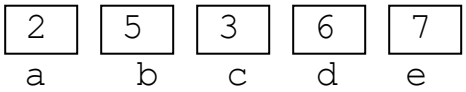
BÀI 4
KỸ THUẬT LẬP TRÌNH VỚI MẢNG

I. Mảng một chiều

I.1. Khái niệm và cách khai báo

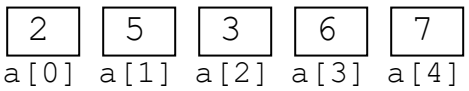
Bài toán: hãy lưu trữ một dãy số gồm 5 phần tử: {2, 5, 3, 6, 7}

Cách 1: Sử dụng 5 ô nhớ (5 biến) cùng kiểu. Các biến được đặt tên lần lượt là: a, b, c, d, e. Khi đó, các phần tử được chứa trong 5 ô nhớ này như sau:



Vì cần lưu trữ 5 giá trị khác nhau nên việc dùng 5 ô nhớ khác nhau là cần thiết. Tuy nhiên, phương pháp này tỏ không khả thi do sử dụng quá nhiều tên biến, dẫn tới khó kiểm soát các biến, đặc biệt trong trường hợp số phần tử của dãy quá lớn.

Cách 2: Vẫn sử dụng 5 ô nhớ cùng kiểu nhưng tất cả các ô được đặt chung một tên (a chẳng hạn). Để phân biệt các ô với nhau, người ta đánh chỉ số cho từng ô. Chỉ số là các số nguyên liên tiếp, tính từ 0. Như vậy ta thu được:



Kết quả ta có được một cấu trúc dữ liệu khắc phục được nhược điểm của cách 1. Cấu trúc dữ liệu này gọi là mảng một chiều.

Mảng (hay mảng một chiều) là một cấu trúc bộ nhớ bao gồm một dãy liên tiếp các ô nhớ cùng tên, cùng kiểu nhưng khác nhau về chỉ số, dùng để lưu trữ một dãy các phần tử cùng kiểu.

Cú pháp khai báo mảng:

<Kiểu mảng> <Tên mảng> [<Số phần tử tối đa>];

Trong đó:

<Kiểu mảng>: Là kiểu dữ liệu của mỗi phần tử trong mảng, có thể là một kiểu dữ liệu chuẩn hoặc kiểu dữ liệu tự định nghĩa.

<Tên mảng>: Được đặt tuỳ ý tuân theo quy ước đặt tên biến trong C++.

<Số phần tử tối đa>: Là một số nguyên (hoặc một biến đang chứa giá trị nguyên) chỉ ra số ô nhớ tối đa được dành cho mảng cũng như số phần tử tối đa mà mảng có thể chứa được.

Ví dụ: Khai báo int a[3]; sẽ cấp phát 3 ô nhớ liên tiếp cùng kiểu nguyên (2 byte/ 1 ô) dành cho mảng a. Mảng này có thể chứa được tối đa 3 số nguyên.

I.2. Các thao tác cơ bản trên mảng một chiều

- Nhập mảng: Giả sử ta cần nhập mảng a gồm n phần tử. Cách duy nhất là nhập từng phần tử cho mảng. Do vậy, ta cần sử dụng một vòng lặp for duyệt qua từng phần tử và nhập dữ liệu cho chúng. Nhưng trước tiên, cần nhập số phần tử thực tế của mảng (n).

```
cout<< "n="; cin>>n;
for(int i=0; i<n; i++)
{
   cout<< "a["<<i<< "]=";
   cin>>a[i];
}
```

- Xuất mảng: tương tự như nhập mảng, ta cũng cần sử dụng vòng lặp for để xuất từng phần tử của mảng lên màn hình (lựa chọn 1 trong 4 lệnh cout trong vòng lặp for):

```
for(int i = 0; i<n; i++)
   cout<<a[i];
   cout<<a[i]<< "  ";
   cout<<a[i]<<endl;
   cout<<setw(5)<<a[i];
```

- **Duyệt mảng**: là thao tác thăm lần lượt từng phần tử của mảng. Thao tác duyệt mảng có trong hầu hết các bài toán sử dụng mảng.

Duyệt xuôi: (từ trái qua)

```
for(int i = 0; i<n; i++)
{
        thăm phần tử a[i]
}
```

Duyệt ngược: (từ phải qua)

```
for(int i = n-1; i>=0; i--)
{
        thăm phần tử a[i]
}
```

I.3. Các bài toán cơ bản

a. Bài toán sắp xếp mảng

Bài toán: cho một dãy gồm n phần tử. Hãy sắp xếp dãy theo chiều tăng dần.

Để giải quyết bài toán này, trước tiên ta cần lưu trữ dãy các phần tử đã cho vào bộ nhớ, như vậy ta cần sử dụng một mảng một chiều. Sau đó, có rất nhiều phương pháp để sắp một mảng theo chiều tăng dần. Sau đây ta xem xét một số cách phổ biến:

- **Phương pháp 1:** Sắp xếp nổi bọt – Bubble sort

Ý tưởng của phương pháp như sau:

- Sắp lần lượt từng phần tử của dãy, bắt đầy từ phần tử đầu tiên.

