); [store any iterator without declaration]

Vector

vector<T>vc;
assign(),erase()-(1,2), insert()-(1), begin(), end(), rbegin(), rend(), clear(), push_back(), pop_back(), empty(), size(), swap(), reserve(), resize(), emplace_back()

Map

map<T1,T2>mp;
It uses red-blcak tree to implement so each complexity is log(n).

count(), at(), swap(), insert()-(2), erase(), rend(), rbegin(), find(), emplace(), upper_bound(), lower_bound(), size(), empty(), begin(), end(), clear(), [= assign one map to other], []

Unordered Map

unordered_map<T1, T2> mp;
Use hash table to implement . So time complexity is O(1) for find and insert elements. **N.B.** use it where each number is distinct otherwise time complexity may raise to O(n). good for **string**

Queue

queue<T>q;
empty(), size(), swap(), emplace(), front(), back(), push(g), pop()

Priority_Queue

priority_queue <int> pq; [descending|max heap]
priority_queue <int,vector<int>,greater<int> > pq; [ascending|min heap]

push(), top(), pop(), empty()

Deque

deque<T>dq;
push_back(x); push_front(y); front(); back();
pop_back(x); pop_front(y); clear(); size();
But empty() is no available

Stack

stack<T>st;
push(), top(), pop(), empty()

Set

set<T>s;
count(), at(), swap(), insert()-(2), erase(), rend(), rbegin(), find(), emplace(), upper_bound(), lower_bound(), size(), empty(), begin(), end(), clear(), [= assign one map to other]

Unordered Set

unordered_set <string> s;
Use hash table . good for string.

Multi set

multiset <int, greater <int> > s; [ascending order]
insert(40)-(1,2), erase()-(1,2,3), clear(), find(20), count(const g), rbegin(),swap()

multiset <int> set2(set1.begin(), set1.end());
[assigning the elements from set1 to set2]

cout<<*set1.lower_bound() <<endl;
cout<<*set1.upper_bound() <<endl;

Bitset

- bitset<MX_SIZE> s;
- bitset<10> s(string("0010011010")); // from right to left
- s.count() number of ones
- Any bitwise operation can be perform
- cout<<s<<endl; print all bits
- Think as a bit collection and can be access like vector

Policy Based Data Structure

Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

typedef
tree<T,null_type,less<T>,rb_tree_tag,tree_order_statistics_node_update> ordered_set;
```

[desc][greater for asc]

Ordered Multiset :

typedef
tree<T,null_type,less_equal<T>,rb_tree_tag,tree_order_statistics_node_update> ordered_set;

```
st.insert(val);
auto it = *st.find_by_order(a-1);
printf("%d\n",it.second);
st.erase(it);
```

**order_of_key works same as lower_bound()
**can be used as map using pair
find_by_order() [returns an iterator to the k-th largest element (counting from zero)]
order_of_key() [the number of items in a set that are strictly smaller than our item.]

And other set function works here.

STL upper and Lower Bound

```
// std :: upper_bound with vectors
#include <bits/stdc++.h>
int main(){
        std::vector<int> v{ 10,20, 20, 30, 40, 50 };
        //ub(51)->6  //ub(50)->6  //ub(49)->4
//ub(10)->1 /ub(9) ->0
        //std::upper_bound() returns an iterator to
the upper bound of the value passed to it.
         //iterator to 1st number larger than it.

         //lb(51)->5 //lb(50)->4 //lb(49)->4
//lb(15)->1 //lb(10)->0 //lb(9)->0
         auto upper2 = std::lower_bound(v.begin(),
v.end(), 20);
```

```
        std::cout << "\nupper_bound for element 35 is at position : "
                  << (upper1 - v.begin());
        std::cout << "\nupper_bound for element 45 is at position : "
                  << (upper2 - v.begin());
        return 0;
}
```

