ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

GIỚI THIỆU

Cũng giống như một chiếc ô tô, thuật toán chính là động cơ của chiếc ô tô đó, trong các ứng dụng hiện tại bất kì ứng dụng nào có sức ảnh hưởng đều liên quan đến thuật toán, ứng dụng càng nổi tiếng thì thuật toán đằng sau nó cực kì phức tạp, có thể nhắc đến các hệ thống như Facebook, Google, Amazon...

Với sức mạnh tính toán của máy tính thông thường hiện tại có thể tính toàn được hàng triệu cho đến hàng tỉ phép tính trên 1 giây nhưng như vậy vẫn là quá quá bé nhỏ so với ngành lập trình của chúng ta. Một ví dụ đơn giản thực hiện sắp xếp 1 dãy có 100.000.000 phần tử với cách sắp xếp thông thường sẽ phải thực hiện mất 1x10^16 phép tính khoảng 100 nghìn tỉ phép tính. Tính trung bình mỗi giây một máy tính sẽ tính được khoảng 1x10^8 phép tính tương đương với 100 triệu phép tính trên 1s thì khi sắp xếp một dãy 100 triệu phần tử sẽ mất khoảng 100.000.000 giây khoảng 1157 ngày ~ 3 năm. Nhưng nếu chạy trên thuật toán quiksort thì số lượng phép tính chỉ khoảng 1x10^11 phép tính thì tổng số thời gian sẽ còn khoảng 100000s tương đương với khoảng 1 ngày => các bạn đã thấy sự lợi hại của thuật toán chưa =))

CÁC KHÁI NIỆM CƠ BẢN

Phần này khá đi sâu vào toán học nếu bạn đọc không hiểu thì có thể bỏ qua:

Gọi f, g là các hàm số dương không giảm trên tập số nguyên dương (lưu ý rằng *hàm thời gian chạy* thỏa mãn điều kiện này). Ta nói rằng "$f(N)$ thuộc $O(g(N))$" (cách đọc: "f thuộc O-lớn của g") nếu tồn tại các giá trị c và N_0 thỏa mãn điều kiện sau:

$$\forall N > N_0; f(N) < c.g(N)$$

Mệnh đề trên có thể diễn dịch như sau: $f(N)$ thuộc $O(g(N))$ nếu với c nào đó toàn bộ đồ thị của hàm f nằm dưới đồ thị của hàm $c.g$. Chú ý rằng điều này có nghĩa là tốc độ tăng của hàm f không vượt quá độ tăng của hàm g. (ND: ký hiệu O-lớn là ký hiệu chỉ **tập hợp của các hàm số**, vì vậy ở đây quan hệ giữa f và $O(g(N))$ là *(phần tử) thuộc (tập hợp)*.)

Thay vì viết "$f(N)$ thuộc $O(g(N))$" ta thường viết là "$f(N) = O(g(N))$". Chú ý là dấu "=" không có tính đối xứng - viết "$O(g(N)) = f(N)$" là sai và không có ý nghĩa gì, đồng thời mệnh đề "$g(N) = O(f(N))$" cũng không phải luôn đúng (sẽ được chỉ ra ở phần sau của bài viết).

Định nghĩa trên được biết tới là ký pháp O-lớn và được sử dụng để chỉ ra cận trên của tốc độ tăng của một hàm số.

Xét hàm số $f(N) = 1.5N^2 - 0.5N$ trong ví dụ 2. Ta có thể phát biểu rằng $f(N) = O(N^2)$ (một trường hợp khả dĩ cho các hằng số là $c = 2$ và $N_0 = 0$). Điều này có nghĩa là hàm f **không tăng (tiệm cận) nhanh hơn** N^2.

Lưu ý rằng *thời gian chạy* chính xác của hàm f không cho ta câu trả lời cho câu hỏi "Chương trình trên chạy mất bao lâu trên máy tính?". Nhận định quan trọng rút ra từ ví dụ trên là *thời gian chạy* của hàm f đó là hàm bậc hai. Nếu ta tăng gấp đôi kích cỡ đầu vào, thời gian chạy sẽ tăng xấp xỉ 4 lần thời gian chạy ban đầu, không quan trọng máy tính của ta nhanh như thế nào.

Việc xác định được cận trên $O(N^2)$ của $f(N)$ cũng đưa ta tới cùng nhận định như trên - đảm bảo rằng độ tăng của hàm thời gian chạy tối đa là hàm bậc hai.

Vì vậy, chúng ta sẽ sử dụng ký pháp O-lớn để mô tả độ phức tạp *thời gian* (và đôi khi là cả *bộ nhớ*) của các thuật toán. Với thuật toán trong ví dụ 2 ta sẽ nói "Độ phức tạp thời gian của thuật toán này là $O(N^2)$" hoặc ngắn gọn hơn "Thuật toán này là $O(N^2)$".

Theo cách tương tự ta sẽ định nghĩa Ω (Omega-lớn) and Θ (Theta-lớn).

Ta nói rằng $f(N) = \Omega(g(N))$ nếu $g(N) = O(f(N))$, nói cách khác nếu f tăng nhanh hơn hoặc bằng g.

Ta nói rằng $f(N) = \Theta(g(N))$ nếu $f(N) = O(g(N))$ và $g(N) = O(f(N))$, nói cách khác nếu cả hai hàm số có độ hiệu quả xấp xỉ bằng nhau.

Dễ dàng nhận thấy là hàm Omega-lớn dùng để chỉ cận dưới và hàm Theta-lớn dùng để chỉ đánh giá chặt (cả hai cận) của một hàm số. Có những đánh giá cận khác tương tự nhưng ít phổ biến hơn.