- Giả sử cần sắp phần tử thứ i, ta tiến hành duyệt lần lượt qua tất cả các phần tử đứng sau nó trong dãy, nếu gặp phần tử nào nhỏ hơn phần tử đang sắp thì đổi chỗ.

Giả sử ta sắp mảng a gồm n phần tử, giải thuật được mô tả chi tiết như sau:

```
for(int i = 0; i < n; i++)  // với mỗi phần tử a[i]
for(int j = i+1; j<n; j++)
   if(a[j]  < a[i])    Đổi chỗ a[i] cho a[j]
```

Để đổi chỗ a[i] cho a[j], ta có thể sử dụng hàm **swap(a[i], a[j])**. Tuy nhiên, ta cũng có thể tự đổi chỗ. Muốn vậy, ta sử dụng một biến tg có cùng kiểu và gán một trong 2 giá trị (a[i] hoặc a[j]) vào đó. Sau đó gán giá trị còn lại sang giá ô nhớ vừa gán vào tg. Cuối cùng ta gán giá trị đang chứa trong tg vào ô nhớ thứ 2 này.

```
for(int i = 0; i < n; i++)
for(int j = i+1; j<n; j++)
if(a[j]  < a[i])
{
        tg = a[i];
        a[i] = a[j];
        a[j] = tg;
}
```

Sắp xếp bằng phương pháp này trung bình cần $n^2/ 2$ lần so sánh và $n^2/2$ lần hoán vị. Trong trường hợp tồi nhất ta cũng cần số lần so sánh và hoán vị như vậy.

Giả sử ta có mảng a = {3, 5, 2, 7, 4, 8}. Hình ảnh của a sau các lần lặp sắp xếp nổi bọt như sau:

Bắt đầu sắp i = 0	3	5	2	7	4	8
Hết 1 vòng lặp j; i=1;	2	5	3	7	4	8
Hết 1 vòng lặp j; i=2;	2	3	5	7	4	8
Hết 1 vòng lặp j; i=3;	2	3	4	7	5	8
Hết 1 vòng lặp j; i=4;	2	3	4	5	7	8

Sau đây là hàm sắp nổi bọt mảng a gồm n phần tử thực:

```
void SapNoiBot(float a[], int n)
{
        for (int i=0; i<n; i++)
        for (int j=i+1; j<n; j++)
        if(a[i] > a[j])
        {
                float tg = a[i];
                a[i] = a[j];
                a[j] = tg;
        }
}
```

- **Phương pháp sắp xếp chọn** – Selection sort

Trong phương pháp sắp sếp nổi bọt, để sắp một phần tử nhiều khi ta phải đổi chỗ nhiều lần phần tử đang sắp với các phần tử đứng sau nó. Một ý tưởng rất hay là làm sao

chỉ đổi chỗ 1 lần duy nhất khi sắp một phần tử trong dãy. Đây chính là ý tưởng của phương pháp sắp xếp chọn.

Để làm được điều này, khi sắp phần tử thứ i, người ta tiến hành tìm phần tử nhỏ nhất trong số các phần tử đứng sau nó kể cả phần tử đang sắp rồi tiến hành đổi chỗ phần tử nhỏ nhất tìm được với phần tử đang sắp.

Ví dụ: Với dãy a = {1, 6, 4, 2, 5, 7}, để sắp phần tử thứ 2 (6) người ta tiến hành tìm phần tử nhỏ nhất trong số các phần tử {6, 4, 2, 5, 7}. Khi đó Min(6, 4, 2, 5, 7) = 2 và phần tử 2 được đảo chỗ cho phần tử 6. Kết quả thu được:

1	2	4	6	5	7

Trước tiên ta xem xét bài toán tìm phần tử nhỏ nhất của một dãy các phần tử:

- Lấy một phần tử ngẫu nhiên trong dãy làm phần tử nhỏ nhất (Min) (thường lấy phần tử đầu tiên).

- Duyệt qua tất cả các phần tử của dãy, nếu gặp phần tử nào nhỏ hơn Min thì gán Min bằng phần tử đó.

Ví dụ: Tìm số nhỏ nhất trong mảng a gồm n phần tử:

```
Min = a[0];
for(int i = 0; i<n; i++)
        if (a[i] < Min) Min = a[i];
```

Khi kết thúc vòng lặp, ta thu được giá trị nhỏ nhất của dãy đang chứa trong biến Min.

