Chương 4. SẮP XẾP VÀ TÌM KIẾM

Sắp xếp và tìm kiếm là các bài toán kinh điển trong tin học. Đây là những bài toán không thể thiếu trong các hệ thống thông tin quản lý. Việc sắp xếp giúp bố trí lại tập dữ liệu, góp phần giúp quá trình khai thác và tìm kiếm dữ liệu được thực hiện hiệu quả hơn. Chẳng hạn các email trong hộp thư được sắp theo thứ tự thời gian nhận giúp việc quản lý và tìm kiếm được thực hiện dễ dàng, tiện lợi. Các tập tin và thư mục trong máy tính luôn được sắp xếp theo một thuộc tính nào đó: tên, kích thước, ngày tạo, ngày chỉnh sửa, … Các từ trong một từ điển luôn được sắp theo thứ tự alphabet giúp tra cứu tiện lợi và nhanh chóng.

4.1. Bài toán sắp xếp

4.1.1. Phát biểu bài toán

Cho dãy số $a_1, a_2, \dots, a_n$. Hãy sắp xếp để tạo thành dãy có thứ tự không giảm, nghĩa là các phần tử của dãy thỏa điều kiện $a_1 \le a_2 \le \cdots \le a_n$.

Có rất nhiều phương pháp sắp xếp khác nhau. Mỗi phương pháp được áp dụng trong một số trường hợp cụ thể. Ta xem xét 2 phương pháp sắp xếp phổ biến sau.

4.1.2. Phương pháp đổi chỗ trực tiếp (interchange sort)

Ý tưởng: dãy có thứ tự $a_1 \le a_2 \le \cdots \le a_n$ suy ra $\forall i < j, a_i \le a_j$.

Thuật toán: Xét tất cả cặp phần tử $a_i, a_j (i < j)$. Nếu $a_i \ge a_j$ thì đổi chỗ cặp phần tử này.

Đoạn chương trình thực hiện

```
void InterchangeSort(int n)
{
    for (int i = 1; i < n; ++i)
        for (int j = i+1; j <= n; ++j)
            if (a[i] > a[j]) swap(a[i], a[j]);
}
```

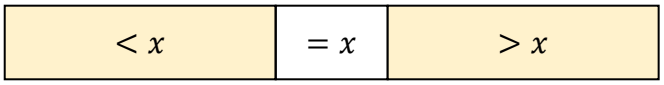
Đánh giá thuật toán:

- Số lần kiểm tra điều kiện của câu lệnh `if (a[i] >= a[j])` là $n(n-1)/2$. Số lần thực hiện phép đổi chỗ phụ thuộc vào điều kiện.
- Trường hợp tốt nhất: điều kiện `a[i] >= a[j]` luôn luôn đúng, số phép đổi chỗ là 0.
- Trường hợp xấu nhất: điều kiện `a[i] >= a[j]` luôn luôn sai, số phép đổi chỗ là $n(n-1)/2$.
- Độ phức tạp của thuật toán $O(n^2)$.

4.1.3. Phương pháp sắp xếp nhanh (quick sort)

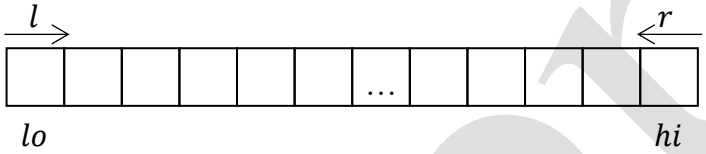
Ý tưởng: phân hoạch dãy cần sắp thành 2 đoạn con

- Đoạn trái: gồm các phần tử có giá trị nhỏ hơn chốt.
- Đoạn phải: gồm các phần tử có giá trị lớn hơn chốt.
- Chốt là giá trị của một phần tử trong dãy được sử dụng làm tiêu chí phân hoạch dãy.