Một vài mệnh đề sử dụng các ký pháp trên:

- $1.5N^2 - 0.5N = O(N^2)$.
- $47NlogN = O(N^2)$.
- $NlogN + 1000047N = \Theta(NlogN)$.
- Tất cả các đa thức bậc k là $O(N^k)$.
- Độ phức tạp thời gian của thuật toán trong ví dụ 2 là $\Theta(N^2)$.
- Nếu một thuật toán thuộc $O(N^2)$, nó cũng thuộc $O(N^5)$.
- Các thuật toán sắp xếp dựa trên phép so sánh đều là $\Omega(NlogN)$.
- Thuật toán sắp xếp trộn *MergeSort* chạy trên mảng gồm N phần tử thực hiện xấp xỉ $NlogN$ phép so sánh. Vì vậy độ phức tạp thời gian của *MergeSort* là $\Theta(NlogN)$. Nếu mệnh đề trước đó là đúng thì MergeSort tiệm cận thuật toán tối ưu nhất trong các thuật toán sắp xếp dựa trên phép so sánh.
- Thuật toán trong ví dụ 2 sử dụng $\Theta(N)$ bytes bộ nhớ.
- Hàm số cho biết số răng của một người ở một thời điểm xác định là $O(1)$.
- Một thuật toán quay lui đơn giản giải các bài toán trên bàn cờ vua là $O(1)$ vì cây vị trí mà thuật toán duyệt qua có kích cỡ giới hạn. (Tuy nhiên giá trị hằng số trong $O(1)$ này lại rất lớn)
- Mệnh đề "Độ phức tạp thời gian của thuật toán này tối thiểu là $O(N^2)$" là vô nghĩa. (Diễn dịch: "Giá trị tối thiểu của độ phức tạp thời gian của thuật toán này tối đa là xấp xỉ hàm bậc hai"). Phát biểu đúng là: "Độ phức tạp thời gian của thuật toán này là $\Omega(N2)$."

Khi nói về độ phức tạp thời gian/bộ nhớ của một thuật toán, thay vì sử dụng ký pháp Theta-lớn $\Theta(f(N))$ ta có thể đơn giản chỉ ra tên của lớp hàm chứa hàm f. Ví dụ với $f(N) = \Theta(N)$, ta gọi thuật toán đó là *tuyến tính*. Một vài ví dụ khác:

- $f(N) = \Theta(logN)$: hàm log
- $f(N) = \Theta(N^2)$: hàm bậc 2
- $f(N) = \Theta(N^3)$: hàm bậc 3
- $f(N) = O(N^k)$: hàm đa thức
- $f(N) = \Omega(2^N)$: hàm mũ

Xác định thời gian chạy dựa vào đánh giá cận trên

Với hầu hết các thuật toán thường gặp trong thực tế, giá trị hằng số của O (hoặc Θ) thường là khá nhỏ. Nếu một thuật toán là $\Theta(N^2)$, độ phức tạp chính xác là vào khoảng $10N^2$ chứ không phải 10^7N^2.

Nói cách khác: nếu hằng số là lớn thì thường là các hằng số đó có liên quan tới các đại lượng có sẵn trong đề bài. Trong trường hợp này cách làm thông thường là gán một tên gọi cho hằng số đó và thêm nó vào đánh giá độ phức tạp theo ký pháp, thay vì bỏ qua như ta đã làm với số 1.5 ở ví dụ 2.

Ví dụ: bài toán yêu cầu đếm số lần xuất hiện của mỗi loại ký tự trong một chuỗi độ dài N ký tự. Thuật toán đơn giản nhất là duyệt cả chuỗi một lần cho mỗi loại ký tự. Kích cỡ bảng chữ cái không thay đổi (ví dụ nhiều nhất là 255 ký tự trong ngôn ngữ C), vì vậy thuật toán là tuyến tính với N. Mặc dù vậy, ta nên viết độ phức tạp thời gian là $\Theta(|S| \times N)$, trong đó S là bảng chữ cái được sử dụng (Lưu ý là có một thuật toán tốt hơn giải bài toán này trong $\Theta(|S| + N)$.

Trong một kỳ thi trên TopCoder, một thuật toán thực thi 1 000 000 000 phép nhân hiếm khi chạy trong giới hạn thời gian cho phép. Thực tế này cộng với quan sát ở trên và một vài kinh nghiệm với các bài toán trên TopCoder giúp ta tổng kết bảng sau:

complexity	maximum N
$\Theta(N)$	100 000 000
$\Theta(NlogN)$	40 000 000
$\Theta(N^2)$	10 000
$\Theta(N^3)$	500
$\Theta(N^4)$	90
$\Theta(2^N)$	20
$\Theta(N!)$	11

MỘT SỐ KHÁI NIỆM QUAN TRỌNG

1. Thời gian thực hiện chương trình

Thời gian thực hiện một chương trình là một hàm của kích thước dữ liệu vào, ký hiệu T(n) trong đó n là kích thước (độ lớn) của dữ liệu vào. Chương trình tính tổng của n số có thời gian thực hiện là T(n) = cn trong đó c là một hằng số. Thời gian thực hiện chương trình là một hàm không âm, tức là T(n) ≥ 0 với mọi n ≥ 0.

2. Đơn vị đo thời gian thực hiện

Đơn vị của T(n) không phải là đơn vị đo thời gian bình thường như giờ, phút giây... mà thường được xác định bởi số các lệnh được thực hiện trong một máy tính lý tưởng. Khi ta nói thời gian thực hiện của một chương trình là T(n) = Cn thì có nghĩa là chương trình ấy cần Cn chỉ thị thực thi. - Kí hiệu O lớn (big-O) thể hiện độ phức tạp của thuật toán bất kì. Một thuật toán với f=T(n), thì ta nói nó có độ phức tạp là O(f).