Tuy nhiên, áp dụng giải thuật này vào phương pháp sắp xếp chọn ta cần phải lưu ý một số điểm. Chẳng hạn ta cần biết chính xác vị trí của phần tử Min nằm ở đâu để tiến hành đổi chỗ chứ không quan tâm tới giá trị Min là bao nhiêu. Tuy nhiên giải thuật tìm Min lại chỉ cho biết giá trị Min mà không cho biết vị trí. Nếu muốn biết, ta lại phải sử dụng 1 vòng lặp duyệt lại từ đầu để tìm vị trí Min. Do vậy, khi cài đặt phương pháp sắp xếp chọn, để tránh trường hợp sử dụng nhiều vòng lặp sẽ làm tăng số thao tác của giải thuật, ta chỉ chú ý tới việc tìm **vị trí** phần tử Min.

- Sắp xếp từng phần tử của dãy, bắt đầu từ phần tử đầu tiên.

- Giả sử sắp phần tử a[i], ta gán Min = i rồi duyệt qua các phần tử đứng sau nó (i+1 trở đi). Nếu phần tử a[j] nào nhỏ hơn phần tử a[Min] thì gán Min bằng vị trí của a[j] (tức Min=j). Cuối cùng ta đổi chỗ a[i] cho a[Min].

```
for(int i = 0; i < n; i++)
{
        Min = i;
        for(int j = i+1; j<n; j++)
                if(a[j]  < a[Min])
                        Min = j;
    tg = a[i];
       a[i] = a[Min];
       a[Min] = tg;
}
```

Sắp xếp bằng phương pháp chọn cần $n^2/2$ lần so sánh và n lần hoán vị.

Giả sử ta có mảng a = {3, 5, 2, 7, 4, 8}. Hình ảnh của a qua các lần lặp sắp xếp chọn như sau:

3	5	2	7	4	8	Min = 2
2	5	3	7	4	8	Min = 3
2	3	5	7	4	8	Min = 4
2	3	4	7	5	8	Min = 5
2	3	4	5	7	8	Min = 7
2	3	4	5	7	8	Min = 8

Sau đây là hàm sắp xếp chọn với tham số là mảng a gồm n phần tử nguyên:

```
void SapChon(int a[], int n)
{
        for (int i=0; i<n; i++)
        {
                int min = i;
                for (int j = i+1; j<n; j++)
                        if (a[j] < a[min]) min = j;

                swap(a[i], a[min]);
        }
}
```

- **Phương pháp sắp xếp chèn**

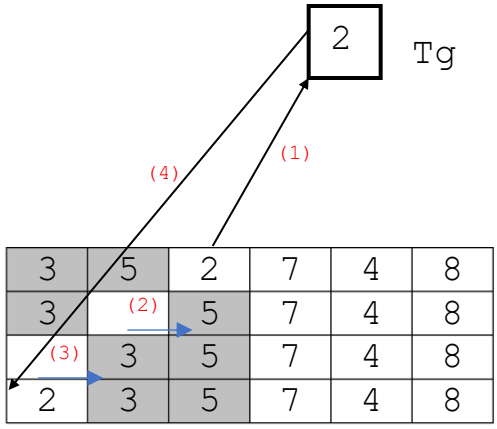
Một thuật toán gần như đơn giản ngang với thuật toán sắp xếp chọn nhưng có lẽ mềm dẻo hơn, đó là sắp xếp chèn. Đây là phương pháp người ta dùng để sắp xếp các thanh ngang cho một chiếc thang.

Đầu tiên, người ta rút ngẫu nhiên 1 thanh ngang và đặt vào 2 thanh dọc để làm thang. Tiếp đó, người ta lần lượt chèn từng thanh ngang vào sao cho không phá vỡ tính được sắp của các thanh đã được đặt trên 2 thanh dọc.

Giả sử với mảng a = {3, 5, 2, 7, 4, 8}. Giả sử các phần tử 3 và 5 đã được chèn vào đúng vị trí (đã được sắp):

3	5

Ta xem xét quá trình chèn phần tử tiếp theo vào mảng (giả sử chèn 2 vào). Khi đó, quá trình diễn ra như sau:



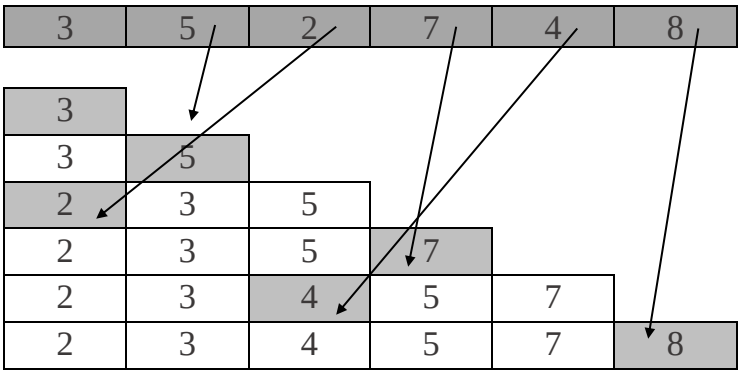
3	5	2	7	4	8
3		5	7	4	8
	3	5	7	4	8
2	3	5	7	4	8

- Đặt phần tử 2 vào biến tg.
- Duyệt qua các phần tử đứng trước phần tử 2 (các phần tử đã được sắp). Nếu gặp phần tử nào lớn hơn 2 thì đẩy phần tử đó sang phải 1 vị trí. Ngược lại, nếu gặp phần tử nhỏ hơn 2 thì chèn 2 vào ngay sau phần tử nhỏ hơn này. Nếu đã duyệt hết các phần tử đứng trước mà vẫn chưa tìm thấy phần tử nhỏ hơn 2 thì chèn 2 vào đầu mảng.