Hình 4.1.3.1. minh họa x là chốt

Để phân hoạch dãy, ta sử dụng 2 con trỏ l, r xuất phát từ 2 đầu của dãy (dãy gồm các phần tử nằm từ vị trí lo đến vị trí hi), mỗi con trỏ sẽ phát hiện phần tử nằm sai đoạn để đổi chỗ cặp phần tử này.



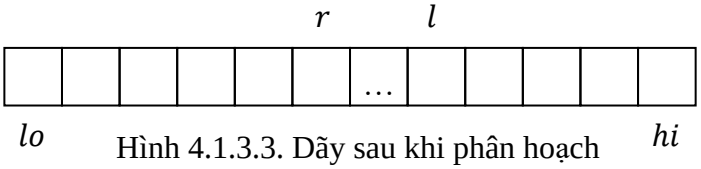
Hình 4.1.3.2. Sử dụng con trỏ l xuất phát từ lo và con trỏ r xuất phát từ hi

Đoạn chương trình phân hoạch dãy thành 2 đoạn con theo chốt x

```
while (l <= r)
{
    while (a[l] < x) ++l;
    while (a[r] > x) --r;
    if (l <= r)
    {
        swap(a[l], a[r]);
        ++l, --r;
    }
}
```

Sau đoạn lệnh trên, dãy được phân hoạch thành 2 đoạn con (như hình minh họa 4.1.3.3):

- Đoạn trái: gồm các phần tử từ vị trí lo đến vị trí r và có giá trị $< x$
- Đoạn phải: gồm các phần tử từ vị trí l đến vị trí hi và có giá trị $> x$



Hình 4.1.3.3. Dãy sau khi phân hoạch

Các phần tử trong mỗi đoạn con này chưa đảm bảo có thứ tự, vì vậy ta tiếp tục phân hoạch từng đoạn con này cho đến khi không thể phân hoạch được nữa (đoạn chỉ có 1 phần tử).

Để ý rằng, giá trị chốt ảnh hưởng đến tính cân bằng về số lượng phần tử của mỗi đoạn con sau thao tác phân hoạch. Nếu chốt là giá trị trung vị của dãy sẽ phân hoạch dãy thành 2 đoạn con số phần tử chênh lệch nhau ít nhất (giá trị trung vị của dãy là giá trị của phần tử ở vị trí giữa dãy có thứ tự), trong trường hợp tối ưu thì dãy được phân hoạch thành 2 đoạn con có số phần tử bằng nhau. Nếu chốt có giá trị lớn nhất hoặc nhỏ nhất dãy sẽ phân hoạch dãy thành 2 đoạn con có số phần tử chênh lệch nhau nhiều nhất.

Mặt khác, ta nhận thấy rằng dãy sẽ nhanh phân hoạch thành các đoạn con có độ dài 1 hơn nếu thao tác phân hoạch chia dãy thành 2 đoạn con cân bằng nhau về số phần tử. Xét dãy có n phần tử:

- Trường hợp chốt được chọn luôn là giá trị lớn nhất hoặc nhỏ nhất: sau mỗi lần phân hoạch, dãy được chia thành 2 dãy con có độ dài 1 và $n-1$. Như vậy sau n lần phân hoạch, dãy được chia thành các dãy con có độ dài 1.
- Trường hợp chốt được chọn luôn là giá trị trung vị: sau mỗi lần phân hoạch, dãy được phân hoạch thành 2 dãy con có độ dài cỡ $n/2$. Do đó ở lần phân hoạch thứ k, dãy được chia thành các đoạn con có cùng số phần tử cỡ $n/2^k$. Thao tác phân hoạch dừng khi $n = 2^k \Longrightarrow k = \log_2 n$.

Như vậy nếu chốt là giá trị trung vị thì thuật toán sẽ đạt hiệu quả tối ưu. Tuy nhiên, việc chọn chốt là giá trị trung vị lại khá tốn kém. Do đó người ta thường chọn chốt là một giá trị nằm ngẫu nhiên từ vị trí lo đến vị trí hi, khi đó thuật toán sẽ đạt hiệu quả gần tối ưu.

