TÌM KIẾM & SẮP XẾP

Tìm kiếm

- Tìm kiếm tuần tự
- Tìm kiếm nhị phân

Ví dụ – Tìm vị trí X trong dãy

- Bài toán: Tìm vị trí X trên mảng a đang có N thành phần.
- Giải pháp: Tìm tuần tự

```
//input: dãy (a, N), X
//output: Vị trí của X, -1 nếu không có

int     Search(int a[], int N, int X)
{
   for (int i = 0; i < N; i ++)
         if (a[i] == X)
               return i;
   return -1;
}
```

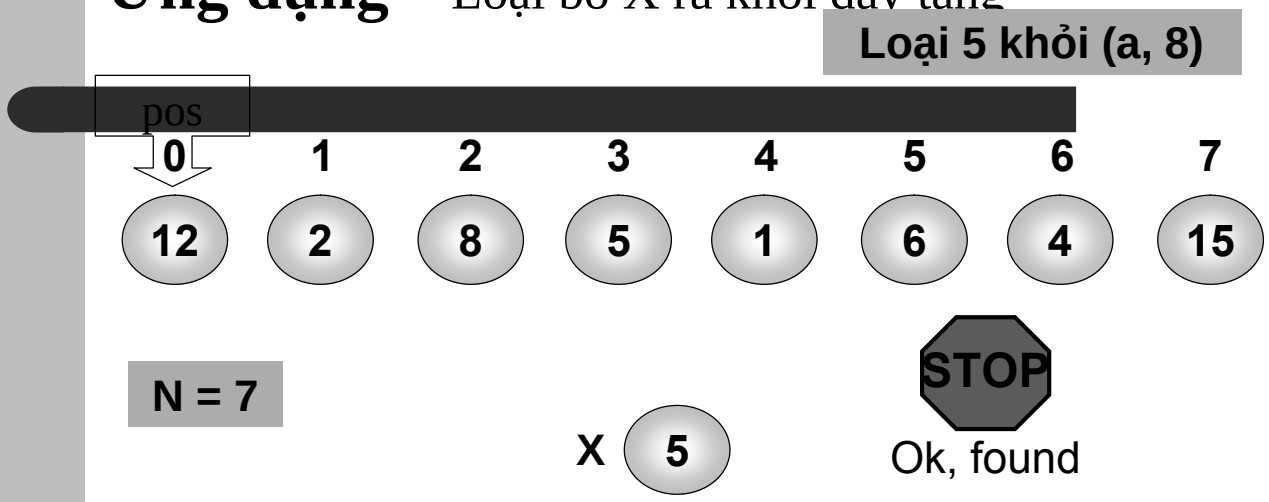
Tìm kiếm

- Tìm kiếm tuần tự (tìm kiếm tuyến tính)
  - Thời gian tồi nhất để thực hiện giải thuật:n x constant
  - Tỷ suất tăng của giải thuật là n
  - Độ phức tạp thuật toán $O(n)$

Ứng dụng – Loại bỏ một thành phần dữ liệu

- Bài toán: loại bỏ thành phần dữ liệu X ra khỏi mảng a đang có N thành phần.
- Hướng giải quyết: xác định vị trí của X, nếu tìm thấy thì dồn các phần tử ở phía sau lên để lấp vào chỗ trống. 2 trường hợp:
  - Dãy không có thứ tự: lấp phần tử cuối lên
  - Dãy đã thứ tự: dời tất cả các phần tử ở sau ví trí của X lên trước 1 vị trí.

Ứng dụng – Loại bỏ X ra khỏi dãy tăng



Tìm vị trí của 5

Dồn các vị trí 4, 5, 6, 7 lên

Ứng dụng – Loại bỏ X ra khỏi dãy tăng

```
//input: dãy (a, N), X
//output: dãy (a, N) đã loại bỏ 1 thành phần X

int    Remove(int a[], int &N, int X)
{
   int pos = Search(a, N, X);
   if (pos == -1) //không có X trong dãy
        return 0;
   N --;
   for (; (pos < N); pos ++)
        a[pos] = a[pos + 1];
   return 1;
}
```