3. Cách tính độ phức tạp của giải thuật

- Quy tắc bỏ hằng số: Nếu đoạn chương trình P có thời gian thực hiện T(n) = O(c1.f(n)) với c1 là một hằng số dương thì có thể coi đoạn chương trình đó có độ phức tạp tính toán là O(f(n)).
- Quy tắc cộng – lấy max: Nếu T1(n) và T2(n) là thời gian thực hiện của hai đoạn chương trình P1 và P2; và T1(n)=O(f(n)), T2(n)=O(g(n)) thì thời gian thực hiện của đoạn hai chương trình đó nối tiếp nhau là T(n)=O(max(f(n),g(n)))

VD: Lệnh gán x=15 tốn một hằng thời gian hay O(1), Lệnh đọc dữ liệu cout<<x; tốn một hằng thời gian hay O(1). Vậy thời gian thực hiện cả hai lệnh trên nối tiếp nhau là O(max(1,1))=O(1)

- Quy tắc nhân: Nếu T1(n) và T2(n) là thời gian thực hiện của hai đoạn chương trình P1, P2 và T1(n) = O(f(n)), T2(n)=O(g(n)) thì thời gian thực hiện của đoạn hai đoạn chương trình đó lồng nhau là T(n) = O(f(n).g(n)) VD: Chương trình vẽ hình chữ nhật có chiều dài là x, chiều rộng là y bằng kí hiệu *'': với mỗi một hàng phải thực hiện y lệnh in cout<<""*; và lặp lại x lần. Vậy thời gian thực hiện đoạn chương trình trên là O(x.y).
- Quy tắc tổng quát để phân tích một chương trình

– Thời gian thực hiện của mỗi lệnh gán, READ, WRITE là O(1).

– Thời gian thực hiện của một chuỗi tuần tự các lệnh được xác định bằng qui tắc cộng. Như vậy thời gian này là thời gian thi hành một lệnh nào đó lâu nhất trong chuỗi lệnh.

– Thời gian thực hiện cấu trúc IF là thời gian lớn nhất thực hiện lệnh sau THEN hoặc sau ELSE và thời gian kiểm tra điều kiện. Thường thời gian kiểm tra điều kiện là O(1).

– Thời gian thực hiện vòng lặp là tổng (trên tất cả các lần lặp) thời gian thực hiện thân vòng lặp. Nếu thời gian thực hiện thân vòng lặp không đổi thì thời gian thực hiện vòng lặp là tích của số lần lặp với thời gian thực hiện thân vòng lặp.

– Thời gian thực hiện giải thuật đệ quy với một số nguyên n là O(log(n))

Một số ví dụ:

Tính tổng các số nguyên từ 1 -> n

Bài này ai dùng công thức thì 1 dòng là ra: n*(n+1)/2. Giải thuật này có độ phức tạp là O(1) (1 phép toán). Với các bạn dùng vòng lặp từ 1 -> n để tính tổng, độ phức tạp là O(n). Với n bằng 1 tỷ, tương đương bạn thực hiện 1 tỷ lần phép toán cộng, gây tốn quá nhiều thời gian.

Bài toán kiểm tra số nguyên tố.

Các bạn chạy để kiểm tra từ 1->n, độ phức tạp là O(n). Các bạn chạy từ 1->sqrt(n) (căn bậc 2 của n) thì đã giảm rất nhiều phép toán, nếu bạn nào còn tăng bước nhảy lên bằng 2 (kiểm tra có chia hết cho 2,3, 5, 7, 9, 11, ... thay vì 2,3,4,5,6, ...) thì số phép toán lại giảm thêm nữa. Do đó, ngay từ bài số nguyên tố,

việc sử dụng vòng lặp để kiểm tra giúp ta tối ưu cực nhiều. Bạn có thể thử bài này với số n cực lớn và gọi đi gọi lại nhiều lần để đo độ chênh lệch thời gian.

Bài toán kiểm tra số nguyên tố.

Các bạn chạy để kiểm tra từ 1->n, độ phức tạp là O(n). Các bạn chạy từ 1->sqrt(n) (căn bậc 2 của n) thì đã giảm rất nhiều phép toán, nếu bạn nào còn tăng bước nhảy lên bằng 2 (kiểm tra có chia hết cho 2,3, 5, 7, 9, 11, ... thay vì 2,3,4,5,6, ...) thì số phép toán lại giảm thêm nữa. Do đó, ngay từ bài số nguyên tố, việc sử dụng vòng lặp để kiểm tra giúp ta tối ưu cực nhiều. Bạn có thể thử bài này với số n cực lớn và gọi đi gọi lại nhiều lần để đo độ chênh lệch thời gian.

CÁCH TÍNH THỜI GIAN CHẠY CHƯƠNG TRÌNH

- Thời gian chạy của một chương trình chính là thời gian từ lúc bắt đầu cho tới khi chương trình kết thúc. Thông thường ta thường đo thời gian của một hàm, một thuật toán nào đó. Chủ yếu là dùng để đánh giá độ phức tạp, tốc độ và khả năng tối ưu của chương trình.

- Biết được thời gian xử lý của thuật toán sẽ giúp người lập trình đưa ra được các phương pháp xử lý cần thiết. Có thể chúng ta sẽ làm giảm được thời gian chạy của chương trình, tối ưu thuật toán của mình một cách tốt nhất.

