Lời nói đầu

- "C++ được đánh giá là ngôn ngữ mạnh vì tính mềm dẻo, gần gũi với ngôn ngữ máy. Ngoài ra, với khả năng lập trình theo mẫu (template), C++ đã khiến ngôn ngữ lập trình trở thành khái quát, không cụ thể và chi tiết như nhiều ngôn ngữ khác. Sức mạnh của C++ đến từ STL, viết tắt của Standard Template Library - một thư viện template cho C++ với những cấu trúc dữ liệu cũng như giải thuật được xây dựng tổng quát mà vẫn tận dụng được hiệu năng và tốc độ của C. Với khái niệm template, những người lập trình đã đề ra khái niệm lập trình khái lược (generic programming), C++ được cung cấp kèm với bộ thư viện chuẩn STL.

  Bộ thư viện này thực hiện toàn bộ các công việc vào ra dữ liệu (iostream), quản lý mảng (vector), thực hiện hầu hết các tính năng của các cấu trúc dữ liệu cơ bản (stack, queue, map, set...). Ngoài ra, STL còn bao gồm các thuật toán cơ bản: tìm min, max, tính tổng, sắp xếp (với nhiều thuật toán khác nhau), thay thế các phần tử, tìm kiếm (tìm kiếm thường và tìm kiếm nhị phân), trộn. Toàn bộ các tính năng nêu trên đều được cung cấp dưới dạng template nên việc lập trình luôn thể hiện tính khái quát hóa cao. Nhờ vậy, STL làm cho ngôn ngữ C++ trở nên trong sáng hơn nhiều."

  Trích **"Tổng quan về thư viện chuẩn STL"**
- STL khá là rộng nên tài liệu này mình chỉ viết để định hướng về cách sử dụng STL cơ bản để các bạn ứng dụng trong việc giải các bài toán tin học đòi hỏi đến cấu trúc dữ liệu và giải thuật.
- Mình chủ yếu sử dụng các ví dụ, cũng như nguồn tài liệu từ trang web www.cplusplus.com , các bạn có thể tham khảo chi tiết ở đó nữa.
- Để có thể hiểu được những gì mình trình bày trong này, các bạn cần có những kiến thức về các cấu trúc dữ liệu, cũng như một số thuật toán như sắp xếp, tìm kiếm...
- Việc sử dụng thành thạo STL sẽ là rất quan trọng nếu các bạn có ý định tham gia các kì thi như Olympic Tin Học, hay ACM. "STL sẽ nối dài khả năng lập trình của các bạn" (trích lời thầy Lê Minh Hoàng).
- Mọi ý kiến đóng góp xin gửi về địa chỉ: manhdjeu@gmail.com

Mục lục

- Thư viện mẫu chuẩn STL trong C++ chia làm 4 thành phần là:
    - Containers Library : chứa các cấu trúc dữ liệu mẫu (template)
        - Sequence containers
            - Vector
            - Deque
            - List
        - Containers adpators
            - Stack
            - Queue
            - Priority_queue
        - Associative containers
            - Set
            - Multiset
            - Map
            - Multimap
            - Bitset
    - Algorithms Library: một số thuật toán để thao tác trên dữ liệu
    - Iterator Library: giống như con trỏ, dùng để truy cập đến các phần tử dữ liệu của container.
    - Numeric library:

- Để sử dụng STL, bạn cần khai báo từ khóa "using namespace std;" sau các khai báo thư viện (các "#include", hay "#define",...)
- Ví dụ:

```
#include <iostream>
#include <stack>  //khai báo sử dụng container stack
#define n 100
using namespace std;  //khai báo sử dụng STL
main() {
      ....
}
```

- Việc sử dụng các hàm trong STL tương tự như việc sử dụng các hàm như trong class. Các bạn đọc qua một vài ví dụ là có thể thấy được quy luật.

I. ITERATOR (BIẾN LẶP):

- Trong C++, một biến lặp là một đối tượng bất kì, trỏ tới một số phần tử trong 1 phạm vi của các phần tử (như mảng hoặc container), có khả năng để lặp các phần tử trong phạm vi bằng cách sử dụng một tập các toán tử (operators) (như so sánh, tăng (++), ...)
- Dạng rõ ràng nhất của iterator là một con trỏ: Một con trỏ có thể trỏ tới các phần tử trong mảng, và có thể lặp thông qua sử dụng toán tử tăng (++). Tuy nhiên, cũng có

các dạng khác của iterator. Ví dụ: mỗi loại container (chẳng hạn như vector) có một loại iterator được thiết kế để lặp các phần tử của nó một cách hiệu quả.

- Iterator có các toán tử như:
    - So sánh: "==" , "!=" giữa 2 iterator.
    - Gán: "=" giữa 2 iterator.
    - Cộng trừ: "+","-" với hằng số và "++","—".
    - Lấy giá trị: "*".

II. CONTAINERS (THƯ VIỆN LƯU TRỮ)

- Một container là một đối tượng cụ thể lưu trữ một tập các đối tượng khác (các phần tử của nó). Nó được thực hiện như các lớp mẫu (class templates).
- Container quản lý không gian lưu trữ cho các phần tử của nó và cung cấp các hàm thành viên (member function) để truy cập tới chúng, hoặc trực tiếp hoặc thông qua các biến lặp (iterator – giống như con trỏ).
- Container xây dựng các cấu trúc thường sử dụng trong lập trình như: mảng động - dynamic arrays (vector), hàng đợi – queues (queue), hàng đợi ưu tiên – heaps (priority queue), danh sách kiên kết – linked list (list), cây – trees (set), mảng ánh xạ - associative arrays (map),...
- Nhiều container chứa một số hàm thành viên giống nhau. Quyết định sử dụng loại container nào cho nhu cầu cụ thể nói chung không chỉ phụ thuộc vào các hàm được cung cấp mà còn phải dựa vào hiệu quả của các hàm thành viên của nó (độ phức tạp (từ giờ mình sẽ viết tắt là ĐPT) của các hàm). Điều này đặc biệt đúng với container dãy (sequence containers), mà trong đó có sự khác nhau về độ phức tạp đối với các thao tác chèn/xóa phần tử hay truy cập vào phần tử.

1. Iterator:

Tất cả các container ở 2 loại: Sequence container và Associative container đều hỗ trợ các iterator như sau (ví dụ với vector, những loại khác có chức năng cũng vậy).

```
/*khai báo iterator "it"*/
vector <int> :: iterator it;
/* trỏ đến vị trí phần tử đầu tiên của vector */
it=vector.begin();
/*trỏ đến vị trí kết thúc (không phải phần tử cuối cùng nhé) của vector) */
it=vector.end();
/* khai báo iterator ngược "rit" */
vector <int> :: reverse_iterator rit; rit = vector.rbegin();
/* trỏ đến vị trí kết thúc của vector theo chiều ngược (không phải phần tử đầu tiên nhé*/
rit = vector.rend();
```

Tất cả các hàm iterator này đều có độ phức tạp O(1).

2. Vector (Mảng động):

Khai báo vector:

```
#include <vector>
...
/* Vector 1 chiều */

/* tạo vector rỗng kiểu dữ liệu int */
vector <int> first;

//tạo vector với 4 phần tử là 100
vector <int> second (4,100);

// lấy từ đầu đến cuối vector second
vector <int> third (second.begin(),second.end())

//copy từ vector third
vector <int> four (third)

/*Vector 2 chiều*/

/* Tạo vector 2 chiều rỗng */
vector < vector <int> > v;

/* khai báo vector 5×10 */
vector < vector <int> > v (5, 10) ;

/* khai báo 5 vector 1 chiều rỗng  */
vector < vector <int> > v (5) ;

//khai báo vector 5*10 với các phần tử khởi tạo giá trị là 1
vector < vector <int> > v (5, vector <int> (10,1) ) ;
```

Các bạn chú ý 2 dấu "ngoặc" không được viết liền nhau.
Ví dụ như sau là **sai**:

```
/*Khai báo vector 2 chiều SAI*/
vector <vector <int>> v;
```

Các hàm thành viên:

Capacity:

- size : trả về số lượng phần tử của vector. ĐPT O(1).
- empty : trả về true(1) nếu vector rỗng, ngược lại là false(0). ĐPT O(1).

Truy cập tới phần tử:

- operator [] : trả về giá trị phần tử thứ []. ĐPT O(1).
- at : tương tự như trên. ĐPT O(1).
- front: trả về giá trị phần tử đầu tiên. ĐPT O(1).
- back: trả về giá trị phần tử cuối cùng. ĐPT O(1).

Chỉnh sửa:

- push_back : thêm vào ở cuối vector. ĐPT O(1).
- pop_back : loại bỏ phần tử ở cuối vector. ĐPT O(1).
- insert (iterator,x): chèn "x" vào trước vị trí "iterator" (x có thể là phần tử hay iterator của 1 đoạn phần tử…). ĐPT O(n).
- erase : xóa phần tử ở vị trí iterator. ĐPT O(n).
- swap : đổi 2 vector cho nhau (ví dụ: first.swap(second);). ĐPT O(1).
- clear: xóa vector. ĐPT O(n).