Kết thúc quá trình này, phần tử 2 đã được chèn đúng vị trí và 3 phần tử đã được sắp là:

2	3	5

Toàn bộ quá trình sắp mảng a được biểu diễn trong bảng sau:



Như vậy trong quá trình thực hiện, để chèn 1 phần tử vào đúng vị trí của nó, ta luôn thực hiện 2 công việc: *Đẩy một phần tử sang phải 1 vị trí* hoặc *chèn phần tử cần chèn vào vị trí của nó*. Nếu gọi phần tử cần chèn là a[i] đang chứa trong biến tg và j là biến duyệt qua các phần tử đứng trước a[i] thì:

- Khi chưa hết mảng và gặp một phần tử lớn hơn phần tử cần chèn thì *đẩy nó sang phải 1 vị trí*: **while (j >= 0 && a[j] > tg) a[j+1] = a[j];**

- Ngược lại thì *chèn tg vào sau j*: **a[j+1] = tg;**

```
void SapChen(int a[100], int n)
{
        for (int i=1; i< n; i++)
        {
                int Tg = a[i]; int j = i-1;
                while (j > = 0 && a[j] > Tg)
                {
                        a[j+1] = a[j];
                        j--;
                 }
                a[j+1]=Tg;
        }
}
```

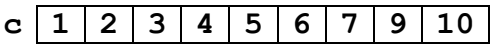
Sắp xếp bằng phương pháp chèn trong trường hợp trung bình cần $n^2/4$ lần so sánh và $n^2/8$ lần hoán vị. Trường hợp tồi nhất gấp đôi số lần này.

- **Phương pháp sắp xếp trộn: Merge sort**

Bài toán trộn: Cho mảng a gồm n phần tử và mảng b gồm m phần tử đã sắp tăng. Hãy trộn hai mảng để thu được một mảng thứ 3 cũng được sắp tăng.

Trước tiên, ta xét hai mảng a và b như ví dụ như sau:

Mảng c sau thu được sau khi trộn a và b là:



Để có được mảng c, ta làm như sau:

B1. Cho biến i xuất phát từ đầu mảng a (i=0) và biến j xuất phát từ đầu mảng b (j=0).

B2. Ta so sánh a[i] và a[j] rồi lấy phần tử nhỏ hơn trong hai phần tử đó đặt vào mảng c.

Nếu lấy a[i], ta phải tăng i lên 1 đơn vị (i++) và tương tự, nếu lấy b[j], ta tăng j lên 1 đơn vị (j++). Lặp lại B2 cho tới khi 1 trong 2 mảng đã được lấy hết.

Với mảng a,b ở trên, dễ thấy giải thuật trên sẽ dừng lại khi mảng a đã được lấy hết. Tuy nhiên, khi đó, mảng b vẫn còn các phần tử 6, 7, 9, 10 chưa được lấy. Công việc tiếp theo là chuyển toàn bộ các phần tử "còn thừa" này từ b sang c.

Hàm sau thực hiện trộn 2 mảng a, b theo thuật toán trên.

```
int c[100];
void Tron(int a[50],int n, int b[50], int m)
{
        int i=0, j=0, k=0;
        while(i<n&&j<m)
                if(a[i]<b[j])
                        {c[k]=a[i]; i++; k++;}
                else
                        {c[k]=b[j]; j++; k++;}
        // Gắn đuôi-----------------------------
        if(i<n)
                for(int t=i; t<n; t++)  {c[k]=a[t]; k++;}
        else
                for(int t=j; t<m; t++)  {c[k]=b[t]; k++;}
}
```

Dễ thấy quá trình cho i chạy trên a và j chạy trên b sẽ buộc phải dừng lại khi 1 trong 2 mảng đã được duyệt hết. Nếu không, biến i hoặc j sẽ chạy quá giới hạn của mảng a hoặc b. Khi dừng lại, một trong 2 mảng có thể chưa được duyệt hết và còn thừa một số phần tử và quá trình chuyển các phần tử này sang c được diễn ra.

Một cải tiến nhỏ giúp cho i và j không bao giờ chạy vượt quá giới hạn của 2 mảng a và b cho tới khi ta lấy xong tất cả các phần tử của a và b đặt sang c. Muốn vậy, trước khi thực hiện giải thuật trên, ta làm như sau:

- Gọi Max là phần tử lớn nhất trong số các phần tử của cả a và b.
- Chèn giá trị Max + 1 vào vị trí cuối cùng của mảng a và mảng b.