```
sort (v.begin(), v.end()); // 5 5 5 7 7
vector<int>::iterator lower,upper;
lower = lower_bound (v.begin(), v.end(), 5);
upper = upper_bound (v.begin(), v.end(), 5);
lower- v.begin()  = 0 // pos: which is first >=val
upper - v.begin() = 3 // position : which is > val
```

Hidden function (Bit functions)

- **__gcd(value1, value2)** // Euclidean GCD
- **__builtin_ffs(x)** This function returns 1 + least significant 1-bit of x. If x == 0, returns 0. Here x is int, this function with suffix 'l' gets a long argument and with suffix 'll' gets a long long argument. **e.g.** __builtin_ffs(10) = 2 because 10 is '...10 **1** 0' in base 2 and first 1-bit from right is at index 1 (0-based) and function returns 1 + index.
- **__builtin_clz(x)** This function returns number of leading 0-bits of x which starts from most significant bit position. x is unsigned int and like previous function this function with suffix 'l gets a unsigned long argument and with suffix 'll' gets a unsigned long long argument. If x == 0, returns an undefined value. **e.g.** __builtin_clz(16) = 27 because 16 is ' **...** 10000'. Number of bits in a unsigned int is 32. so function returns 32 — 5 = 27.
- **__builtin_ctz(x)**This function returns number of trailing 0-bits of x which starts from least significant bit position. x is unsigned int and like previous function this function with suffix 'l' gets a unsigned long argument and with suffix 'll' gets a unsigned long long argument. If x == 0, returns an undefined value. **e.g.** __builtin_ctz(16) = 4 because 16 is '...1 **0000** '. Number of trailing 0-bits is 4.
- **__builtin_popcount(x)** This function returns number of 1-bits of x. x is unsigned int and like previous function this function with suffix 'l' gets a unsigned long argument and with suffix 'll' gets a unsigned long long argument. If x == 0, returns an undefined value. **e.g.** __builtin_popcount(14) = 3 because 14 is '... **111** 0' and has three 1-bits.

[C++ STL: Order of magnitude faster hash tables with Policy Based Data Structures]

The Policy Hash Table has 3-6x faster insertion/deletion and 4-10x increase for writes/reads.

```
#include <ext/pb_ds/assoc_container.hpp>

using namespace __gnu_pbds;

gp_hash_table<int, int> table;
```

[Implicit cartesian tree in GNU C++ STL]

Briefly speaking this data structure can fastly insert arbitrary blocks of array to any position and erase them. So it's very similar to [implicit cartesian tree]. It's used sometimes to handle a very long strings.I used [this] task for testing. Here you should quickly move the block [l,r] to the beginning of the array 10^5 times and besides the size of array is not greater than 10^5:

```
#include <iostream>

#include <cstdio>

#include <ext/rope> //header with rope

using namespace std;

using namespace __gnu_cxx; //namespace with rope and some additional stuff

int main(){

        ios_base::sync_with_stdio(false);

        rope <int> v; //use as usual STL container

        int n, m;
```

```
	cin >> n >> m;

	for(int i = 1; i <= n; ++i) v.push_back(i);
//initialization

	int l, r;

	for(int i = 0; i < m; ++i){

		cin >> l >> r;

		--l, --r;

		rope <int> cur = v.substr(l, r - l + 1);

		v.erase(l, r - l + 1);

		v.insert(v.mutable_begin(), cur);

	}

	for(rope <int>::iterator it =
v.mutable_begin(); it != v.mutable_end(); ++it)

	cout << *it << " ";

	return 0;

}
```

Policy based data structures

```
#include <ext/pb_ds/assoc_container.hpp> //
Common file

#include <ext/pb_ds/tree_policy.hpp> // Including
tree_order_statistics_node_update
```

After closer inspection you may find that the last two files contained in the library

```
#include <ext/pb_ds/detail/standard_policies.hpp>

typedef tree<

int,

null_type,

less<int>,

rb_tree_tag,

tree_order_statistics_node_update>
```

Policy Based Priority Queue

```
#include <ext/pb_ds/priority_queue.hpp>

priority_queue<int, less<int>, pairing_heap_tag>
r_c;

r_c.push(1);

for (size_t i = 0; i < 10; ++i)

r_c.push(i);

cout << endl << "All values in the container:" <<
endl;

typedef typename Cntnr::const_iterator
const_iterator;

for (const_iterator it = r_c.begin(); it != r_c.end();
++it)

cout <<* it << endl;

while (!r_c.empty())