- Ngôn ngữ C/C++ có cung cấp một thư viện thời gian **time.h** cung hàm **clock()** giúp trả về thời gian hiện tại. Dựa vào đặc điểm này ta có thể tính được thời gian từ khi bắt đầu đến khi kết thúc thuật toán.

- Ý tưởng:

+ Khai báo biến để lấy thời gian lúc bắt đầu: start = clock();

+ Khai báo biến thứ 2 để lấy thời gian lúc kết thúc: end = clock();

+ Khai báo thêm một biến time_used dùng để lưu thời gian chạy

+ Thời gian chạy sẽ bằng lúc kết thúc trừ lúc bắt đầu.

- Công thức:

time_used = (end-start)/CLOCK_PER_SEC;

CLOCK_PER_SEC là một tích tắc trong một giây. Hàm clock() trả về thời gian là tích tắc cho nên mới có công thức bên trên

Cú pháp khai báo biến: **clock_t <tên biến>**

clock_t là một kiểu dữ liệu được định nghĩa sẵn trong thư viện time.h

Ví dụ: đo thời gian chạy của thuật toán tìm n!:

Code c++:

```
#include<iostream>
#include<time.h> // Thư viện thời gian
using namespace std;
// tinh N giai thua su dung de quy
int factorial(int n){
if(n==1)
return 1;
return(n*factorialmain(n-1));
}
void solve(int n){
cout<<"Nhap n: ";
cin>>n;
clock_t start, end; // Khai báo biến thời gian
double time_use; // Thời gian sử dụng
start = clock(); // Lấy thời gian trước khi thực hiện thuật toán
cout<<"Giai thua cua "<<n<<" la: "<<factorial(n)<<endl; // Thực hiện thuật toán
end = clock(); // lấy thời gian sau khi thực hiện
time_use = (double)(end - start) / CLOCKS_PER_SEC; //Tính thời gian sử dụng
cout<<"Thoi gian chay factorial(n): "<<time_use;
}
int main(){
int n;
solve(n);
}
```

Kết quả sau khi chạy chương trình:

NHẬP XUẤT TRONG C++

Việc nhập xuất là việc cơ bản nhất trong competitive programming, bất kể một bài toán nào đều có input và output. Mình xin giới thiệu với các bạn các cách nhập xuất hay dùng

Dòng lệnh chuẩn để vào ra file khi đề có yêu cầu sử dụng vào ra bằng file

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

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

```
int main()
{
freopen("INPUT.TXT", "r", stdin);
freopen("OUTPUT.TXT", "w", stdout);
int x;
cin >> x;
cout << x;
return 0;
}
```

Ví dụ 1:

BÀI 1. HÁ MIỆNG CHỜ SUNG

Có lẽ ai cũng biết ý nghĩa của thành ngữ "Há miệng chờ sung" là gì? Truyện kể rằng xưa kia có một anh chàng lười, cả ngày chỉ muốn ngồi một chỗ, không làm gì và cũng không di chuyển đi đâu cả. Một hôm anh ta quyết định ngồi cả ngày dưới gốc cây sung, cứ quả sung nào rơi xuống gần anh ta thì anh ta với tay nhặt lấy để ăn, còn nếu xa hơn thì anh ta mặc kệ.

Yêu cầu: Hãy cho biết ngày hôm đó chàng lười kể trên ăn được bao nhiêu quả sung?

***Dữ liệu vào*:** Từ bàn phím gồm:

- Dòng đầu tiên là hai số nguyên dương N và K (N là số lượng quả sung rụng, K là độ dài cánh tay của anh chàng lười; $N, K \leq 1000$).
- Dòng thứ 2 gồm hai số nguyên x, y là tọa độ vị trí ngồi của chàng lười $(-1000 \leq x, y \leq 1000)$.
- N dòng sau, mỗi dòng chứa hai số nguyên là tọa độ của quả sung rụng xuống (các giá trị nhập vào có trị tuyệt đối nhỏ hơn hoặc bằng 1000).

Dữ liệu ra: Đưa ra màn hình một số nguyên duy nhất là số lượng quả sung mà chàng lười ăn được.

Ví dụ:

BÀN PHÍM	MÀN HÌNH
5 3 2 1 1 0 2 1 3 -1 4 8 5 1	4

- Lưu ý nhỏ là khi đọc đề bài các bạn nên đặt biến giống với yêu cầu đề bài.

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

int main()
{
    int n,k;
    cin>>n>>k;
    int x,y;
    cin>>x>>y;
    int a[n+1],b[n+1];
    for(int i=0;i<n;i++)
    {
        cin>>a[i]>>b[i];
        //hoặc các bạn có thể nhập như sau
        //int u,v;
        //cin>>u>>v;
    }
}
```

Ví dụ 2:

Đề bài này yêu cầu vào ra bằng file nên chúng ta sẽ sử dụng cách đọc ghi giống như phần đầu

BÀI 2. GẶP MẶT HỌC SINH XUẤT SẮC

Để có số lượng học sinh mời dự buổi gặp mặt những học sinh tiêu biểu có thành tích xuất sắc trong các kỳ thi học sinh giỏi cấp tỉnh năm học 2016-2017. Khi cô hiệu trưởng tổng hợp danh sách thí sinh đạt giải từ các kỳ thi thấy rằng có nhiều học sinh tham gia các kỳ thi khác nhau và đạt giải. Em hãy giúp cô hiệu trưởng thực hiện nhiệm vụ sau:

Từ danh sách các bạn học sinh đạt giải qua các kỳ thi hãy chọn ra danh sách rút gọn và sắp xếp danh sách theo thứ tự ưu tiên là tổng số giải mà học sinh đạt được.

Dữ liệu vào: File văn bản GAPMAT.INP gồm n dòng là họ và tên của học sinh qua các kỳ thi ($1 \leq n \leq 255$).

Dữ liệu ra: File văn bản GAPMAT.OUT gồm danh sách họ và tên học sinh đã rút gọn và sắp xếp theo thứ tự ưu tiên như trên.

Ví dụ:

GAPMAT.INP	GAPMAT.OUT
NguyenVanCong HoangTrongThuy TranDaiNghia VuVietXuan HoangCam HoangTrongThuy TranDaiNghia HoangTrongThuy	HoangTrongThuy TranDaiNghia NguyenVanCong VuVietXuan HoangCam