Với mảng trên, giá trị Max = 10 và hai mảng sau khi chèn Max+1 vào cuối sẽ có dạng:

a	2	3	5	11

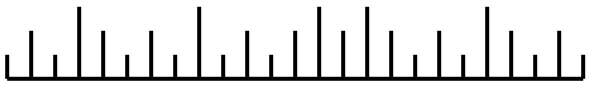
b	1	4	6	7	9	10	11

Khi i hoặc j chạy tới cuối mảng, ta gặp phải giá trị 11 là giá trị lớn hơn bất kỳ giá trị mào của hai mảng a và b ban đầu, do đó, theo giải thuật thì i và j sẽ bị dừng lại tại đó và không thể chạy vượt ra khỏi giới hạn của mảng.

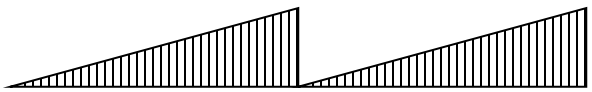
```
int c[100];
void Tron2(int a[50],int n, int b[50], int m)
{
        int Max=a[n-1];
        if(Max<b[m-1]) Max=b[m-1];
        a[n]=b[m]=Max+1;
        //-----------------------
        int i=0, j=0;
        for(int k=0; k<n+m; k++)
                if(a[i]<b[j])
                        {c[k]=a[i]; i++;}
                else
                        {c[k]=b[j]; j++;}
}
```

Ý tưởng của giải thuật sắp xếp trộn như sau:

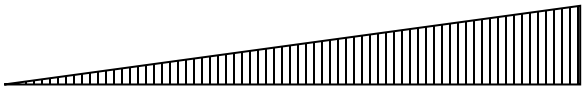
- Giả sử có mảng a chưa sắp:



- Chia mảng a làm hai phần bằng nhau và sắp trên 2 nửa của a.



- Trộn 2 nửa đã sắp để thu được mảng a cũng được sắp:



Việc sắp trên 2 nửa của a cũng được áp dụng phương pháp sắp xếp trộn này, do đó 2 nửa lại tiếp tục được chia đôi, trộn…Như vậy ta có một giải thuật đệ quy.

Giải thuật sau sử dụng mảng b làm mảng trung gian. Mỗi khi cần trộn 2 nửa của mảng a ta sao một nửa đầu của a sang b, một nửa còn lại cũng đặt vào cuối của b nhưng theo thứ tự ngược lại và tiến hành trộn với i chạy từ đầu mảng b còn j chạy từ cuối mảng b. Việc trộn 2 nửa sau khi 2 nửa đã được đặt chung vào 1 mảng b như vậy sẽ không cần phải sử dụng phần tử chặn Max+1 như trên nữa.

```
int a[100], b[100];
void MergeSort(int l, int r)
{
        if(r>l)
        {
                int m=(l+r)/2;
        MergeSort(l,m); MergeSort(m+1, r);
        //Sao chep nua dau cua a sang b
        for(int i=m; i>=l; i--) b[i]=a[i];
        //Sao chep nua con lai cua a sang b theo thu tu nguoc lai
                for(int j=m+1; j<=r;j++) b[r+m+1-j]=a[j];
        //i chay tu dau mang b, j chay tu cuoi mang b va tron
        i=l; j=r;
        for(int k=l; k<=r; k++)
                if(b[i]<b[j]) {a[k]=b[i];i++;}
                else    {a[k]=b[j];j--;}
        }
}
```

Sắp xếp bằng trộn sử dụng khoảng n lg(n) lần so sánh để sắp bất kỳ một mảng nào gồm n phần tử.

Sắp xếp bằng trộn cũng sử dụng thêm một mảng b phụ trợ có kích thước n. Người ta đã chứng minh sắp xếp bằng trộn là ổn định và không bị ảnh hưởng bởi thứ tự ban đầu của dữ liệu.

b. Bài toán tìm kiếm

Cho một mảng a gồm n phần tử và một phần tử c cùng kiểu. Hỏi c có xuất hiện trong a không?

- **Phương pháp tìm kiếm tuần tự:**

Một phương pháp đơn giản để giải quyết bài toán trên là duyệt mảng. Giải thuật được trình bày như sau:

- Sử dụng một biến đếm d = 0;
- Duyệt qua các phần tử của mảng, nếu gặp c ta tăng biến đếm: d++;

Kết thúc quá trình duyệt mảng, nếu d bằng 0 chứng tỏ c không xuất hiện trong mảng và ngược lại.

Phương pháp trên cần thiết phải quyệt qua tất cả các phần tử của mảng một cách tuần tự, do vậy, độ phức tạp của giải thuật là O(n) với n là kích thước của mảng.

Nếu mảng a đã được sắp (tăng hoặc giảm) thì do tính chất đặc biệt này, ta có thể giải quyết bài toán mà không cần duyệt qua tất cả các phần tử của mảng. Phương pháp đó gọi là tìm kiếm nhị phân.