Nhận xét:

- Sử dụng vector sẽ tốt khi:
  - Truy cập đến phần tử riêng lẻ thông qua vị trí của nó O(1)
  - Chèn hay xóa ở vị trí cuối cùng O(1).
- Vector làm việc giống như một "mảng động".

Ví dụ 1: Ví dụ này chủ yếu để làm quen sử dụng các hàm chứ không có đề bài cụ thể.

```
#include <iostream>
#include <vector>
using namespace std;
vector <int> v; //Khai báo vector
vector <int>::iterator it;  //Khai báo iterator
vector <int>::reverse_iterator rit; //Khai báo iterator ngược
int i;
main() {
	for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
	cout << v.front() << endl;         // In ra 1
	cout << v.back() << endl;          // In ra 5

	cout << v.size() << endl;          // In ra 5

	v.push_back(9);                    // v={1,2,3,4,5,9}
	cout << v.size() << endl;          // In ra 6

	v.clear();                         // v={}
	cout << v.empty() << endl;         // In ra 1 (vector rỗng)

	for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
	v.pop_back();                      // v={1,2,3,4}
	cout << v.size() << endl;          // In ra 4

	v.erase(v.begin()+1);              // Xóa ptử thứ 1 v={1,3,4}
	v.erase(v.begin(),v.begin()+2);    // v={4}
	v.insert(v.begin(),100);           // v={100,4}
	v.insert(v.end(),5);               // v={100,4,5}

	/*Duyệt theo chỉ số phần tử*/
	for (i=0;i<v.size();i++) cout << v[i] << " "; // 100 4 5
	cout << endl;



   /*Chú ý: Không nên viết
```

```
        for (i=0;i<=v.size()-1;i++) ...
        Vì nếu vector v rỗng thì sẽ dẫn đến sai khi duyệt !!!
        */

        /*Duyệt theo iterator*/
        for (it=v.begin();it!=v.end();it++)
            cout << *it << " " ;
        //In ra giá trị ma iterator đang trỏ tới "100 4 5"
        cout << endl;

        /*Duyệt iterator ngược*/
        for (rit=v.rbegin();rit!=v.rend();rit++)
            cout << *rit << " "; // 5 4 100
        cout << endl;

        system("pause");
}
```

Ví dụ 2: Cho đồ thị vô hướng G có n đỉnh (các đỉnh đánh số từ 1 đến n) và m cạnh và không có khuyên (đường đi từ 1 đỉnh tới chính đỉnh đó).
Cài đặt đồ thị bằng danh sách kề và in ra các cạnh kề đối với mỗi cạnh của đồ thị.
Dữ liệu vào:

- Dòng đầu chứa n và m cách nhau bởi dấu cách
- M dòng sau, mỗi dòng chứa u và v cho biết có đường đi từ u tới v. Không có cặp đỉnh u,v nào chỉ cùng 1 đường đi.

Dữ liệu ra:

- M dòng: Dòng thứ i chứa các đỉnh kề cạnh i theo thứ tự tăng dần và cách nhau bởi dấu cách.

Giới hạn: 1 <= n,m <= 10000
Ví dụ:

INPUT	OUTPUT
6 7	2 3 5 6
1 2	1 3 6
1 3	1 2 5
1 5	
2 3	1 3
2 6	1 2
3 5	
6 1	

Chương trình mẫu:

```
#include <iostream>
#include <vector>
using namespace std;
vector < vector <int> > a (10001);
//Khai báo vector 2 chiều với 10001 vector 1 chiều rỗng
int m,n,i,j,u,v;
main() {
        /*Input data*/
```

```
        cin >> n >> m;
        for (i=1;i<=m;i++) {
            cin >> u >> v;
            a[u].push_back(v);
            a[v].push_back(u);
        }
        /*Sort cạnh kề*/
        for (i=1;i<=m;i++)
            sort(a[i].begin(),a[i].end());
        /*Print Result*/
        for (i=1;i<=m;i++) {
            for (j=0;j<a[i].size();j++) cout << a[i][j] << " ";
            cout << endl;
        }
        system("pause");
}
```

3. Deque (Hàng đợi hai đầu):

- Deque (thường được phát âm giống như "deck") là từ viết tắt của double-ended queue (hàng đợi hai đầu).
- Deque có các ưu điểm như:
    - Các phần tử có thể truy cập thông cua chỉ số vị trí của nó. O(1)
    - Chèn hoặc xóa phần tử ở cuối hoặc đầu của dãy. O(1)

Khai báo: `#include <deque>`

Capacity:

- size : trả về số lượng phần tử của deque. ĐPT O(1).
- empty : trả về true(1) nếu deque rỗng, ngược lại là false(0). ĐPT O(1).

Truy cập phần tử:

- operator [] : trả về giá trị phần tử thứ []. ĐPT O(1).
- at : tương tự như trên. ĐPT O(1).
- front: trả về giá trị phần tử đầu tiên. ĐPT O(1).
- back: trả về giá trị phần tử cuối cùng. ĐPT O(1).

Chỉnh sửa:

- push_back : thêm phần tử vào ở cuối deque. ĐPT O(1).
- push_front : thêm phần tử vào đầu deque. ĐPT O(1).
- pop_back : loại bỏ phần tử ở cuối deque. ĐPT O(1).
- pop_front : loại bỏ phần tử ở đầu deque. ĐPT O(1).
- insert (iterator,x): chèn "x" vào trước vị trí "iterator" (x có thể là phần tử hay iterator của 1 đoạn phần tử…). ĐPT O(n).
- erase : xóa phần tử ở vị trí iterator. ĐPT O(n).
- swap : đổi 2 deque cho nhau (ví dụ: first.swap(second);). ĐPT O(n).

clear: xóa vector. ĐPT O(1).

4. List (Danh sách liên kết):

- List được thực hiện như danh sách nối kép (doubly-linked list). Mỗi phần tử trong danh sách nối kép có liên kết đến một phần tử trước đó và một phần tử sau nó.

- Do đó, list có các ưu điểm như sau:
    - Chèn và loại bỏ phần tử ở bất cứ vị trí nào trong container. O(1).
- Điểm yếu của list là khả năng truy cập tới phần tử thông qua vị trí. O(n).
- Khai báo: #include <list>

Các hàm thành viên:

Capacity:
- size : trả về số lượng phần tử của list. ĐPT O(1).
- empty : trả về true(1) nếu list rỗng, ngược lại là false(0). ĐPT O(1).

Truy cập phần tử:
- front: trả về giá trị phần tử đầu tiên. ĐPT O(1).
- back: trả về giá trị phần tử cuối cùng. ĐPT O(1).

Chỉnh sửa:
- push_back : thêm phần tử vào ở cuối list. ĐPT O(1).
- push_front : thêm phần tử vào đầu list. ĐPT O(1).
- pop_back : loại bỏ phần tử ở cuối list. ĐPT O(1).
- pop_front : loại bỏ phần tử ở đầu list. ĐPT O(1).
- insert (iterator,x): chèn "x" vào trước vị trí "iterator" (x có thể là phần tử hay iterator của 1 đoạn phần tử…). ĐPT là số phần tử thêm vào.
- erase : xóa phần tử ở vị trí iterator. ĐPT là số phần tử bị xóa đi.
- swap : đổi 2 list cho nhau (ví dụ: first.swap(second);). ĐPT O(1).
- clear: xóa list. ĐPT O(n).

Operations:
- splice : di chuyển phần tử từ list này sang list khác. ĐPT O(n).
- remove (const) : loại bỏ tất cả phần tử trong list bằng const. ĐPT O(n).
- remove_if (function) : loại bỏ tất các phần tử trong list nếu hàm function return true . ĐPT O(n).
- unique : loại bỏ các phần tử bị trùng lặp hoặc thỏa mãn hàm nào đó. ĐPT O(n). Lưu ý: Các phần tử trong list phải được sắp xếp.
- sort : sắp xếp các phần tử của list. O(NlogN)
- reverse : đảo ngược lại các phần tử của list. O(n).

5. Stack (Ngăn xếp):

- Stack là một loại container adaptor, được thiết kế để hoạt động theo kiểu LIFO (Last - in first - out) (vào sau ra trước), tức là một kiểu danh sách mà việc bổ sung và loại bỏ một phần tử được thực hiển ở cuối danh sách. Vị trí cuối cùng của stack gọi là đỉnh (top) của ngăn xếp.

Khai báo: `#include <stack>`

Các hàm thành viên:
- size : trả về kích thước hiện tại của stack. ĐPT O(1).
- empty : true stack nếu rỗng, và ngược lại. ĐPT O(1).
- push : đẩy phần tử vào stack. ĐPT O(1).
- pop : loại bỏ phần tử ở đỉnh của stack. ĐPT O(1).
- top : truy cập tới phần tử ở đỉnh stack. ĐPT O(1).