Cài đặt hàm sắp xếp nhanh

```
void quicksort(int lo, int hi)
{
    int l = lo, r = hi, x = a[l + rand() % (r-l+1)];
    while (l <= r)
    {
        while (a[l] < x) ++l;
        while (a[r] > x) --r;
        if (l <= r)
            swap(a[l++], a[r--]);
    }
    if (lo < r) quicksort(lo, r);
    if (l < hi) quicksort(l, hi);
}
```

Đánh giá thuật toán:

- Độ phức tạp của thao tác phân hoạch dãy n phần tử thành 2 dãy con là $O(n)$.
- Trường hợp xấu nhất (chốt luôn có giá trị lớn nhất hoặc nhỏ nhất): thuật toán thực hiện n bước phân hoạch. Độ phức tạp của thuật toán $O(n^2)$.
- Trường hợp tối ưu (chốt luôn là giá trị trung vị): thuật toán thực hiện $\log_2 n$ bước phân hoạch. Độ phức tạp của thuật toán $O(n \times \log n)$.
- Trường hợp trung bình (chốt được chọn là phần tử ngẫu nhiên trong dãy cần phân hoạch): độ phức tạp trung bình cỡ $O(n \times \log n)$.

4.1.4. Sắp xếp trên nhiều khóa

Bài toán sắp xếp như trình bày ở trên chỉ xét trên dãy số nguyên, nghĩa là mỗi phần tử chỉ có 1 thuộc tính. Sắp xếp trên nhiều khóa là bài toán sắp xếp một danh sách mà mỗi phần tử của danh sách gồm có nhiều thuộc tính.

Trong thực tế, sắp xếp trên nhiều khóa là bài toán rất thông dụng. Chẳng hạn như sắp xếp một danh sách học sinh, mỗi học sinh gồm có 2 thuộc tính là `họ lót` và `tên`, danh sách được sắp tăng dần theo `tên`, nếu cùng `tên` thì sắp tăng dần theo `họ lót`. Ta xét bài toán sau:

Cho danh sách gồm n phần tử $a_1, a_2, \dots, a_n$, trong đó phần tử a_i có 2 thuộc tính nguyên (x, y). Hãy sắp xếp danh sách có thứ tự tăng dần theo thuộc tính x, nếu có cùng thuộc tính x thì sắp tăng dần theo thuộc tính y.

Ví dụ: xét danh sách gồm 10 phần tử sau

i	1	2	3	4	5	6	7	8	9	10
x	3	4	8	6	3	4	12	8	3	5
y	9	7	3	4	2	2	1	6	7	10

Danh sách sau khi được sắp xếp có thứ tự tăng dần theo x, cùng x thì sắp tăng dần theo y:

i	1	2	3	4	5	6	7	8	9	10
x	3	3	3	4	4	5	6	8	8	12
y	2	7	9	2	7	10	4	3	6	1

Ta thấy rằng việc sắp thứ tự danh sách phụ thuộc vào thao tác so sánh 2 phần tử để xác định thứ tự của chúng (phần tử nhỏ hơn đứng trước). Với cặp phần tử (a_i, a_j), $(a_i < a_j)$ nếu thỏa điều kiện sau:

$$(a_i.x < a_j.x) \text{ hoặc } (a_i.x = a_j.x \text{ và } a_i.y < a_j.y)$$

Do đó thao tác sắp xếp danh sách 2 khóa được thực hiện tương tự như trên danh sách 1 khóa, chỉ khác nhau ở thao tác so sánh 2 phần tử.