```
#include<iostream>
using namespace std;
int main()
{
    freopen("GAPMAT.INP", "r", stdin);
    freopen("GAPMAT.OUT", "w", stdout);

    string s;
    while(cin>>s)
    {
        //code
    }
}
```

Ví dụ 3:

BÀI 3. CHIA NHÓM

Để chuẩn bị cho năm học mới 2016-2017, chính quyền địa phương xây cho các cháu mầm non một lớp học mới ở trên đồi bằng nguồn vốn xã hội hóa. Cùng chung tay góp sức, Đoàn trường huy động đoàn viên chuyển gạch từ chân đồi lên địa điểm xây trường mới, các bạn đoàn viên có dáng vóc và thể lực khác nhau nên số lượng viên gạch có thể chuyển cũng khác nhau. Để tạo động lực thi đua, Bí thư Đoàn có ý định chia các đoàn viên thành hai nhóm với điều kiện là tổng số viên gạch mà mỗi bạn có thể mang và số đoàn viên trong nhóm của bạn so với nhóm kia là nhỏ nhất. Em hãy giúp Bí thư Đoàn thực hiện yêu cầu trên.

Dữ liệu vào: Từ bàn phím mảng A gồm n phần tử ($2 \le n \le 256$), phần tử thứ i là bạn đoàn viên thứ i và số lượng viên gạch chuyển là a_i ($1 \le a_i \le 10^6$).

Dữ liệu ra: In ra màn hình, gồm 2 dòng:

- Dòng thứ nhất: giá trị đầu tiên là tổng số viên gạch, các giá trị tiếp theo là danh sách đoàn viên của các bạn nhóm 1.

- Dòng thứ hai: giá trị đầu tiên là tổng số viên gạch, các giá trị tiếp theo là danh sách đoàn viên của các bạn nhóm 2.

(các giá trị cách nhau một khoảng trắng)

Ví dụ:

BÀN PHÍM	MÀN HÌNH	GIẢI THÍCH
3 5 6 3 5 7 8 4 3	22 1 2 3 7 22 4 5 6 8 9	i = 1 2 3 4 5 6 7 8 9 a_i = **3 5 6** 3 5 7 **8** 4 3

```
#include<iostream>
using namespace std;
int main()
{
    int n;
    while(cin>>n)
    {
        //code here
    }
}
```

Ví dụ 4:

BÀI 2: TAM GIÁC VUÔNG

Được học về Định lí Pytago đảo, Tèo biết rằng "*Nếu một tam giác có bình phương của một cạnh bằng tổng bình phương các cạnh còn lại thì tam giác đó là tam giác vuông*". Tèo viết lên giấy N số nguyên dương đôi một khác nhau $a_1, a_2, ..., a_n$ và muốn chọn một bộ ba số (a_i, a_j, a_k) với $a_i < a_j < a_k$ và $i \neq j \neq k$ sao cho ba số này là độ dài ba cạnh của một tam giác vuông.

- Yêu cầu: Hỏi Tèo có bao nhiêu cách chọn một bộ số như trên.

- Dữ liệu vào: Từ tệp văn bản TGV.INP gồm:

+ Dòng 1: Ghi số nguyên dương N $(3 \leq N \leq 1000)$.

+ Dòng 2: Ghi N số nguyên dương $a_1, a_2, ..., a_n$ đôi một khác nhau $(a_i \leq 10^5)$.

- Dữ liệu ra: Ghi ra tệp văn bản TGV.OUT số nguyên duy nhất là kết quả tìm được.

- Ví dụ:

TGV.INP	TGV.OUT
6 1 2 3 5 7 4	1

```
#include<iostream>
using namespace std;
int main()
{
    freopen("TGV.INP", "r", stdin);
    freopen("TGV.OUT", "w", stdout);
    int n;
    cin>>n;
    int a[n+1];
    for(int i=0;i<n;i++)
        cin>>a[i];
}
```

Ví dụ 5:

Bài 4: CHIA QUÀ (6,0 điểm)

Thầy Tuyên có N hộp quà được xếp thành một hàng ngang, hộp quà thứ i có A_i chiếc kẹo (*với* $1 \le i \le N$, $A_i \le 10^9$).

Nhân dịp tết trung thu, thầy Tuyên chia số hộp quà đó cho hai anh em là con trai và con gái của thầy. Mỗi người trong hai anh em sẽ nhận được đúng K hộp quà liên tiếp, nếu còn dư hộp quà nào thầy sẽ cất để dùng cho dịp sau. Là người anh, bạn hãy chọn ra K hộp quà cho em và K hộp quà cho mình sao cho tổng số kẹo của cả hai anh em là lớn nhất?

Dữ liệu vào: Từ tệp CHIAQUA.INP có cấu trúc như sau:

- Dòng 1: Chứa hai số nguyên dương N, K ($2 \le N \le 10^6$, $K \le N/2$).
- Dòng 2: Chứa N số nguyên dương A_i ($A_i \le 10^9$ với $1 \le i \le N$), mỗi giá trị cách nhau một khoảng trắng.

Dữ liệu ra: Ghi vào tệp CHIAQUA.OUT một số nguyên dương duy nhất là tổng số kẹo nhiều nhất mà hai anh em nhận được.

Ví dụ:

CHIAQUA.INP	CHIAQUA.OUT
9 3 2 **6 1 5** 3 **8 2 9** 1	31

Ràng buộc:

- *Có 50% điểm ứng với $N \le 300$.*
- *Có 30% điểm ứng với $300 < N \le 5000$.*
- *Có 20% điểm ứng với $5000 < N \le 10^6$.*

----------HẾT----------