Chương trình demo:

```
#include <iostream>
#include <stack>
using namespace std;
stack <int> s;
int i;
main() {
	for (i=1;i<=5;i++) s.push(i); // s={1,2,3,4,5}
	s.push(100);                  // s={1,2,3,4,5,100}
	cout << s.top() << endl;      // In ra 100
	s.pop();                      // s={1,2,3,4,5}
	cout << s.empty() << endl;    // In ra 0
	cout << s.size() << endl;     // In ra 5
	system("pause");
}
```

6. Queue (Hàng đợi):

- Queue là một loại container adaptor, được thiết kế để hoạt động theo kiểu FIFO (First - in first - out) (vào trước ra trước), tức là một kiểu danh sách mà việc bổ sung được thực hiển ở cuối danh sách và loại bỏ ở đầu danh sách.
- Trong queue, có hai vị trí quan trọng là vị trí đầu danh sách (front), nơi phần tử được lấy ra, và vị trí cuối danh sách (back), nơi phần tử cuối cùng được thêm vào.

Khai báo: `#include <queue>`

Các hàm thành viên:

- size : trả về kích thước hiện tại của queue. ĐPT O(1).
- empty : true nếu queue rỗng, và ngược lại. ĐPT O(1).
- push : đẩy vào cuối queue. ĐPT O(1).
- pop: loại bỏ phần tử (ở đầu). ĐPT O(1).
- front : trả về phần tử ở đầu. ĐPT O(1).
- back: trả về phần tử ở cuối. ĐPT O(1).

Chương trình demo:

```
#include <iostream>
#include <queue>
using namespace std;
queue <int> q;
int i;
main() {
	for (i=1;i<=5;i++) q.push(i); // q={1,2,3,4,5}
	q.push(100);                  // q={1,2,3,4,5,100}
	cout << q.front() << endl;    // In ra 1
	q.pop();                      // q={2,3,4,5,100}
	cout << q.back() << endl;     // In ra 100
	cout << q.empty() << endl;    // In ra 0
	cout << q.size() << endl;     // In ra 5
	system("pause");
}
```

7. Priority Queue (Hàng đợi ưu tiên):

- Priority queue là một loại container adaptor, được thiết kế đặc biệt để phần tử ở đầu luôn luôn lớn nhất (theo một quy ước về độ ưu tiên nào đó) so với các phần tử khác.
- Nó giống như một heap, mà ở đây là heap max, tức là phần tử có độ ưu tiên lớn nhất có thể được lấy ra và các phần tử khác được chèn vào bất kì.
- Phép toán so sánh mặc định khi sử dụng priority queue là phép toán less (Xem thêm ở thư viện **functional**)
- Để sử dụng priority queue một cách hiệu quả, các bạn nên học cách viết hàm so sánh để sử dụng cho linh hoạt cho từng bài toán.
- Khai báo: `#include <queue>`

```
/*Dạng 1 (sử dụng phép toán mặc định là less)*/
priority_queue <int> pq;

/* Dạng 2 (sử dụng phép toán khác) */
priority_queue <int,vector<int>,greater<int> > q; //phép toán greater
```

Phép toán khác cũng có thể do người dùng tự định nghĩa. Ví dụ:
Cách khai báo ở dạng 1 tương đương với:

```
/* Dạng sử dụng class so sánh tự định nghĩa */
struct cmp{
	bool operator() (int a,int b) {return a<b;}
};


main() {
	…
	priority_queue <int,vector<int>,cmp > q;
}
```

Các hàm thành viên:

- size : trả về kích thước hiện tại của priority queue. ĐPT O(1)
- empty : true nếu priority queue rỗng, và ngược lại. ĐPT O(1).
- push : đẩy vào priority queue. ĐPT O(logN).
- pop: loại bỏ phần tử ở đỉnh priority queue. ĐPT O(logN).
- top : trả về phần tử ở đỉnh priority queue. ĐPT O(1).

Chương trình Demo 1:

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

main() {
	priority_queue <int> p;  // p={}
	p.push(1);               // p={1}
	p.push(5);               // p={1,5}
	cout << p.top() << endl; // In ra 5
	p.pop();                 // p={1}
	cout << p.top() << endl; // In ra 1
```

```
        p.push(9);              // p={1,9}
        cout << p.top() << endl; // In ra 9
        system("pause");
}
```

Chương trình Demo 2:

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

main() {
        priority_queue < int , vector <int> , greater <int> > p;  // p={}
        p.push(1);              // p={1}
        p.push(5);              // p={1,5}
        cout << p.top() << endl; // In ra 1
        p.pop();                // p={5}
        cout << p.top() << endl; // In ra 5
        p.push(9);              // p={5,9}
        cout << p.top() << endl; // In ra 5
        system("pause");
}
```

Chương trình Demo 3:

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

struct cmp{
        bool operator() (int a,int b) {return a<b;}
};

main() {
        priority_queue < int , vector <int> , cmp > p;  // p={}
        p.push(1);              // p={1}
        p.push(5);              // p={1,5}
        cout << p.top() << endl; // In ra 1
        p.pop();                // p={5}
        cout << p.top() << endl; // In ra 5
        p.push(9);              // p={5,9}
        cout << p.top() << endl; // In ra 5
        system("pause");
}
```

8. Set (Tập hợp):

- Set là một loại associative containers để lưu trữ các phần tử không bị trùng lặp (unique elements), và các phần tử này chính là các khóa (keys).
- Khi duyệt set theo iterator từ begin đến end, các phần tử của set sẽ tăng dần theo phép toán so sánh.
- Mặc định của set là sử dụng phép toán less, bạn cũng có thể viết lại hàm so sánh theo ý mình.

- Set được thực hiện giống như cây tìm kiếm nhị phân (Binary search tree).

Khai báo:

```
#include <set>
set <int> s;
set <int, greater<int> > s;
```

Hoặc viết class so sánh theo ý mình:

```
struct cmp{
	bool operator() (int a,int b) {return a<b;}
};
set <int,cmp > myset ;
```

Capacity:

- size : trả về kích thước hiện tại của set. ĐPT O(1)
- empty : true nếu set rỗng, và ngược lại. ĐPT O(1).

Modifiers:

- insert : Chèn phần tử vào set. ĐPT O(logN).
- erase : có 2 kiểu xóa: xóa theo iterator, hoặc là xóa theo khóa. ĐPT O(logN).
- clear : xóa tất cả set. ĐPT O(n).
- swap : đổi 2 set cho nhau. ĐPT O(n).

Operations:

- find : trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy itarator trỏ về "end" của set. ĐPT O(logN).
- lower_bound : trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí "end" của set. ĐPT O(logN).
- upper_bound: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí "end" của set.. ĐPT O(logN).
- count : trả về số lần xuất hiện của khóa trong container. Nhưng trong set, các phần tử chỉ xuất hiện một lần, nên hàm này có ý nghĩa là sẽ return 1 nếu khóa có trong container, và 0 nếu không có. ĐPT O(logN).

Chương trình Demo 1:

```
#include <iostream>
#include <set>

using namespace std;

main() {
	set <int> s;
	set <int> ::iterator it;
	s.insert(9);                // s={9}
	s.insert(5);                // s={5,9}
	cout << *s.begin() << endl; //In ra 5
	s.insert(1);                // s={1,5,9}
	cout << *s.begin() << endl; // In ra 1

	it=s.find(5);
	if (it==s.end()) cout << "Khong co trong container" << endl;
	else  cout << "Co trong container" << endl;

	s.erase(it);                // s={1,9}
	s.erase(1);                 // s={9}
```

```
        s.insert(3);                // s={3,9}
        s.insert(4);                // s={3,4,9}

        it=s.lower_bound(4);
        if (it==s.end()) cout << "Khong co phan tu nao trong set khong be hon 4" << endl;
        else cout << "Phan tu be nhat khong be hon 4 la " << *it << endl;  // In ra 4

        it=s.lower_bound(10);
        if (it==s.end()) cout << "Khong co phan tu nao trong set khong be hon 10" << endl;
        else cout << "Phan tu be nhat khong be hon 10 la " << *it << endl; // Khong co ptu nao

        it=s.upper_bound(4);
        if (it==s.end()) cout << "Khong co phan tu nao trong set lon hon 4" << endl;
        else cout << "Phan tu be nhat lon hon 4 la " << *it << endl;       // In ra 9

        /* Duyet set */

        for (it=s.begin();it!=s.end();it++) {
            cout << *it <<  " ";
        }
        // In ra 3 4 9

        cout << endl;
        system("pause");
}
```

Lưu ý: Nếu bạn muốn sử dụng hàm lower_bound hay upper_bound để tìm phần tử lớn nhất "bé hơn hoặc bằng" hoặc "bé hơn" bạn có thể thay đổi cách so sánh của set để tìm kiếm. Mời bạn xem chương trình sau để rõ hơn:

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

using namespace std;

main() {
        set <int, greater <int> > s;
        set <int, greater <int> > :: iterator it; // Phép toán so sánh là greater

        s.insert(1);                    // s={1}
        s.insert(2);                    // s={2,1}
        s.insert(4);                    // s={4,2,1}
        s.insert(9);                    // s={9,4,2,1}

        /* Tim phần tử lớn nhất bé hơn hoặc bằng 5 */
        it=s.lower_bound(5);
        cout << *it << endl;            // In ra 4

        /* Tim phần tử lớn nhất bé hơn 4 */
        it=s.upper_bound(4);
        cout << *it << endl;            // In ra 2

        system("pause");
}
```

9. Mutiset (Tập hợp):

- Multiset giống như Set nhưng có thể chứa các khóa có giá trị giống nhau.