Để khai báo kiểu dữ liệu cho phần tử có 2 thuộc tính thì ta có thể sử dụng kiểu dữ liệu tự định nghĩa `struct` như sau:

```
struct element
{
    int x, y;
} a[maxN];

//với 2 phần tử a, b của danh sách, điều kiện để a đứng trước b
bool compare(element a, element b)
{
    return (a.x < b.x || (a.x == b.x && a.y < b.y));
}
```

hoặc kiểu `pair` như sau:

```
typedef pair<int, int> pii;
pii a[maxN];

//với 2 phần tử a, b của danh sách, điều kiện để a đứng trước b
bool compare(pii a, pii b)
{
    return (a.first < b.first ||
            (a.first == b.first && a.second < b.second));
}
```

Đoạn chương trình minh họa sắp xếp danh sách 2 khóa bằng phương pháp đổi chỗ trực tiếp:

```
void InterchangeSort(int n)
{
    for (int i = 1; i < n; ++i)
        for (int j = i+1; j <= n; ++j)
            if (compare(a[j], a[i])) swap(a[i], a[j]);
}
```

Đoạn chương trình minh họa sắp xếp danh sách 2 khóa bằng phương pháp quick sort:

```
void quicksort(int lo, int hi)
{
    int l = lo, r = hi;
    pii x = a[l + rand() % (r-l+1)];
    while (l <= r)
    {
        while (compare(a[l], x)) ++l;
        while (compare(x, a[r])) --r;
        if (l <= r)
            swap(a[l++], a[r--]);
    }
    if (lo < r) quicksort(lo, r);
    if (l < hi) quicksort(l, hi);
}
```

4.2. Bài toán tìm kiếm

4.2.1. Phát biểu bài toán

Cho dãy số nguyên $a_1, a_2, \dots, a_n$ và số nguyên x. Hãy cho biết có tồn tại phần tử có giá trị x trong dãy. Nếu có hãy chỉ ra 1 vị trí bất kỳ. Ngược lại xuất kết quả là -1.

Có hai phương pháp tìm kiếm thông dụng trên dãy: tìm kiếm tuần tự và tìm kiếm nhị phân.

4.2.2. Tìm kiếm tuần tự

Phương pháp tìm kiếm tuần tự được áp dụng để tìm kiếm phần tử trong dãy không có thứ tự.

Ý tưởng: duyệt lần lượt từng phần tử của dãy cho đến khi tìm thấy giá trị x hoặc duyệt hết dãy và không tìm thấy.

Cài đặt hàm tìm tuần tự

```
int SeqSearch(int n, int x)
{
    for (int i = 1; i <= n; ++i)
        if (a[i] == x)
            return i;
    return -1;
}
```

Đánh giá thuật toán: $O(n)$

4.2.3. Tìm kiếm nhị phân

Điều kiện áp dụng: không gian tìm kiếm cần phải có thứ tự (tăng dần hoặc giảm dần) theo tiêu chí tìm kiếm.

Một số ví dụ ứng dụng tìm kiếm nhị phân

- Tìm số báo danh trong một danh sách có thứ tự tăng dần theo số báo danh.
- Tìm họ và tên của một thí sinh trong danh sách có thứ tự tăng dần theo tên và họ.
- Tra nghĩa của một từ trong một từ điển (các từ trong từ điển được sắp thứ tự alphabet – tăng dần theo chữ cái).

Ý tưởng: dựa vào tính có thứ tự của không gian tìm kiếm để loại bỏ đi một phần không gian không tồn tại phần tử cần tìm kiếm. Phần không gian tìm kiếm bị loại bỏ bằng một nửa so với không gian tìm kiếm ban đầu.

Thuật toán: tìm vị trí của phần tử có giá trị x trong dãy $a_1 \le a_2 \le \cdots \le a_n$. Tổng quát, ta cần tìm vị trí của x trong dãy $a_{lo} \le a_{lo+1} \le \cdots \le a_{hi}$

- Bước 1: Nếu dãy không có phần tử ($lo > hi$) thì dừng thất bại.
- Bước 2: Đặt $m = (lo + hi)/2$.
- Bước 3: Kiểm tra $x = a_m$.

  Nếu đúng thì dừng thành công.