```
#include<iostream>
using namespace std;
int main()
{
    freopen("CHIAQUA.INP", "r", stdin);
    freopen("CHIAQUA.OUT", "w", stdout);
    int n,k;
    cin>>n>>k;
    int a[n+1];
    for(int i=0;i<n;i++)
        cin>>a[i];
}
```

Ví dụ 6:

BÀI 1. TÌM ĐIỂM KHÁC NHAU

Nam rất thích chơi trò chơi tìm điểm khác nhau giữa hai bức ảnh, cậu thường gặp trò chơi này trong các tờ tạp chí cũ của bố. Biết được sở thích của Nam, bố cậu đã download phiên bản điện tử của trò chơi này trên máy tính và cho Nam chơi thử. Nam nhận thấy mọi việc phức tạp hơn với trò chơi trên máy tính như: thời gian được ấn định, kích thước bức ảnh lớn dần lên sau mỗi màn chơi... Bạn hãy giúp Nam viết một chương trình để chơi trò này nhé.

Dữ liệu vào: Nhập từ bàn phím gồm

- Dòng 1: Ghi hai số nguyên M và N là số dòng và số cột của bức ảnh ($1 \le M, N \le 1000$).

- M dòng tiếp theo: Dòng thứ i ghi N kí tự trong tập từ 'a' đến 'z' với kí tự thứ j là điểm ảnh ở trong dòng i, cột j của bức ảnh thứ nhất.

- M dòng tiếp theo: Dòng thứ i ghi N kí tự trong tập từ 'a' đến 'z' với kí tự thứ j là điểm ảnh ở trong dòng i, cột j của bức ảnh thứ hai.

Dữ liệu ra: Ghi ra màn hình số cặp điểm khác nhau giữa hai bức ảnh.

Ví dụ:

Dữ liệu vào	Dữ liệu ra
2 4 aabc bbcd aabc bddd	2

```
#include<iostream>
using namespace std;
int main()
{
    int m,n;
    cin>>m>>n;

    string s;
    for(int i=0;i<m;i++)
        cin>s;
}
```

ví dụ 7:

GIÁ TRỊ NHỎ NHẤT CỦA BIỂU THỨC

Bài làm tốt nhất

Cho mảng A[], B[] đều có N phần tử. Nhiệm vụ của bạn là tìm giá trị nhỏ nhất của biểu thức P = A[0]*B[0] + A[1]*B[1] + ..+A[N-1]*B[N-1] bằng cách tráo đổi vị trí các phần tử của cả mảng A[] và B[].

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm 3 dòng: dòng thứ nhất đưa vào số phần tử của mảng N; dòng tiếp theo đưa vào N số A[i]; dòng cuối cùng đưa vào N số B[i] các số được viết cách nhau một vài khoảng trống.
- T, N, A[i], B[i] thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^7$; $0 \le A[i], B[i] \le 10^{18}$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
2 7 1 6 3 4 5 2 7 1 1 1 2 3 4 3 7 1 6 3 5 5 2 2 0 1 9 0 1 2 3	45 27

- Với trường hợp có bộ test thì ta sẽ cần thêm 1 biến t để đọc bộ test rồi thực hiện như bình thường

```
#include<iostream>
using namespace std;
int main()
{
    int t;
    cin>>t;
    while(t--)
    {
        int n;
        cin>>n;
        long long a[n+1],b[n+1];
        for(int i=0;i<n;i++)
            cin>>a[i];
        for(int i=0;i<n;i++)
            cin>>b[i];
    }
}
```

Ví dụ 8:

```
input:
anh tim noi nho
anh tim qua khu
nho lam ki uc anh va em
tra lai anh yeu thuong ay
xin nguoi hay tro lai noi day
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int t;
    cin>>t;
    cin.ignore();
    while(t--)
    {
        string s;
        getline(cin,s);
    }
```

CÁC THỦ THUẬT TRONG LẬP TRÌNH THI ĐẤU

FAST IN/OUT

Trong lập trình thi đấu thì việc đọc đầu vào khá quan trọng vì phải tiết kiệm thời gian càng nhanh càng tốt bạn có thể dùng scanf/printf của ngôn ngữ lập trình C để có tốc độ nhanh hơn. Tuy nhiên ta cũng có thể sử dụng cin/cout một cách bình thường với cách sử dụng như sau

```
ios_base::sync_with_stdio(false);
```

câu lệnh này giúp bật chức năng đồng bộ hóa quá trình đọc ghi của C++ giống như ngôn ngữ lập trình C

```
cin.tie(0);
//đồng bộ hóa hàm nhập
cout.tie(0);
//đồng bộ hóa hàm in
```

Thư viện STL

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

thư viện này là thư viện trung gian có tất cả các thư viện cần thiết bên trong nên trong lập trình thi đấu chúng ta chỉ cần khai báo thư viện này

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

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    return 0;
}
```