- Khai báo : giống như set.
- **Các hàm thành viên:**

Capacity:
- size : trả về kích thước hiện tại của multiset. ĐPT O(1)
- empty : true nếu multiset rỗng, và ngược lại. ĐPT O(1).

Chỉnh sửa:
- insert : Chèn phần tử vào set. ĐPT O(logN).
- erase :
  - xóa theo iterator ĐPT O(logN)
  - xóa theo khóa: xóa tất cả các phần tử bằng khóa trong multiset ĐPT: O(logN) + số phần tử bị xóa.
- clear : xóa tất cả set. ĐPT O(n).
- swap : đổi 2 set cho nhau. ĐPT O(n).

Operations:
- find : trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy itarator trỏ về "end" của set. ĐPT O(logN). Dù trong multiset có nhiều phần tử bằng khóa thì nó cũng chỉ iterator đến một phần tử.
- lower_bound : trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí "end" của set. ĐPT O(logN).
- upper_bound: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí "end" của set.. ĐPT O(logN).
- count : trả về số lần xuất hiện của khóa trong multiset. ĐPT O(logN) + số phần tử tìm được.

Chương trình Demo:

```
#include <iostream>
#include <set>
using namespace std;
main() {
	multiset <int> s;
	multiset <int> :: iterator it;
	int i;
	for (i=1;i<=5;i++) s.insert(i*10); // s={10,20,30,40,50}
	s.insert(30);                      // s={10,20,30,30,40,50}
	cout << s.count(30) << endl;       // In ra 2
	cout << s.count(20) << endl;       // In ra 1
	s.erase(30);                       // s={10,20,40,50}

	/* Duyet set */

	for (it=s.begin();it!=s.end();it++) {
	    cout << *it <<  " ";
	}
	// In ra 10 20 40 50
	cout << endl;
	system("pause");
}
```

10. Map (Ánh xạ):

- Map là một loại associative container. Mỗi phần tử của map là sự kết hợp của khóa (key value) và ánh xạ của nó (mapped value). Cũng giống như set, trong map không chứa các khóa mang giá trị giống nhau.
- Trong map, các khóa được sử dụng để xác định giá trị các phần tử. Kiểu của khóa và ánh xạ có thể khác nhau.
- Và cũng giống như set, các phần tử trong map được sắp xếp theo một trình tự nào đó theo cách so sánh.
- Map được cài đặt bằng red-black tree (cây đỏ đen) – một loại cây tìm kiếm nhị phân tự cân bằng. Mỗi phần tử của map lại được cài đặt theo kiểu pair (xem thêm ở thư viện **utility**).

Khai báo:

```
#include <map>
...
map <kiểu_dữ_liệu_1,kiểu_dữ_liệu_2>
// kiểu dữ liệu 1 là khóa, kiểu dữ liệu 2 là giá trị của khóa.
```

Sử dụng class so sánh:

Dạng 1:

```
struct cmp{
	bool operator() (char a,char b) {return a<b;}
};
.....
map <char,int,cmp> m;
```

- Truy cập đến giá trị của các phần tử trong map khi sử dụng iterator:

Ví dụ ta đang có một iterator là it khai báo cho map thì:

```
	(*it).first;    // Lấy giá trị của khóa, kiểu_dữ_liệu_1
	(*it).second;   // Lấy giá trị của giá trị của khóa, kiểu_dữ_liệu_2
	(*it)          // Lấy giá trị của phần tử mà iterator đang trỏ đến, kiểu pair

	it->first;      // giống như (*it).first
	it->second;     // giống như (*it).second
```

Capacity:

- size : trả về kích thước hiện tại của map. ĐPT O(1)
- empty : true nếu map rỗng, và ngược lại. ĐPT O(1).

Truy cập tới phần tử:

- operator [khóa]: Nếu khóa đã có trong map, thì hàm này sẽ trả về giá trị mà khóa ánh xạ đến. Ngược lại, nếu khóa chưa có trong map, thì khi gọi [] nó sẽ thêm vào map khóa đó. ĐPT O(logN)

Chỉnh sửa

- insert : Chèn phần tử vào map. Chú ý: phần tử chèn vào phải ở kiểu "pair". ĐPT O(logN).
- erase :
  - xóa theo iterator ĐPT O(logN)
  - xóa theo khóa: xóa khóa trong map. ĐPT: O(logN).
- clear : xóa tất cả set. ĐPT O(n).
- swap : đổi 2 set cho nhau. ĐPT O(n).

Operations:

- find : trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy iterator trỏ về "end" của map. ĐPT O(logN).
- lower_bound : trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí "end" của map. ĐPT O(logN).
- upper_bound: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí "end" của map. ĐPT O(logN).
- count : trả về số lần xuất hiện của khóa trong multiset. ĐPT O(logN)

Chương trình demo:

```
#include <iostream>
#include <map>
#include <vector>
using namespace std;
main() {
	map <char,int> m;
	map <char,int> :: iterator it;

	m['a']=1;                              // m={{'a',1}}
	m.insert(make_pair('b',2));            // m={{'a',1};{'b',2}}
	m.insert(pair<char,int>('c',3) );      // m={{'a',1};{'b',2};{'c',3}}

	cout << m['b'] << endl;                // In ra 2
	m['b']++;                              // m={{'a',1};{'b',3};{'c',3}}

	it=m.find('c');                        // it point to key 'c'

	cout << it->first << endl;             // In ra 'c'
	cout << it->second << endl;            // In ra 3

	m['e']=100;                           //m={{'a',1};{'b',3};{'c',3};{'e',100}}

	it=m.lower_bound('d');                 // it point to 'e'
	cout << it->first << endl;             // In ra 'e'
	cout << it->second << endl;            // In ra 100

	system("pause");
}
```

11. Multi Map (Ánh xạ):

Giống như map nhưng có thể chứa các phần tử có khóa giống nhau, do đó nó khác map ở chỗ không có opearator[].

III. STL ALGORITHMS (THƯ VIỆN THUẬT TOÁN):

1. Giới thiệu tổng quan:

Thư viện algorithm nằm trong thư viện chuẩn (Standard Template Libarary) của C++, nó được thiết kế để thực hiện các thuật toán cơ bản, các thao tác thường sử dụng (như là tìm kiếm, sắp xếp, trộn, chèn, xóa …) trên một khoảng liên tiếp của dãy các phần tử có cùng kiểu dữ liệu.
Một khoảng là một dãy các đối tượng có thể truy cập thông qua các biến lặp (iterator) hoặc các con trỏ (pointer). Ví dụ như là một mảng hoặc một số STL container.
Các hàm STL Algorithm chia thành 7 loại cơ bản:

- Các thao tác không thay đổi đoạn phần tử (Non-modifying sequence operations)
- Các thao tác thay đổi đoạn phần tử (Modifying sequence operations)
- Sắp xếp (Sorting)
- Tím kiếm nhị phân (Binary Search)
- Trộn (Merge)
- Hàng đợi ưu tiên (Heap)
- Min/max

Để sử dụng thư viện algorithm bạn cần phải khai báo: #include <algorithm>

Sau đây sẽ là phần trình bày chi tiết của mỗi loại.

2. Các thao tác không thay đổi đoạn phần tử:

2.1. for_each:

```
template <class InputIterator, class Function>
   Function for_each (InputIterator first, InputIterator last, Function f);
```

Ý nghĩa:

- Thực hiện hàm *'f'* cho mỗi phần tử thuộc nửa đoạn *[first,last)*

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- f: hàm duy nhất nhận phần tử trong đoạn làm tham số. Nó có thể là con trỏ đến hàm hoặc đối tượng mà lớp overloads toán tử (). Xem ví dụ để hiểu rõ hơn.

Ví dụ:

```
// for_each example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void myfunction (int i) {
  cout << " " << i;
}

struct myclass {
  void operator() (int i) {cout << " " << i;}
```

```
} myobject;

int main () {
  vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(20);
  myvector.push_back(30);

  cout << "myvector contains:";
  for_each (myvector.begin(), myvector.end(), myfunction);

  // or:
  cout << "\nmyvector contains:";
  for_each (myvector.begin(), myvector.end(), myobject);

  cout << endl;

  return 0;
}
```

Output:

```
myvector contains: 10 20 30
myvector contains: 10 20 30
```

2.2. find:

Dạng:

```
template <class InputIterator, class T>
   InputIterator find ( InputIterator first, InputIterator last, const T& value );
```

Ý nghĩa:

- Tìm kiếm phần tử. Trả về con trỏ đến phần tử đầu tiên trong đoạn *[first,last)* mà so sánh bằng *'value'*, hoặc trả về *'last'* nếu không tìm thấy.