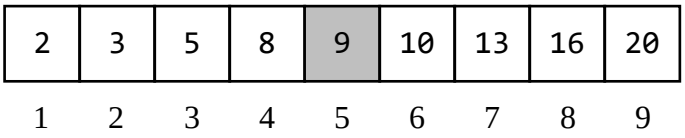
  Ngược lại đến Bước 4.
- Bước 4: Kiểm tra $x < a_m$.

  Nếu đúng thì đoạn $[m, hi]$ không chứa x. Vùng tìm kiếm là $[lo, m-1]$. Quay về Bước 1.

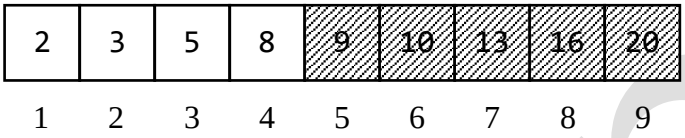
Ngược lại đoạn $[lo, m]$ không chứa x. Vùng tìm kiếm là $[m+1, hi]$. Quay về Bước 1.

Ví dụ: tìm phần tử $x = 5$ trong dãy $a = [2, 3, 5, 8, 9, 10, 13, 16, 20]$.

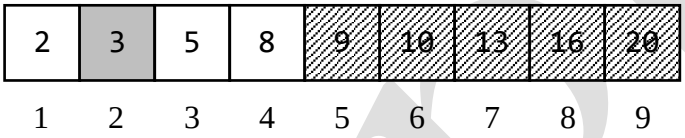
- Lần thứ 1: so sánh $x = 5$ với phần tử giữa đoạn $[1..9]$ (vị trí 5).



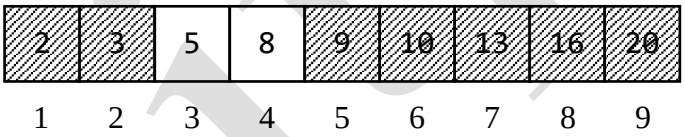
  + Do $x < a[5]$ nên x không thể xuất hiện trong đoạn $[5..9]$. Vùng tìm kiếm là $[1..4]$.



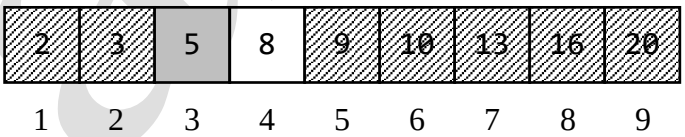
- Lần thứ 2: so sánh $x = 5$ với phần tử giữa đoạn $[1..4]$ (vị trí 2).



  + Do $x > a[2]$ nên x không thể xuất hiện trong đoạn $[1..2]$. Vùng tìm kiếm là $[3..4]$.



- Lần thứ 3: so sách $x = 5$ với phần tử giữa đoạn $[3..4]$ (vị trí 3).



  + Vì $x = a[3]$ nên thuật toán dừng thành công.

Cài đặt hàm tìm kiếm nhị phân

```
int BinSearch(int n, int x) {
    int lo = 1, hi = n;
    while (lo <= hi) {
        int m = (lo + hi)/2;
        if (x == a[m]) return m;
        if (x < a[m]) hi = m-1;
        else lo = m+1;
    }
    return -1;
}
```

- *Thuật toán có độ phức tạp:* $O(\log n)$.

4.2.4. Kỹ thuật chặt nhị phân

Ý tưởng của phương pháp tìm kiếm nhị phân có thể áp dụng để giải bài toán tìm giá trị lớn nhất hoặc nhỏ nhất trong đoạn giá trị $[lo..hi]$ thỏa điều kiện nào đó, còn gọi là phương pháp chặt nhị phân. Một cách hình thức, bài toán có thể được mô hình hóa như sau: tìm giá trị x lớn nhất hoặc nhỏ nhất trong miền giá trị $[lo..hi]$ thỏa hàm số $f(x)$.