- Lưu ý: khi sử dụng '\n' sẽ nhanh hơn khi sử dụng <<endl; nên mình khuyến nghị các bạn sử dụng '\n' thay vì sử dụng endl;

Ví dụ khi vào ra bình thường

```
using namespace std;
int main()
{
    int n, k, t;
    int cnt = 0;
    cin >> n >> k;
    for (int i=0; i<n; i++)
    {
        cin >> t;
        if (t % k == 0)
            cnt++;
    }
    cout << cnt << "\n";
    return 0;
}
```

vào ra khi sử dụng Fast in/out

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

int main()
{
    // added the two lines below
    ios_base::sync_with_stdio(false);
```

```
    cin.tie(NULL);

    int n, k, t;
    int cnt = 0;
    cin >> n >> k;
    for (int i=0; i<n; i++)
    {
        cin >> t;
        if (t % k == 0)
            cnt++;
    }
    cout << cnt << "\n";
    return 0;
}
```

Một vài thủ thuật

- Kiểm tra số chẵn lẻ mà không sử dụng % mà ta sử dụng phép "&" gọi la phép end trong bảng logic

```
if (num & 1)
   cout << "ODD";
else
   cout << "EVEN";
```

ví dụ:

num = 5 -> hệ nhị phân 101&1 = 001 -> true

num = 4 -> hệ nhị phân 100&1 = 000 -> false

- Nhân chia nhanh cho 2

```
n = n << 1;   // nhân với 2
n = n >> 1;   // chia cho 2
```

Với cách sử dụng này sẽ nhanh hơn phép nhân chia cho 2 bình thường vì giúp máy bỏ qua quá trình trung gian khi tính đổi sang hệ cơ số 10

- Đổi chỗ 2 phần tử xử dụng XOR

```
a ^= b;
b ^= a;
a ^= b;
```

phương thức này sẽ nhanh hơn là khi sử dụng 3 biến trung gian để đổi chỗ 2 phần tử cho nhau

- Sử dụng **emplace_back()** thay cho **push_back()** thay vì phân bổ bộ nhớ ở một nơi khác và sau đó thêm trực tiếp cấp phát bộ nhớ trong vùng chứa
- Tính ước chung lớn nhất một cách nhanh chóng với thuật toán nền là xây dựng dựa trên thuật toán euclid

```
int a = 5;
int b = 4;
cout<<__gcd(4,5);

output: 1
```

Tương tự việc tính bội chung lớn nhất sẽ được thực hiện một cách dễ dàng

- Tối đa độ dài của 1 mảng được khai báo trong hàm main là 10^6 phần tử, nhưng khi khai báo mảng toàn cục thì có thể lên đến 10^7
- cách sử dụng hàm quicksort

```
sort(a,a+n);// với mảng bắt đầu từ 0 với a là mảng, n là số lượng phần tử của mảng
sort(a+1,a+n+1);//với mảng bắt đầu từ 1 với a là mảng, n là số lượng phần tử của mảng

sort(v.begin(), v.end()); // với vector
```

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

    int a[] = {1, 3, 2, 10, 4, 5};
    sort(a,a+6);
    for(int i=0;i<6;i++)
        cout<<a[i]<<" ";
}
```

- Tìm số chữ số trong 1 số bất kì

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n = 132544;
```

```
    int k = floor(log10(n)) + 1;
    cout<<k;
}

output: 6
```

- Khởi tạo 1 mảng chứa các giá trị 0 hoặc -1

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a[10][10];
    memset(a,0,sizeof(a));
    for(int i=0;i<10;i++)
    {
        for(int j=0;j<10;j++)
            cout<<a[i][j]<<" ";
        cout<<endl;

    }
}

output:
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

STL for newbies

- I. ITERATOR (BIẾN LẶP): MỤC LỤC
- II. CONTAINERS (THƯ VIỆN LƯU TRỮ)
    - a. Iterator:
    - b. Vector (Mảng động):
    - c. Deque (Hàng đợi hai đầu):
    - d. List (Danh sách liên kết):

    - e. Stack (Ngăn xếp):
    - f. Queue (Hàng đợi):
    - g. Priority Queue (Hàng đợi ưu tiên):
    - h. Set (Tập hợp):
    - i. Mutiset (Tập hợp):
    - j. Map (Ánh xạ):
    - k. Multi Map (Ánh xạ):

- III. STL ALGORITHMS (THƯ VIỆN THUẬT TOÁN):

    - a. Min, max
        - 1.1. min.....
        - 1.2. max
        - 1.3. next_permutation.....
        - 1.4. prev_permution
    - b. Sắp xếp:
    - c. Tìm kiếm nhị phân (các hàm đối với đoạn đã sắp xếp):
        - 3.1. binary_search:
        - 3.2. lower_bound:
        - 3.3. upper_bound

- IV. THƯ VIỆN STRING C++:

Lời nói đầu

- Tác giả Điêu Xuân Mạnh D11 PTIT (HV Công Nghệ Bưu Chính Viễn Thông 'nơi mình cũng đang theo học')

- "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

http://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)
.

- 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 #include //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): MỤC LỤC

- 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 :: 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 :: 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: o Truy cập đến phần tử riêng lẻ thông qua vị trí của nó O(1) o 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
1 2
1 3
1 5
2 3
2 6
3 5
6 1



2 3 5 6
1 3 6
1 2 5



1 3
1 2
```

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: o Các phần tử có thể truy cập thông cua chỉ số vị trí của nó. O(1) o Chèn hoặc xóa phần tử ở cuối hoặc đầu của dãy. O(1)