- **Phương pháp tìm kiếm nhị phân:**

Giả sử ta cần tìm kiếm c trong một đoạn từ vị trí L tới vị trí R trong mảng a (trường hợp tìm kiếm trên toàn bộ mảng thì L=0 và R=n-1). Ta làm như sau:

- Gọi M là vị trí giữa của đoạn mảng ta đang tìm kiếm (M = (L+R)/2). Trước tiên ta tiến hành kiểm tra a[M]. Khi đó, chỉ có thể xảy ra một trong 3 trường hợp sau:

a[M] = c: kết luận c có trong mảng a và ta có thể dừng quá trình tìm kiếm.

a[M] > c: vì mảng được sắp (giả sử sắp tăng) nên rõ ràng c (nếu có) chỉ nằm trong đoạn bên phải tức [M+1, R]. Ta tiến hành lặp lại quá trình tìm kiếm trên đoạn [M+1, R], tức cận trái L=M+1.

a[M] < c: khi đó c (nếu có) chỉ nằm trong đoạn bên trái của mảng tức [L, M-1]. Ta tiến hành lặp lại quá trình tìm kiếm trên đoạn [L, M-1], tức cận phải R = M-1.

Kết thúc quá trình tìm kiếm là khi xảy ra một trong hai trường hợp:

[1]. Nếu L > R chứng tỏ C không xuất hiện trong a.

[2]. Nếu a[M] = c chứng tỏ c xuất hiện trong a tại vị trí M.

Vậy quá trình chia đôi-tìm kiếm sẽ được lặp lại nếu: (a[M] !=c && L<=R). Hàm sau thực hiện việc tìm kiếm nhị phân trên một mảng a có kích thước n với một khoá c cần tìm, sử dụng vòng lặp:

```
void TKNP_Lap(int a[100], int n, int c)
{
        int L=0, R=n-1, M;
        do
        {
                M = (L+R)/2;
                if (a[M]>c) R=M-1;
                if (a[M]<c) L=M+1;
        }
        while(a[M]!=c && L<R);
        if(a[M]==c)
                cout<<c<<" xuat hien tai "<<M;
        else
                cout<<c<<" khong xuat hien";
}
```

Việc chia đôi và tìm kiếm trên một nửa của mảng được lặp đi lặp lại cho tới khi xảy ra 1 trong 2 trường hợp : *tìm thấy* và *không tìm thấy* gợi ý cho ta một cài đặt đệ quy cho hàm này:

- **Trường hợp suy biến:** là trường hợp a[M] = c hoặc L > R. Khi đó, nếu a[M]=c hàm trả về giá trị M là vị trí cuất hiện của c trong mảng; nếu L > R thì c không xuất hiện trong mảng và hàm trả về giá trị -1.

- **Trường hợp tổng quát:** là trường hợp a[M] > c hoặc a[M] < c. Khi đó việc gọi đệ quy là cần thiết với các cận L hoặc R được thay đổi cho phù hợp.

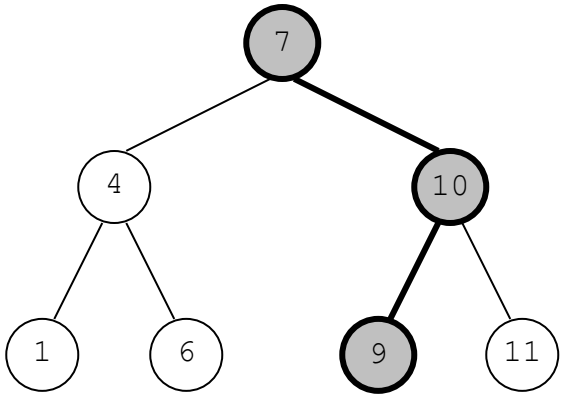
```
int TKNP_DQ(int a[100], int c, int L, int R)
{
        int M=(L+R)/2;
        if(a[M]==c)
                return M;
        else if(L>R)
                return -1;
        else //trường hợp tổng quát
          if(a[M]>c)    return TKNP_DQ(a,c,L,M-1);
          else          return TKNP_DQ(a,c,M+1,R);
}
```

Giải thuật trên giống như việc ta thăm phần tử chính giữa của đoạn mảng đang tìm kiếm, nếu không gặp c ta có thể "rẽ" sang bên trái hoặc bên phải để tìm tiếp tuỳ thuộc vào giá trị của phần tử chính giữa này. Việc này giống như việc tìm kiếm trên cây nhị phân (cây mà mỗi nút có 2 con sao cho các giá trị thuộc nhánh trái nhỏ hơn giá trị của nút và các giá trị của nhánh bên phải lớn hơn giá trị của nút).