Điều kiện để có thể áp dụng phương pháp chặt nhị phân cho bài toán này đó là $\forall x < y: f(x) \leq f(y)$ hoặc $\forall x < y: f(x) \geq f(y)$, nói cách khác hàm $f(x)$ là hàm đồng biến hoặc nghịch biến trên miền giá trị của x.

Ứng dụng: Cho số nguyên dương $M(M \leq 10^{18})$. Tìm số nguyên dương n nhỏ nhất thỏa điều kiện

$$1 + 2 + \cdots + n \geq M(*)$$

Lời giải:

Bản chất của bài toán là tìm kiếm giá trị n nhỏ nhất thỏa điều kiện bất phương trình $(*)$.

Cách 1: Tìm kiếm tuần tự

- Xét n bắt đầu từ 1 trở đi và kiểm tra tổng nếu thỏa $\geq M$ thì dừng thuật toán.

Đoạn chương trình minh họa

```
S = n = 0;
while (S < M)
{
    ++n;
    S = S + n;
}
cout<<n;
```

Phân tích độ phức tạp của thuật toán:

$$S = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \geq M \Longrightarrow n^2 + n \geq 2 \times M$$

Số lần lặp của thuật toán cỡ

$$\left\lceil \frac{\left(-1 + \sqrt{1 + 8 * M}\right)}{2} \right\rceil$$

Vậy thuật toán có độ phức tạp $O\left(\sqrt{8 * M}\right)$.

Cách 2:

- Gọi $f(n) = 1 + 2 + \cdots + n$. Ta có $\forall n_1 < n_2: f(n_1) < f(n_2)$. Do đó ta có thể áp dụng kỹ thuật chặt nhị phân giá trị n trong đoạn $\left[1..\sqrt{8 * M}\right]$.

Đoạn chương trình minh họa

```
typedef long long ll;
ll Sum(ll n)
{
    return (n*(n+1)/2);
}

void Solve(ll M)
{
    ll lo = 1, hi = sqrt(8*M), n;
    while (lo <= hi)
    {
        ll x = (lo + hi)/2;
        if (Sum(x) >= M)
        {
            hi = x-1;
            n = x; //x là nghiệm ứng viên
        }
        else lo = x+1;
    }
    cout<<n;
}
```

Thuật toán chặt nhị phân giá trị n trong đoạn từ 1 đến $\sqrt{8 * M}$ nên có độ phức tạp $O(\log \sqrt{8 * M})$.

4.3. Sử dụng hàm của thư viện

4.3.1. Hàm sắp xếp của thư viện

Thư viện `<algorithm>` cung cấp hàm `sort` hỗ trợ sắp xếp dãy bất kỳ nếu xác định được phép so sánh giữa 2 phần tử trong dãy.

Sắp xếp tăng dần:

- Xét dãy số nguyên $a_0, a_1, \dots, a_{n-1}$, để sắp xếp dãy thì gọi lệnh: `sort(a, a+n)`
- Xét dãy số nguyên $a_1, a_2, \dots, a_n$, để sắp xếp dãy thì gọi lệnh: `sort(a+1, a+n+1)`
- Tổng quát, để sắp xếp các phần tử $a_l, a_{l+1}, \dots, a_r$ có thứ tự tăng dần, các phần tử còn lại giữ nguyên thứ tự thì gọi lệnh: `sort(a+l, a+r+1)`
- Với `a` là một `vector` thì câu lệnh sắp xếp các phần tử tăng dần: `sort(a.begin(), a.end())`

Hàm `sort` mặc định sắp xếp các phần tử có thứ tự tăng dần nên không cần phải chỉ định thao tác so sánh giữa các phần tử trong dãy. Để sắp xếp dãy có thứ tự giảm dần hoặc dãy mà mỗi phần tử có nhiều thuộc tính thì cần chỉ định thao tác so sánh giữa các phần tử để xác định thứ tự thông qua hàm so sánh có cấu trúc như sau:

```
//x và y là 2 phần tử của dãy cần sắp xếp
bool <tên hàm>(<type> x, <type> y)
{
```

```
    return <điều kiện để x đứng trước y>;
}
```