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- value: phần tử tìm kiếm

Ví dụ:

```
// find example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[] = { 10, 20, 30 ,40 };
  int * p;

  // pointer to array element:
  p = find(myints,myints+4,30);
  ++p;
  cout << "The element following 30 is " << *p << endl;

  vector<int> myvector (myints,myints+4);
  vector<int>::iterator it;

  // iterator to vector element:
```

```
  it = find (myvector.begin(), myvector.end(), 30);
  ++it;
  cout << "The element following 30 is " << *it << endl;

  return 0;
}
```

Output:

```
The element following 30 is 40
The element following 30 is 40
```

2.3. find_if:

Dạng:

```
template <class InputIterator, class Predicate>
   InputIterator find_if ( InputIterator first, InputIterator last, Predicate pred
);
```

Ý nghĩa:

- Tìm kiếm phần tử. Trả về con trỏ đến phần tử đầu tiên trong đoạn *[first,last)* mà khi áp dụng *'pred'* trả vê true, hoặc trả về *'last'* nếu không tìm thấy.

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- pred: giống như *'f'* trong hàm 'for_each' ở trên, nhưng thay vì thực hiện một nhiệm vụ nào đó thì 'pred' phải trả về giá trị *'true'* hoặc *'false'* để so sánh.

Ví dụ:

```
// find_if example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool IsOdd (int i) {
  return ((i%2)==1);
}

int main () {
  vector<int> myvector;
  vector<int>::iterator it;

  myvector.push_back(10);
  myvector.push_back(25);
  myvector.push_back(40);
  myvector.push_back(55);

  it = find_if (myvector.begin(), myvector.end(), IsOdd);
  cout << "The first odd value is " << *it << endl;

  return 0;
}
```

Output:

```
The first odd value is 25
```

2.4. count:

Dạng:

```
template<class InputIterator, class Type>
   typename iterator_traits<InputIterator>::difference_type count(
      InputIterator _First,
      InputIterator _Last,
      const Type& _Val
   );
```

Ý nghĩa:

- Trả về số phần tử trong đoạn *[first,last)* mà bằng *'value'*.

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- value: phần tử so sánh để đếm số lần xuất hiện

Ví dụ:

```
// count algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int mycount;

  // counting elements in array:
  int myints[] = {10,20,30,30,20,10,10,20};   // 8 elements
  mycount = (int) count (myints, myints+8, 10);
  cout << "10 appears " << mycount << " times.\n";

  // counting elements in container:
  vector<int> myvector (myints, myints+8);
  mycount = (int) count (myvector.begin(), myvector.end(), 20);
  cout << "20 appears " << mycount  << " times.\n";

  return 0;
}
```

Output:

```
10 appears 3 times.
   20 ppears 3 times.
```

2.5. count_if:

Dạng:

```
template<class InputIterator, class Predicate>
   typename iterator_traits<InputIterator>::difference_typecount_if(
      InputIterator _First,
      InputIterator _Last,
      Predicate _Pred
   );
```

Ý nghĩa:

- Trả về số phần tử trong đoạn *[first,last)* mà khi áp dụng *'pred'* trả về *'true'*.

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).

- pred: giống như *'f'* trong hàm 'for_each' ở trên, nhưng thay vì thực hiện một nhiệm vụ nào đó thì 'pred' phải trả về giá trị *'true'* hoặc *'false'* để so sánh.

Ví dụ:

```
// count_if example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool IsOdd (int i) { return ((i%2)==1); }

int main () {
  int mycount;

  vector<int> myvector;
  for (int i=1; i<10; i++) myvector.push_back(i); // myvector: 1 2 3 4 5 6
7 8 9

  mycount = (int) count_if (myvector.begin(), myvector.end(), IsOdd);
  cout << "myvector contains " << mycount  << " odd values.\n";

  return 0;
}
```

Output:

```
myvector contains 5 odd values.
```

3. Các thao tác thay đổi đoạn phần tử:

3.1. Copy:

Dạng:

```
template <class InputIterator, class OutputIterator>
  OutputIterator copy ( InputIterator first, InputIterator last, OutputIterator
result );
```

Ý nghĩa:

- Sao chép đoạn phần tử *[first,last)* đến đoạn phần tử mới bắt đầu từ *'result'*

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- result: trỏ đến phần tử đầu tiên của đoạn kết quả.

Ví dụ:

```
// copy algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[]={10,20,30,40,50,60,70};
  vector<int> myvector;
  vector<int>::iterator it;

  myvector.resize(7);   // allocate space for 7 elements
```

```
  copy ( myints, myints+7, myvector.begin() );

  cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

Output:

```
myvector contains: 10 20 30 40 50 60 70
```

3.2. Swap:

Dạng:

```
template <class T> void swap ( T& a, T& b );
```

Ý nghĩa:

- Trao đổi giá trị của hai đối tượng. Gán giá trị của a cho b, và b cho a.

Tham số:

- a,b: 2 đối tượng có cùng kiểu.

Ví dụ:

```
// swap algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {

  int x=10, y=20;                          // x:10 y:20
  swap(x,y);                               // x:20 y:10
  return 0;
}
```

3.3. Replace:

Dạng:

```
template < class ForwardIterator, class T >
  void replace ( ForwardIterator first, ForwardIterator last,
                 const T& old_value, const T& new_value );
```

Ý nghĩa:

- Thay thế tất cả các phần tử trong đoạn *[first,last)* có giá trị *'old_value'* bằng giá trị mới *'new_value'*.

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- old_value: giá trị cần thay thế.
- new_value: giá trị thay thế vào.

Ví dụ:

```
// replace algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
  vector<int> myvector (myints, myints+8);            // 10 20 30 30 20 10
10 20

  replace (myvector.begin(), myvector.end(), 20, 99); // 10 99 30 30 99 10
10 99

  cout << "myvector contains:";
  for (vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

Output:

```
myvector contains: 10 99 30 30 99 10 10 99
```

3.4. Remove:

Dạng:

```
template < class ForwardIterator, class T >
  ForwardIterator remove ( ForwardIterator first, ForwardIterator last,
                           const T& value );
```

Ý nghĩa:

- Xóa bỏ các phần tử trong đoạn *[first,last)* mà có giá trị là *'value'*.

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- value: giá trị cần thay thế.

Ví dụ:

```
// remove algorithm example
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};      // 10 20 30 30 20 10 10 20

  // bounds of range:
  int* pbegin = myints;                          // ^
  int* pend = myints+sizeof(myints)/sizeof(int); // ^
^

  pend = remove (pbegin, pend, 20);              // 10 30 30 10 10 ?  ?  ?
                                                 // ^              ^
  cout << "range contains:";
  for (int* p=pbegin; p!=pend; ++p)
    cout << " " << *p;
```

```
  cout << endl;

  return 0;
}
```

Output:

```
range contains: 10 30 30 10 10
```

3.5. Unique:

Dạng 1:

```
template <class ForwardIterator>
  ForwardIterator unique ( ForwardIterator first, ForwardIterator last );
```

Dạng 2:

```
template <class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique ( ForwardIterator first, ForwardIterator last,
                           BinaryPredicate pred );
```

Ý nghĩa:

- Loại bỏ các phần tử liên tiếp có giá trị bằng nhau trong đoạn *[first,last)*.
- Sau khi 'unique', hàm trả về con trỏ đến vị trí kết thúc của đoạn mới (khác với phần tử cuối cùng), đoạn cũ vẫn giữ nguyên kích thước, một số phần tử sau vị trí kết thúc có giá trị không xác định.
- Việc so sánh giữa các yếu tố thực hiện bằng cách áp dụng toán tử so sánh *'=='* (dạng 1) hoặc tham số mẫu *'comp'* (dạng 2).

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- comp: hàm so sánh hai đối tượng (có cùng kiểu dữ liệu với đoạn phần tử), trả về *'true'* nếu 2 tham số bằng nhau. ('bằng nhau' ở đây do người sử dụng tự định nghĩa), và false nếu ngược lại.

Ví dụ:

```
// unique algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i, int j) {
  return (i==j);
}

int main () {
  int myints[] = {10,20,20,20,30,30,20,20,10};    // 10 20 20 20 30 30 20
20 10
  vector<int> myvector (myints,myints+9);
  vector<int>::iterator it;

  // using default comparison:
  it = unique (myvector.begin(), myvector.end()); // 10 20 30 20 10 ?  ?  ?
?
                                                  //                ^

  myvector.resize( it - myvector.begin() );       // 10 20 30 20 10

  // using predicate comparison:
  unique (myvector.begin(), myvector.end(), myfunction);   // (no changes)
```

```
  // print out content:
  cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

Output:

```
myvector contains: 10 20 30 20 10
```

3.6. Reverse:

Dạng:

```
template <class BidirectionalIterator>
  void reverse ( BidirectionalIterator first, BidirectionalIterator last);
```

Ý nghĩa:

- Đảo ngược thứ tự các phần tử trong đoạn [first,last).

Tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).

Ví dụ:

```
// reverse algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  vector<int> myvector;
  vector<int>::iterator it;

  // set some values:
  for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

  reverse(myvector.begin(),myvector.end());       // 9 8 7 6 5 4 3 2 1

  // print out content:
  cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

Output:

```
myvector contains: 9 8 7 6 5 4 3 2 1
```

4. Sắp xếp:

4.1. Sort:

Dạng 1:

```
template <class RandomAccessIterator>
  void sort ( RandomAccessIterator first, RandomAccessIterator last );
```

Dạng 2:

```
template <class RandomAccessIterator, class Compare>
  void sort ( RandomAccessIterator first, RandomAccessIterator last,
Compare comp );
```

Ý nghĩa:

- Hàm sort có tác dụng sắp xếp các phần tử trong nửa đoạn [first,last] theo thứ tự tăng dần.
- Các phần tử được so sánh bằng cách sử dụng toán tử '<' trong phiên bản đầu tiên, và toán tử 'comp' trong phiên bản thứ 2.