Giả sử ta có mảng a như sau:

a	1	4	6	7	9	10	11

Các giá trị của a có thể biểu diễn trên cây nhị phân tìm kiếm như sau:



Nếu ta cần tìm một giá trị c bất kỳ, giả sử c = 9, ta chỉ cần đi hết chiều cao của cây. (Đường đi trong trường hợp này được tô đậm). Giả sử mảng có n phần tử, khi đó ta luôn có thể sử dụng một cây có chiều cao không quá $lg_2(n)$ để biểu diễn các giá trị của mảng trên cây. Do vậy, độ phức tạp trong trường của giải thuật này là $O(lg_2(n))$

II. Mảng hai chiều

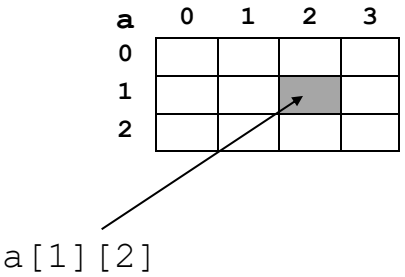
II.1. Các thao tác cơ bản trên mảng hai chiều

Mảng 2 chiều là một cấu trúc bộ nhớ gồm nhiều ô nhớ cùng tên, cùng kiểu nhưng khác nhau về chỉ số dùng để lưu trữ một bảng các phần tử cùng kiểu.

Mỗi bảng các phần tử bao gồm nhiều dòng và nhiều cột, do vậy, chỉ số của một phần tử của mảng được xác định bởi chỉ số của dòng và của cột tương ứng.

Giải sử bảng các phần tử có n dòng và m cột. Các dòng được đánh chỉ số bắt đầu từ 0: 0, 1, 2,…, n-1 và các cột cũng được đánh chỉ số từ 0: 0, 1, 2, …, m-1. Phần tử tại dòng i, cột j có chỉ số là [i][j].

Một mảng a gồm 3 dòng, 4 cột có dạng như sau:



Trên thực tế, mảng hai chiều không có dạng dòng/ cột đẹp đẽ như trên. Nó là một dãy liên tiếp các phần tử. Một mảng hai chiều n dòng m cột là một mảng một chiều có n phần tử, mà mỗi phần tử lại là một mảng một chiều có m phần tử. Do vậy, ta gọi mảng hai chiều là mảng của các mảng một chiều.

Khai báo mảng 2 chiều:

<Kiểu mảng> <Tên mảng> [<Số dòng tối đa>] [<Số cột tối đa>];

Ví dụ: int a[5][10]; khai báo một mảng a có tối đa 5 dòng và 10 cột. Mảng a có thể chứa được tối đa 50 phần tử kiểu nguyên.

Nhập mảng:

- Ta sử dụng hai vòng lặp for lồng nhau để nhập từng phần tử cho mảng. Đoạn trình sau nhập giá trị cho một mảng số a gồm n dòng, m cột:

```
for(int i=0; i<n; i++)
for(int j=0; j<m; j++)
{
        cout<< “a[”<<i<< “][”<<j<< “]=”;
        cin>>a[i][j];
}
```

Xuất mảng:

Ta cũng sử dụng hai vòng for lồng nhau và tạo dáng ma trận cho mảng khi xuất. Đoạn trình sau xuất mảng a gồm n dòng, m cột có tạo dáng ma trận:

```
for(int i=0; i<n; i++)
{
   for(int j=0; j<m; j++)
            cout<<setw(5)<<a[i][j];
    cout<<endl;
}
```

Duyệt mảng:

Việc thăm lần lượt tất cả các phần tử của mảng cần phải sử dụng 2 vòng lặp for lồng nhau:

```
for(int i=0; i<n; i++)
   for(int j=0; j<m; j++)
    {
            //Thăm phần tử có chỉ số [i][j]
    }
```

Thao tác duyệt mảng là thao tác thường xuyên sử dụng trong các bài tập về mảng. Ngay việc nhập, xuất mảng về bản chất cũng là thao tác duyệt mảng.

II.2. Các bài toán cơ bản trên mảng 2 chiều

- **Các bài toán trên ma trận:**

Về mặt hình thức, mảng có dạng ma trận. Do vậy, một dạng bài tập phổ biến về mảng là thực hiện các bài toán trên ma trận. Thuộc dạng này có các bài như: Cộng, trừ, nhân hai ma trận; chuyển vị ma trận, đổi dấu ma trận, nghịch đảo ma trận…

Ví dụ 1: Nhập vào một ma trận a (n x k) và b (k x m) các phần tử thực. Tính ma trận c = a * b.