Ví dụ 4.1: Chương trình sắp xếp danh sách các số nguyên có thứ tự giảm dần

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

bool cmp(int x, int y)
{
    return x > y;
}

int main()
{
    int a[] = { 3, 8, 4, 1, 0, 8, 9, 2, 20, 5 };
    int n = 10;

    sort(a, a + n, cmp); //cách chỉ định thao tác so sánh trong sắp xếp

    cout << "mảng sau khi được sắp xếp : \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```

Ví dụ 4.2: Cho danh sách gồm *n* học sinh, mỗi học sinh gồm 2 thành phần: tên có dạng là một chuỗi kí tự và điểm trung bình là một số thực. Hãy sắp xếp danh sách học sinh có thứ tự giảm dần theo điểm trung bình. Những học sinh có cùng điểm trung bình thì sắp tăng dần theo tên.

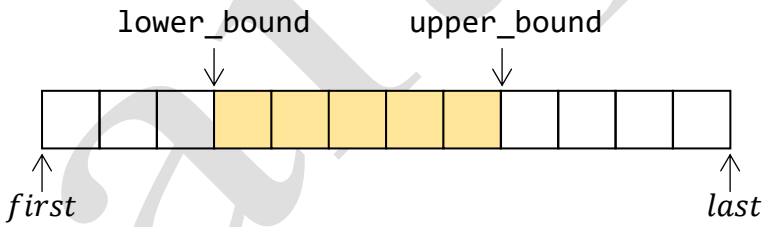
```
struct student
{
    string ten;
    double diem;
} a[maxN];

bool cmp(student x, student y) //điều kiện để x đứng trước y
{
    return (x.diem > y.diem || //giảm dần theo điểm
            (x.diem == y.diem && x.ten < y.ten));
}

int main()
{
    student a[] =  {{"minh", 9.5},
                    {"long", 9.2},
                    {"an", 8.9},
                    {"hung", 9.5},
```

```
                    {"linh", 8.9},
                    {"nghia", 9.1},
                    {"binh", 8.9}};
    int n = 7;

    sort(a, a + n, cmp);

    cout << "danh sách sau khi được sắp xếp : \n";
    for (int i = 0; i < n; ++i)
        cout<<a[i].ten<<" "<<a[i].diem<<"\n";

    return 0;
}
```

4.3.2. Hàm tìm kiếm nhị phân của thư viện

Thư viện `<algorithm>` còn cung cấp 2 hàm `lower_bound` và `upper_bound` hỗ trợ tìm kiếm phần tử theo tư tưởng tìm kiếm nhị phân. Điều kiện để 2 hàm trên thực thi đúng đắn là dãy phải được sắp thứ tự tăng dần theo tiêu chí tìm kiếm.

- Hàm `lower_bound(first, last, val)` trả về con trỏ chỉ đến phần tử *nhỏ nhất* trong vùng từ `first` đến < `last` mà có giá trị ≥ `val`.
- Hàm `upper_bound(first, last, val)` trả về con trỏ chỉ đến phần tử *nhỏ nhất* trong vùng từ `first` đến < `last` mà có giá trị > `val`.



Ví dụ 4.3: Chương trình sau minh họa cách sử dụng 2 hàm

```
int main()
{
    int a[] =  {4, 5, 8, 10, 10, 10, 10, 13, 16, 20}, n = 10;
    //                  ^                ^
    //                  p1               p2
    int p1 = lower_bound(a, a+n, 10) - a; //p1 = 3
    int p2 = upper_bound(a, a+n, 10) - a; //p2 = 7
    int p3 = lower_bound(a, a+n, 50) - a; //p3 = 10
    int p4 = upper_bound(a, a+n, 50) - a; //p4 = 10
    int p5 = lower_bound(a, a+n, 1) - a; //p5 = 0
    int p6 = upper_bound(a, a+n, 1) - a; //p6 = 0
    return 0;
}
```