Các tham số:

- first, last: biến lặp truy cập ngẫu nhiên đến vị trí đầu và cuối, bao gồm phần tử đầu tiên first và tất cả các phần tử giữa first và last (không có phần tử cuối cùng).
- comp: hàm so sánh hai đối tượng (có cùng kiểu dữ liệu với đoạn phần tử), trả về true nếu tham số đầu tiên 'bé hơn' tham số thứ hai ('bé hơn' ở đây do người sử dụng tự định nghĩa), và false nếu ngược lại.

Ví dụ:

```
// sort algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  vector<int> myvector (myints, myints+8);               // 32 71 12 45 26
80 53 33
  vector<int>::iterator it;

  // sử dụng toán tử mặc định (operator <):
  sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26
80 53 33

  // sử dụng hàm để so sánh
  sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26
33 53 80)

  // sử dụng đối tượng để so sánh
  sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45
53 71 80)
```

```
  // In ra các phần tử:
  cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

Output:

```
myvector contains: 12 26 32 33 45 53 71 80
```

Độ phức tạp:
Khoảng $N \times logN$, với N là số phần tử sắp xếp.
Trong trường hợp xấu nhất là N^2 (rất khó xảy ra).

5. Tìm kiếm nhị phân (Đoạn đã sắp xếp):

5.1. lower_bound:

Dạng 1:

```
template <class ForwardIterator, class T>
  ForwardIterator lower_bound ( ForwardIterator first, ForwardIterator last,
                                const T& value );
```

Dạng 2:

```
template <class ForwardIterator, class T, class Compare>
  ForwardIterator lower_bound ( ForwardIterator first, ForwardIterator last,
                                const T& value, Compare comp );
```

Ý nghĩa:

- Trả về con trỏ đến phần tử đầu tiên trong đoạn [first,last) (đã sắp xếp) mà không bé hơn ‘value’. Phép toán so sánh mặc định là toán tử ‘<’ trong dạng 1, hoặc là ‘comp’ trong dạng 2.

Tham số:

- first, last: giống như trong hàm “sort”
- value: phần tử cần tìm kiếm.
- comp: giống như trong hàm “sort”

Ví dụ: xem ví dụ ở phần upper_bound.
Độ phức tạp: Khoảng log(last-first).

5.2. upper_bound:

Dạng 1:

```
template <class ForwardIterator, class T>
  ForwardIterator upper_bound ( ForwardIterator first, ForwardIterator last,
                                const T& value );
```

Dạng 2:

```
template <class ForwardIterator, class T, class Compare>
  ForwardIterator upper_bound ( ForwardIterator first, ForwardIterator last,
                                const T& value, Compare comp );
```

Ý nghĩa:

- Trả về con trỏ đến phần tử đầu tiên trong đoạn [first,last) (đã sắp xếp) mà lớn hơn ‘value’. Phép toán so sánh mặc định là toán tử ‘<’ trong dạng 1, hoặc là ‘comp’ trong dạng 2.

Tham số:

- first, last: giống như trong hàm "sort"
- value: phần tử cần tìm kiếm.
- comp: giống như trong hàm "sort"

Ví dụ:

```
// lower_bound/upper_bound example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
  vector<int>::iterator low,up;

  sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

  low=lower_bound (v.begin(), v.end(), 20); //          ^
  up= upper_bound (v.begin(), v.end(), 20); //                   ^

  cout << "lower_bound at position " << int(low- v.begin()) << endl;
  cout << "upper_bound at position " << int(up - v.begin()) << endl;

  return 0;
}
```

Output:

```
lower_bound at position 3
upper_bound at position 6
```

Độ phức tạp: Khoảng log(last-first).

5.3. binary_search:

Dạng 1:

```
template <class ForwardIterator, class T>
  bool binary_search ( ForwardIterator first, ForwardIterator last,
                       const T& value );
```

Dạng 2:

```
template <class ForwardIterator, class T, class Compare>
  bool binary_search ( ForwardIterator first, ForwardIterator last,
                       const T& value, Compare comp );
```

Ý nghĩa:

- Kiểm tra xem giá trị value có tồn tại trong đoạn đã sắp xếp không.
- Trả về 'true' nếu có một phần tử trong nửa đoạn [first,last) bằng 'value', ngược lại trả về 'false'. Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2. Một 'value' a được coi là tương đương (bằng) 'value' b khi: (!(a<b) && !(b<a)) hoặc (!comp(a,b) && !comp(b,a)).

Tham số:

- first, last: giống như trong hàm "sort"
- value: phần tử cần tìm kiếm.
- comp: giống như trong hàm "sort"

Ví dụ:

```
// binary_search example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

int main () {
  int myints[] = {1,2,3,4,5,4,3,2,1};
  vector<int> v(myints,myints+9);                         // 1 2 3 4 5 4 3
2 1

  // sử dụng toán tử so sánh mặc định '<':
  sort (v.begin(), v.end());

  cout << "looking for a 3... ";
  if (binary_search (v.begin(), v.end(), 3))
    cout << "found!\n"; else cout << "not found.\n";

  // sử dụng hàn myfunction để so sánh:
  sort (v.begin(), v.end(), myfunction);

  cout << "looking for a 6... ";
  if (binary_search (v.begin(), v.end(), 6, myfunction))
    cout << "found!\n"; else cout << "not found.\n";

  return 0;
}
```

Output:

```
looking for a 3... found!
looking for a 6... not found.
```

Độ phức tạp: Khoảng log(last-first).

6. Trộn (thực hiện trên đoạn các phần tử đã sắp xếp):

6.1. Merge:

Dạng 1:

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator merge ( InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result );
```

Dạng 2:

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
  OutputIterator merge ( InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp );
```

Ý nghĩa:

- Kết hợp các phần tử đã được sắp xếp trong đoạn [first1,last1) và [first2,last2) vào đoạn mới bắt đầu bằng *'result'* với các phần tử đã được sắp xếp. Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.

Tham số:

- first1, last1, first2, last2: giống như first và last của hàm 'sort'

- result: trỏ đến phần tử đầu tiên của đoạn kết quả. Chiều dài của đoạn phần tử này bằng tổng chiều dài của 2 đoạn cần 'merge'.
- comp: giống như hàm 'sort'

Ví dụ:

```
// merge algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);
  vector<int>::iterator it;

  sort (first,first+5);
  sort (second,second+5);
  merge (first,first+5,second,second+5,v.begin());

  cout << "The resulting vector contains:";
  for (it=v.begin(); it!=v.end(); ++it)
    cout << " " << *it;

  cout << endl;

  return 0;
}
```

Output:

```
The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
```

Độ phức tạp: tổng số phần tử của hai đoạn đem trộn.

7. Hàng đợi ưu tiên:

7.1. push_heap:

Dạng 1:

```
template <class RandomAccessIterator>
  void push_heap ( RandomAccessIterator first, RandomAccessIterator last );
```

Dạng 2:

```
template <class RandomAccessIterator, class Compare>
  void push_heap ( RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp );
```

Ý nghĩa:

- Đẩy phần tử vào heap.
- Ban đầu có một heap là nửa đoạn [first,last-1), hàm này sẽ đẩy phần tử (last-1) vào vị trí tương ứng của nó.
- Đoạn phần tử là heap có thể xây dựng bằng cách gọi hàm "make_heap" (trình bày ở dưới), sau khi tạo ra, các tính chất của heap sẽ được giữ nguyên bằng cách sử dụng "push_heap" và "pop_heap" tương ứng để bổ sung và loại bỏ phần tử.
- Cần phải nói thêm rằng là: tính chất của heap đó là phần tử đầu tiên luôn có độ ưu tiên lớn nhất (theo phép so sánh), phần tử được 'pop' luôn là phần tử có độ ưu tiên lớn nhất, mỗi khi 'push' vào, phần tử sẽ được đẩy vào chỗ mà phù hợp với độ ưu tiên của nó (xem thêm cấu trúc dữ liệu heap để hiểu rõ). Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.

Tham số:

- first, last: giống như trong hàm "sort"
- comp: giống như trong hàm "sort"

Ví dụ: (Xem ở hàm sort_heap)
Độ phức tạp: log (last – first).

7.2. pop_heap:

Dạng 1:

```
template <class RandomAccessIterator>
  void pop_heap ( RandomAccessIterator first, RandomAccessIterator last );
```

Dạng 2:

```
template <class RandomAccessIterator, class Compare>
  void pop_heap ( RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp );
```

Ý nghĩa:

- Loại bỏ phần tử có độ ưu tiên lớn nhất ra khỏi heap (chính là phần tử đầu tiên của đoạn phần tử đã là heap). Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.

Tham số:

- first, last: giống như trong hàm "sort"
- comp: giống như trong hàm "sort"

Ví dụ: (Xem ở hàm sort_heap)
Độ phức tạp: Khoảng (2*log(last-first))

7.3. make_heap:

Dạng 1:

```
template <class RandomAccessIterator>
  void make_heap ( RandomAccessIterator first, RandomAccessIterator last );
```

Dạng 2:

```
template <class RandomAccessIterator, class Compare>
  void make_heap ( RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp );
```

Ý nghĩa:

- Tạo heap từ đoạn phần tử bằng cách sắp xếp lại các phần tử trong nửa đoạn [first,last) theo quy tắc để chúng tạo thành một heap. Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.
- Bên trong, một heap là một cây mà mỗi nút liên kết đến các phần tử có độ ưu tiên không lớn hơn nó.

Tham số:

- first, last: giống như trong hàm "sort"
- comp: giống như trong hàm "sort"

Ví dụ: (Xem ở hàm sort_heap)
Độ phức tạp: Khoảng (3*(last-first))

7.4. sort_heap:

Dạng 1:

```
template <class RandomAccessIterator>
  void sort_heap ( RandomAccessIterator first, RandomAccessIterator last );
```

Dạng 2:

```
template <class RandomAccessIterator, class Compare>
  void sort_heap ( RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp );
```

Ý nghĩa:

- Sắp xếp lại các phần tử của heap (không phải theo độ ưu tiên mà là theo thứ tự các phần tử - giống như săp xếp thông thường). Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.

Tham số:

- first, last: giống như trong hàm "sort"
- comp: giống như trong hàm "sort"

Ví dụ:

```
// range heap example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[] = {10,20,30,5,15};
  vector<int> v(myints,myints+5);
  vector<int>::iterator it;

  make_heap (v.begin(),v.end());
  cout << "initial max heap   : " << v.front() << endl;

  pop_heap (v.begin(),v.end()); v.pop_back();
  cout << "max heap after pop : " << v.front() << endl;

  v.push_back(99); push_heap (v.begin(),v.end());
  cout << "max heap after push: " << v.front() << endl;

  sort_heap (v.begin(),v.end());

  cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++) cout << " " << v[i];

  cout << endl;

  return 0;
}
```

Output:

```
initial max heap   : 30
max heap after pop : 20
max heap after push: 99
final sorted range : 5 10 15 20 99
```

Độ phức tạp: Khoảng NlogN (với N = last – first)

8. Min / Max:

8.1. min:

Dạng 1:

```
template <class T> const T& min ( const T& a, const T& b );
```

Dạng 2:

```
template <class T, class Compare>
  const T& min ( const T& a, const T& b, Compare comp );
```

Ý nghĩa:

- Trả về phần tử nhỏ hơn trong hai phần tử. Nếu hai phần tử bằng nhau, trả về phần tử a. Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.

Tham số:

- a,b: hai phần tử so sánh
- comp: giống như trong hàm "sort"

Ví dụ:

```
// min example
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
  cout << "min(1,2)==" << min(1,2) << endl;
  cout << "min(2,1)==" << min(2,1) << endl;
  cout << "min('a','z')==" << min('a','z') << endl;
  cout << "min(3.14,2.72)==" << min(3.14,2.72) << endl;
  return 0;
}
```

Output:

```
min(1,2)==1
min(2,1)==1
min('a','z')==a
min(3.14,2.72)==2.72
```

8.2. max:

Dạng 1:

```
template <class T> const T& max ( const T& a, const T& b );
```

Dạng 2:

```
template <class T, class Compare>
  const T& max ( const T& a, const T& b, Compare comp );
```

Ý nghĩa:

- Trả về phần tử lớn hơn trong hai phần tử. Nếu hai phần tử bằng nhau, trả về phần tử a. Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.

Tham số:

- a,b: hai phần tử so sánh
- comp: giống như trong hàm "sort"

Ví dụ:

```
// max example
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
  cout << "max(1,2)==" << max(1,2) << endl;
  cout << "max(2,1)==" << max(2,1) << endl;
  cout << "max('a','z')==" << max('a','z') << endl;
```

```
  cout << "max(3.14,2.73)==" << max(3.14,2.73) << endl;
  return 0;
}
```

Output:

```
max(1,2)==2
max(2,1)==2
max('a','z')==z
max(3.14,2.73)==3.14
```

8.3. next_permutation:

Dạng 1:

```
template <class BidirectionalIterator>
  bool next_permutation (BidirectionalIterator first,
                         BidirectionalIterator last );
```

Dạng 2:

```
template <class BidirectionalIterator, class Compare>
  bool next_permutation (BidirectionalIterator first,
                         BidirectionalIterator last, Compare comp);
```

Ý nghĩa:

- Biến đổi đoạn phần tử về hoán vị kế tiếp của nó theo thứ tự từ điển. Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.
- Hàm sẽ trả về 'true' nếu tìm được hoán vị tiếp theo (tức là chưa phải là hoán vị cao nhất theo thứ tự từ điển), ngược lại là 'false'

Tham số:

- first, last: giống như trong hàm "sort".
- comp: giống như trong hàm "sort".

Ví dụ:

```
// next_permutation
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
  int myints[] = {1,2,3};

  cout << "The 3! possible permutations with 3 elements:\n";

  sort (myints,myints+3);

  do {
    cout << myints[0] << " " << myints[1] << " " << myints[2] << endl;
  } while ( next_permutation (myints,myints+3) );

  return 0;
}
```

Output:

```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Độ phức tạp: Số các phép so sánh, khoảng (last-first).

8.4. prev_permutation:

Dạng 1:

```
template <class BidirectionalIterator>
  bool prev_permutation (BidirectionalIterator first,
                         BidirectionalIterator last );
```

Dạng 2:

```
template <class BidirectionalIterator, class Compare>
  bool prev_permutation (BidirectionalIterator first,
                         BidirectionalIterator last, Compare comp);
```

Ý nghĩa:

- Biến đổi đoạn phần tử về hoán vị trước đó của nó theo thứ tự từ điển. Phép toán so sánh của đoạn phần tử mặc định là toán tử '<' trong dạng 1, hoặc là 'comp' trong dạng 2.
- Hàm sẽ trả về 'true' nếu tìm được hoán vị trước đó (tức là chưa phải là hoán vị thấp nhất theo thứ tự từ điển), ngược lại là 'false'

Tham số:

- first, last: giống như trong hàm "sort".
- comp: giống như trong hàm "sort".

Ví dụ:

```
// prev_permutation
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
  int myints[] = {1,2,3};

  cout << "The 3! possible permutations with 3 elements:\n";

  sort (myints,myints+3);
  reverse (myints,myints+3);

  do {
    cout << myints[0] << " " << myints[1] << " " << myints[2] << endl;
  } while ( prev_permutation (myints,myints+3) );

  return 0;
}
```

Output:

```
The 3! possible permutations with 3 elements:
3 2 1
3 1 2
2 3 1
2 1 3
1 3 2
1 2 3
```

Độ phức tạp: Số các phép so sánh, khoảng (last-first).

9. Một số bài tập áp dụng:

Bài 1: Hoán vị kế tiếp (Link http://www.spoj.pl/PTIT/problems/BCNEPER/):

Trong bài này, bạn hãy viết chương trình nhận vào một chuỗi (có thể khá dài) các ký tự số và đưa ra màn hình hoán vị kế tiếp của các ký tự số đó (với ý nghĩa là hoán vị có giá trị lớn hơn tiếp theo nếu ta coi chuỗi đó là một giá trị số nguyên).

Chú ý: Các ký tự số trong dãy có thể trùng nhau.

Ví dụ:

123 -> 132

279134399742 -> 279134423799

Cũng có trường hợp sẽ không thể có hoán vị kế tiếp. Ví dụ như khi đầu vào là chuỗi 987.

Dữ liệu vào

Dòng đầu tiên ghi số nguyên t là số bộ test ($1 \le t \le 1000$). Mỗi bộ test có một dòng, đầu tiên là số thứ tự bộ test, một dấu cách, sau đó là chuỗi các ký tự số, tối đa 80 phần tử.

Dữ liệu ra

Với mỗi bộ test hãy đưa ra một dòng gồm thứ tự bộ test, một dấu cách, tiếp theo đó là hoán vị kế tiếp hoặc chuỗi "BIGGEST" nếu không có hoán vị kế tiếp.

Example

```
Input:

3

1 123

2 279134399742

3 987

Output:

1 132

2 279134423799

3 BIGGEST
```

Hướng dẫn:

- Bài này có thể sử dụng thuật toán sinh hoán vị kế tiếp. Nhưng cách đơn giản nhất là áp dụng hàm next_permutation.
- Chương trình mẫu:

```
#include <iostream>
#include <string>
using namespace std;
string nextPermutation(string s) {
	if (next_permutation(s.begin(),s.end())) return s;
	else return "BIGGEST";
}
main() {
	long nTest,test;
	string s;
	cin>>nTest;
	while (nTest) {
		cin>>test>>s;
		printf("%d ",test);
		cout<<nextPermutation(s)<<endl;
		nTest--;
	}
}
```

Bài tập tương tự: http://www.spoj.pl/PTIT/problems/BCPERMU/

Liệt kê hoán vị của n phần tử của một tập gồm các số từ 1->n.