{

	cout << r_c.top() << endl;

	r_c.pop();

}
```

Policy Based Trie

```
#include <cassert>

#include <iostream>

#include <ext/pb_ds/assoc_container.hpp>

#include <ext/pb_ds/trie_policy.hpp>
```

```
#include <ext/pb_ds/tag_and_trait.hpp>

using namespace std;

using namespace pb_ds;

using namespace pb_ds;

// A PATRICIA trie with a prefix-search node-updator type. Note that

// since the node updator is trie_prefix_search_node_update, then the

// container includes its method prefix_range.

typedef null_mapped_type mapped_type;

typedef string_trie_e_access_traits<> cmp_fn;

typedef pat_trie_tag tag_type;

typedef trie<string, mapped_type, cmp_fn, tag_type,

trie_prefix_search_node_update> trie_type;

// The following helper function takes a trie object and r_key, a

// const reference to a string, and prints all entries whose key

// includes r_key as a prefix.

void

print_prefix_match(const trie_type& t, const string& key)

{

typedef trie_type::const_iterator const_iterator;

typedef pair<const_iterator, const_iterator> pair_type;

cout << "All keys whose prefix matches \"" << key << "\":" << endl;

const pair_type match_range = t.prefix_range(key);

for (const_iterator it = match_range.first; it != match_range.second; ++it)

cout << *it << ' ';

cout << endl;

}

int main()

{

trie_type t;

// Insert some entries.

assert(t.insert("I").second == true);

assert(t.insert("wish").second == true);

assert(t.insert("that").second == true);

assert(t.insert("I").second == false);

assert(t.insert("could").second == true);

assert(t.insert("ever").second == true);

assert(t.insert("see").second == true);

assert(t.insert("a").second == true);

assert(t.insert("poem").second == true);
```

```
assert(t.insert("lovely").second == true);

assert(t.insert("as").second == true);

assert(t.insert("a").second == false);

assert(t.insert("trie").second == true);


// Now search for prefixes.

print_prefix_match(t, "");

print_prefix_match(t, "a");

print_prefix_match(t, "as");

print_prefix_match(t, "ad");

print_prefix_match(t, "t");

print_prefix_match(t, "tr");

print_prefix_match(t, "trie");

print_prefix_match(t, "zzz");


return 0;

}
```

Raw Strings

You can have UTF-8 strings, Raw strings and more. Here I want to show raw strings. We define a raw string as below:

```
string s = R"(Hello, World!)"; // Stored: "Hello, World!"
```

A raw string skips all escape characters like \n or \". e.g.

```
string str = "Hello\tWorld\n";

string r_str = R"(Hello\tWorld\n)";

cout << str << r_str;
```

Output:

```
Hello    World

Hello\tWorld\n
```

You can also have multiple line raw string:

```
string r_str =

R"(Dear Programmers,

I'm using C++11

Regards, Swift!)";

cout << r_str;
```

Output:

```
Dear Programmer,

I'm using C++11

Regards, Swift!
```

two) Regular Expressions (regex)

Regular expressions are useful tools in programming, we can define a regular expression by regex e.g. regex r = "[a-z]+";. We will use raw string for them because sometimes they have \ and other characters. Look at the example:

```
regex
email_pattern(R"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.
[a-zA-Z0-9-.]+$)"); // This email pattern is not
totally correct! It's correct for most emails.



string

valid_email("swift@codeforces.com"),

invalid_email("hello world");



if (regex_match(valid_email, email_pattern))

   cout << valid_email << " is valid\n";

else

   cout << valid_email << " is invalid\n";



if (regex_match(invalid_email, email_pattern))

   cout << invalid_email << " is valid\n";

else

   cout << invalid_email << " is invalid\n";
```

Output:

```
swift@codeforces.com is valid

hello world is invalid
```