Với hai ma trận a, b đã cho, tích của chúng là ma trận c có kích thước n x m với:

$$c[i][j] = \sum_{t=0}^{k-1} a[i][t] * b[t][j]$$

Chương trình có thể được viết như sau:

```
void nhap(int a[100][100], int n, int m, char k)
```

```
{
        for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
        {
                cout<< k <<"["<<i<< "]["<<j<< "]=";
                cin>>a[i][j];
        }

}
void xuat(int a[100][100], int n, int m)
{
        for(int i=0; i<n; i++)
        {
                for(int j=0; j<m; j++)
                        cout<< setw(5)<<a[i][j];
                cout<<endl;
        }
}
int main()
{
        int a[100][100], b[100][100], c[100][100];
        int n, m, k;
        cout<<"n="; cin>>n;
        cout<<"m="; cin>>m;
        cout<<"k="; cin>>k;
        nhap(a, n, m, 'a');
        nhap(b, m, k, 'b');

        for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
        {
                c[i][j]=0;
                for(int t=0; t<k; t++)          c[i][j]+=a[i][t]*b[t][j];
        }
        xuat(a, n, m);
        xuat(b, m, k);
        xuat(c, n, k);
}
```

- **Thống kê trên ma trận**

Một dạng bài toán khá phổ biến trên mảng hai chiều là thống kê trên ma trận. Việc thống kê một đại lượng nào đó nhằm mục đích kiểm tra một tính chất nào đó của mảng. Một vấn đề khó khăn đặt ra là xác định chính xác xem cần thống kê đại lượng nào nếu đề bài chưa nói rõ.

Ví dụ 1: Nhập vào một ma trận vuông a (n x n) các phần tử thực. Ma trận a được gọi là "hợp lệ" nếu tất cả các phần tử trên dường chéo chính đều bằng 0; các phần tử phía trên đường chéo chính đều dương và các phần tử còn lại đều âm. Hãy kiểm tra xem ma trận vừa nhập có hợp lệ không?

Ta biểu diễn điều kiện để ma trận vuông a là "hợp lệ" như sau:

$$\begin{cases} \forall i = j, a[i][j] = 0 \\ \forall i < j, a[i][j] > 0 \\ \forall i > j, a[i][j] < 0 \end{cases} \qquad \text{(IV.1)}$$

Lấy phủ định của điều kiện này ta thu được điều kiện để một ma trận vuông là "không hợp lệ" như sau:

$$\begin{cases} \exists i = j, a[i][j] \neq 0 \\ \exists i < j, a[i][j] \leq 0 \\ \exists i > j, a[i][j] \geq 0 \end{cases} \qquad \text{(IV.2)}$$

Như vậy việc giải bài toán sẽ trở nên dễ dàng. Theo đó, ta thống kê các phần tử a[i][j] thoả mãn điều kiện IV.2. Nếu không tồn tại phần tử nào thoả mãn IV.2 thì ma trận là hợp lệ.

Hàm **isInvalid**() sau trả về giá trị **true** nếu mảng a hợp lệ, ngược lại trả về giá trị **false**:

```
void nhap(int a[50][50], int n, int m)
{
        for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
        {
                cout<<"a["<<i<< "]["<<j<< "]=";
                cin>>a[i][j];
        }

}
```

```
bool isInvalid(int a[50][50], int n)
{
        for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
        {
                if(i==j && a[i][j]!=0) return false;
                if(i<j && a[i][j]<=0)  return false;
                if(i>j && a[i][j]>=0)  return false;
        }
        return true;
}

int main()
{
      int a[50][50];int n;
      cout<<"n="; cin>>n;
      nhap(a, n, n);

      if(isInvalid(a, n))
              cout<<"Ma tran la hop le !";
      else
              cout<<"Ma tran la khong hop le !";
}
```

Ví dụ 2: Có n cửa hàng kinh doanh trong m tháng. Doanh thu của mỗi cửa hàng trong mỗi tháng đều được lưu trữ trong một ma trận có n dòng, m cột. Một cửa hàng sẽ bị đóng cửa nếu doanh thu của nó giảm liên tiếp trong m-1 tháng (trừ tháng đầu tiên). Hãy cho biết cửa hàng nào trong số n cửa hàng trên sẽ bị đóng cửa?

Bài toán được giải quyết bằng cách duyệt qua từng cửa hàng (0 -> n-1). Với mỗi của hàng i, ta đếm số lần giảm doanh thu của nó. Nếu cửa hàng nào có đúng m-1 lần giảm doanh thu, cửa hàng đó sẽ bị đóng cửa.

```
void nhap(float a[50][50], int n, int m)
```

```
    {
        for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
        {
            cout<<“a[”<<i<< “][”<<j<< “]=”;
            cin>>a[i][j];
        }

    }

void Check(float a[50][50], int n, int m)
{
    for(int i=0; i<n; i++)
    {
        int d=0;
        for(int j=0; j<m-1; j++)   if(a[i][j] > a[i][j+1]) d++;

        if (d==m-1)
            cout<<"Cua hang "<<i+1<<" se bi dong cua !";
    }
}

int main()
{
    float a[50][50];  int n,m;
    cout<<"Nhap so cua hang n=";      cin>>n;
    cout<<"Nhap so thang m=";         cin>>m;
    cout<<"Nhap dt cua tung cua hang-tung thang:";
    nhap(a, n, m);
    Check(a, n, m);
}
```