Input

Dòng duy nhất chứa số n (1<=n<=8)

Output

Các hoán vị sắp xếp theo thứ tự từ điển tăng dần.

Example

```
Input:
3

Output:
123
132
213
231
312
321
```

Bài 2:

Cho N (1<=N<=30000) sinh viên, mỗi sinh viên có 3 thông tin là: mã sinh viên, điểm học tập và điểm rèn luyện.

Yêu cầu: Sắp xếp các sinh viên theo thứ tự tăng dần của điểm học tập, nếu 2 sinh viên có điểm học tập bằng nhau thì sắp xếp theo điểm rèn luyện tăng dần.

Dữ liệu:

- Dòng 1: chứa số N
- Dòng 2..N+1: mỗi dòng chứa 3 số nguyên lần lượt là mã sinh viên, điểm học tập và rèn luyện.

Kết quả: Danh sách sau khi đã sắp xếp theo yêu cầu:

- N dòng, mỗi dòng chứa 3 số nguyên mã sinh viên, điểm học tập, điểm rèn luyện.

Lưu ý: Các số trong input và output trên cùng 1 dòng cách nhau bởi dấu cách.

Ví dụ:

```
Input:
3
1 10 9
2 8 7
3 8 8
Output:
2 8 7
3 8 8
1 10 9
```

Hướng dẫn:

- Với N<=30000 bạn cần sử dụng thuật toán sắp xếp có độ phức tạp cỡ khoảng NlogN, nên ta có thể áp dụng hàm *sort* hoặc cấu trúc *heap* trong thư viện algorithm. Khi sử dụng cần viết lại hàm so sánh cho đúng với cách sắp xếp của bài này.
- Chương trình mẫu:
    - Sử dụng hàm *sort*:

```
#include <iostream>
#include <algorithm>
#define maxn 30000
using namespace std;
struct SV {
	long maSV,DHT,DRL;
};
bool comp(SV a,SV b) {
	return (a.DHT<b.DHT || (a.DHT == b.DHT && a.DRL < b.DRL));
}
main() {
	SV a[maxn];
	long n;
	cin >> n;
	for (long i=0;i<n;i++) {
		cin >> a[i].maSV >> a[i].DHT >> a[i].DRL;
	}
	sort(a,a+n,comp);
	for (long i=0;i<n;i++) {
		cout << a[i].maSV << " " << a[i].DHT << " " << a[i].DRL << endl;
	}
}
```

    - Sử dụng heap : Hàm so sánh có thay đổi một chút, vì phần tử ở vị trí *pop* của heap luôn có độ ưu tiên cao nhất (tức là nếu sử dụng phép toán thấp hơn, thì *pop* luôn là phần tử lớn nhất, trong bài này mỗi lần cần lấy ra phần tử *pop* là bé nhất, nên phép toán so sánh sẽ là lớn hơn).

```
#include <iostream>
#include <algorithm>
#define maxn 30000
using namespace std;
struct SV {
	long maSV,DHT,DRL;
};
bool comp(SV a,SV b) {
	return (a.DHT>b.DHT || (a.DHT == b.DHT && a.DRL > b.DRL));
}
main() {
	SV a[maxn];
	long n;
```

```
        cin >> n;
        for (long i=0;i<n;i++) {
                cin >> a[i].maSV >> a[i].DHT >> a[i].DRL;
                if (i>0) push_heap(a,a+i,comp);
        }
        for (long i=0;i<n;i++) {
                /*phần tử thấp nhất luôn là phần tử đầu tiên (a[0]) */
                cout << a[0].maSV << " " << a[0].DHT << " " << a[0].DRL << endl;
                /*loại bỏ phần tử thấp nhất*/
                pop_heap(a,a+n-i,comp);
        }
}
```

Một số bài áp dụng:

- Bài 1: http://www.spoj.pl/PTIT/problems/BCSAPXEP/

Sắp xếp dãy tăng dần.

Input

- Dòng đầu chứa số n (số phần tử của dãy 1<=n<=1000)

- n dòng sau, mỗi dòng là 1 phần tử của dãy (giá trị tuyệt đối không quá 1000)

Output

Mỗi phần tử của dãy in trên 1 dòng, theo thứ tự tăng dần.

Example

```
Input:
3
3
2
1

Output:
1
2
3
```

- Bài 2: http://www.spoj.pl/PTIT/problems/BCTELEPH/

Cho một danh sách các số điện thoại, hãy xác định danh sách này có số điện thoại nào là phần trước của số khác hay không? Nếu không thì danh sách này được gọi là nhất quán. Giả sử một danh sách có chứa các số điện thoại sau:

- Số khẩn cấp: 911

- Số của Alice: 97625999

- Số của Bob: 91125426

Trong trường hợp này, ta không thể gọi cho Bob vì tổng đài sẽ kết nối bạn với đường dây khẩn cấp ngay khi bạn quay 3 số đầu trong số của Bob, vì vậy danh sách này là không nhất quán.

Dữ liệu vào

Dòng đầu tiên chứa một số nguyên $1 \le t \le 40$ là số lượng bộ test. Mỗi bộ test sẽ bắt đầu với số lượng số điện thoại n được ghi trên một dòng, $1 \le n \le 10000$. Sau đó là n dòng, mỗi dòng ghi duy nhất 1 số điện thoại. Một số điện thoại là một dãy không quá 10 chữ số.

Dữ liệu ra

Với mỗi bộ dữ liệu vào, in ra "**YES**" nếu danh sách nhất quán và "**NO**" trong trường hợp ngược lại.

INPUT	OUTPUT
2 3 911 97625999 91125426 5 113 12340 123440 12345 98346	**NO** **YES**

- Bài 3: http://vn.spoj.pl/problems/QBHEAP/

Cho trước một danh sách rỗng. Người ta xét hai thao tác trên danh sách đó:

Thao tác "+V" (ở đây V là một số tự nhiên <= 1000000000): Nếu danh sách đang có ít hơn 15000 phần tử thì thao tác này bổ sung thêm phần tử V vào danh sách; Nếu không, thao tác này không có hiệu lực.

Thao tác "-": Nếu danh sách đang không rỗng thì thao tác này loại bỏ tất cả các phần tử lớn nhất của danh sách; Nếu không, thao tác này không có hiệu lực

Input

Gồm nhiều dòng, mỗi dòng ghi một thao tác. Thứ tự các thao tác trên các dòng được liệt kê theo đúng thứ tự sẽ thực hiện

Output

Dòng 1: Ghi số lượng những giá trị còn lại trong danh sách.

Các dòng tiếp theo: Liệt kê những giá trị đó theo thứ tự giảm dần, mỗi dòng 1 số

Example

Input:

```
+1
+3
+2
+3
-
+4
+4
-
+2
+9
+7
+8
-
```

Output:

```
4
8
2
2
1
```

IV. THƯ VIỆN STRING C++:

- String là một kiểu đặc biệt của container, thiết kế đặc để hoạt động với các chuỗi kí tự.
- Khai báo: `#include <string>`
- Iterator: Tương tự như trong container, string cũng hỗ trợ các iterator như begin, end, rbegin, rend với ý nghĩa như trước.
- Nhập, xuất string:
  - Sử dụng toán tử >> : tương tự như trong C. Nhập đến khi gặp dấu cách.
  - Sử dụng getline: giống như gets trong C. Nhập cả một dòng kí tự (chứa cả dấu cách nếu có) cho string.
  - Xuất string: sử dụng toán tử << như bình thường.

Các hàm thành viên:

- Trong string bạn có thể sử dụng các toán tử "=" để gán giá trị cho string, hay toán tử "+" để nối hai string với nhau. Ngoài ra, khi so sánh hai string với nhau thì cũng dùng

các toán tử so sánh như '>','<' để so sánh 2 string theo thứ tự từ điển. Chức năng này khá giống với xâu ở trong ngôn ngữ pascal.

- Capacity:
  - size: trả về độ dài của string
  - length: trả về độ dài của string
  - clear : xóa string
  - empty: return true nếu string rỗng, false nếu ngược lại
- Truy cập đến phần tử:
  - operator [chỉ_số]: lấy kí tự vị trí chỉ_số của string
- Chỉnh sửa:
  - push_back: chèn kí tự vào sau string
  - insert (n,x): chèn x vào string ở trước vị trí thứ n. x có thể là string, char,...
  - erase (pos,n): xóa khỏi string "n" kí tự bắt đầu từ vị trí thứ "pos".
  - erase (iterator): xóa khỏi string phần tử ở vị trí iterator.
  - replace (pos, size, s1) : thay thế string từ vị trí "pos", số phần tử thay thế là "size" và thay bằng xâu s1.
  - swap (string_cần_đổi): đổi giá trị 2 xâu cho nhau.
- String operations:
  - c_str : chuyển xâu từ dạng string trong C++ sang string trong C.
  - substr (pos,length): return string được trích ra từ vị trí thứ "pos", và trích ra "length" kí tự.