Khai báo: #include

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: o 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

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

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 phẩn 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

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

/Dạng 1 (sử dụng phép toán mặc định là less)/ priority_queue pq;

/* Dạng 2 (sử dụng phép toán khác) */ priority_queue <int,vector,greater > 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,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 s; set <int, greater > 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 : o xóa theo iterator ĐPT O(logN) o 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 <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_dữliệu (*it).second; // Lấy giá trị của giá trị của khóa, kiểu_dữliệu (*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 : o xóa theo iterator ĐPT O(logN) o 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):

- Khai báo sử dụng: #include
- Các hàm trong STL Algorithm khá nhiều nên mình chỉ giới thiệu sơ qua về một số hàm hay sử dụng trong các bài toán.
- Có một lưu ý nhỏ cho các bạn là khi sử dụng các hàm mà thực hiện trong một đoạn phần tử liên tiếp nào đó thì các hàm trong c++ thường có tác dụng trên nửa đoạn [..). Ví dụ như: bạn muốn hàm f có tác dụng trong đoạn từ 1 n thì các bạn phải gọi hàm trong đoạn từ 1 n+1.

1. Min, max

1.1. min: trả về giá trị bé hơn theo phép so sánh (mặc định là phép toán less):

Ví dụ: min(‘a’,’b’) sẽ return ’a’;min(3,1) sẽ return 1;

1.2. max thì ngược lại với hàm min:

Ví dụ: max(‘a’,’b’) sẽ return ‘b’max(3,1) sẽ return 1.

1.3. next_permutation: hoán vị tiếp theo. Hàm này sẽ return 1 nếu có hoán vị

tiếp theo, 0 nếu không có hoán vị tiếp theo.Ví dụ:// next_permutation#include <iostream>#include

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

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

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
4 2 1
```

1.4. prev_permution: ngược lại với next_permutation

2. Sắp xếp:

- sort: sắp xếp đoạn phần tử theo một trình tự nào đó. Mặc định của sort là sử dụng

operator <.

- Bạn có thể sử dụng hàm so sánh, hay class so sánh tự định nghĩa để sắp xếp cho linh hoạt.

- Chương trình mẫu:

```
// 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};


// using default comparison (operator <):
sort (myints, myints+4); //(12 32 45 71)26 80 53 33


// using function as comp
sort (myints+4, myints+8, myfunction); // 12 32 45 71(26 33 53 80)


for (int i=0;i<8;i++) cout << myints[i] << " ";
cout << endl;


// using object as comp
sort (myints, myints+8, myobject); //(80 71 53 45 33 32 26 12)


for (int i=0;i<8;i++) cout << myints[i] << " ";
cout << endl;
```

```
system("pause");


return 0;
}
```

3. Tìm kiếm nhị phân (các hàm đối với đoạn đã sắp xếp):

3.1. binary_search:

- tìm kiếm xem khóa có trong đoạn cần tìm không. Lưu ý: đoạn tìm kiếm phải được sắp

xếp theo một trật tự nhất đinh. Nếu tìm được sẽ return true, ngược lại return false.

- Dạng 1: binary_search(vị trí bắt đầu, vị trí kết thúc, khóa);
- Dạng 2: binary_search(vị trí bắt đầu, vị trí kết thúc, khóa, phép so sánh);

```
// 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 hàm so sánh tự định nghĩa:
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;
}
```

3.2. lower_bound:

- Hàm lower_bound và upper_bound tương tự như ở trong container set hay map.
- Trả về iterator đến phần tử đầu tiên trong nửa đoạn [first,last] mà không bé hơn khóa

tìm kiếm.

- Dạng 1: lower_bound(đầu , cuối , khóa);
- Dạng 2: lower_bound(đầu , cuối , khóa , phép toán so sánh của đoạn)

3.3. upper_bound

- Trả về iterator đến phần tử đầu tiên trong nửa đoạn [first,last] mà lớn hơn khóa tìm

kiếm.

- Dạng 1: upper_bound (đầu , cuối , khóa);
- Dạng 2: upper_bound (đầu , cuối , khóa, phép toán so sánh của đoạn)
- Chương trình demo:

```
// 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
```

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
- 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: o Sử dụng toán tử >> : tương tự như trong C. Nhập đến khi gặp dấu cách. o Sử dụng getline: giống như gets trong C. Nhập cả một dòng kí tự (chứa cả dấu cách nếu có) cho string. o 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, hay toán tử "==","!=" để so sánh. Chức năng này khá giống với xâu ở trong ngôn ngữ pascal.
- Capacity: o size: trả về độ dài của string o length: trả về độ dài của string o clear : xóa string o empty: return true nếu string rỗng, false nếu ngược lại
- Truy cập đến phần tử: o operator [chỉ_số]: lấy kí tự vị trí chỉ_số của string
- Chỉnh sửa: o push_back: chèn kí tự vào sau string o insert (n,x): chèn x vào string ở trước vị trí thứ n. x có thể là string, char,... o erase (pos,n): xóa khỏi string "n" kí tự bắt đầu từ vị trí thứ "pos". o erase

(iterator): xóa khỏi string phần tử ở vị trí iterator. o replace (pos, size, s1) : thay thế string từ vị trí “pos”, số phần tử thay thế là “size” và thay bằng xâu s1.

o swap (string_cần_đổi): đổi giá trị 2 xâu cho nhau.

- String operations:

```
o c_str : chuyển xâu từ dạng string trong C++ sang string trong C.
o substr (pos,length): return string được trích ra từ vị trí thứ “pos”, và trích ra
“length” kí tự.
```