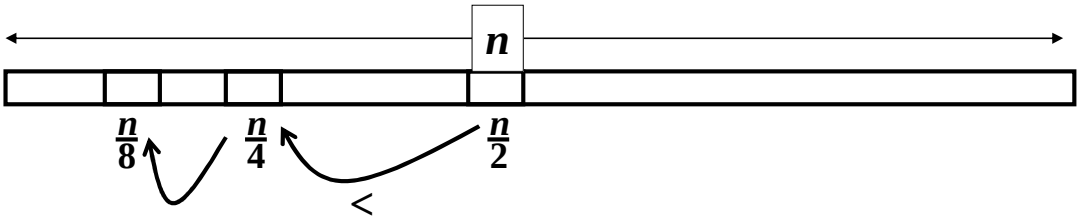
Tìm kiếm

- Tìm kiếm nhị phân
  - Bài toán: Với một dãy số đã sắp xếp, chúng ta có thể áp dụng phương pháp tìm kiếm nhị phân.

Tìm kiếm nhị phân

- Phương pháp tìm:ứng dụng cho các dãy đã sắp xếp thứ tự

  - Kiểm tra phần tử giữa của dãy
    - Nếu khóa cần tìm nhỏ hơn, tiến sang trái
    - Nếu trùng khớp: Kết thúc!
    - Nếu khóa cần tìm lớn hơn, tiến sang phải



  - Bạn có thể chia n thành hai phần: tốn thời gian $\log_2 n$
  - Bởi vậy tìm kiếm nhị phân tốn thời gian = $c \log_2 n$
  - Độ phức tạp tính toán của thuật toán là $O(\log n)$

Ví dụ – Tìm vị trí X trong dãy

- Bài toán: Tìm vị trí X trên mảng a đang có N thành phần (mảng a sắp tăng dần).
- Giải pháp: Tìm nhị phân

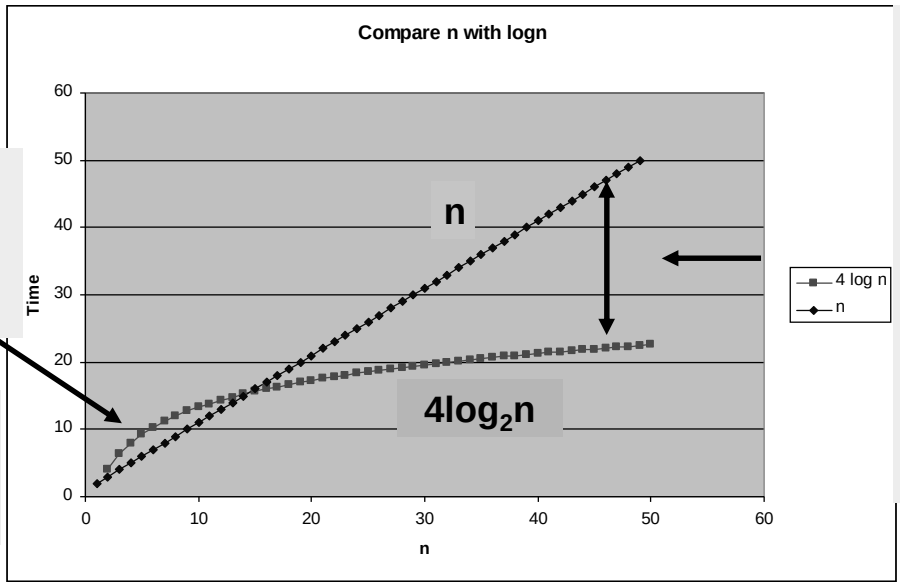
```
//input: dãy (a, N), X
//output: Vị trí của X, -1 nếu không có

int     BinarySearch(int a[], int N, int x)
{int left=0, right=N-1, mid;
 do{
     mid=(left+right)/2;
     if(x==a[mid]) return mid;// x tai mid
     else if (x<a[mid]) right=mid-1;
          else left= mid+1;
    }while (left<=right);
   return -1;
}
```

Tìm kiếm tuần tự & Tìm kiếm nhị phân

- Chọn lựa phương pháp
  - Tuần tự:Thời gian thực hiện tồi nhất: $c_1 n$
  - Tìm kiếm nhị phân:Thời gian thực hiện tồi nhất: $c_2 \log_2 n$

Bài toán nhỏ - Chúng tôi không quan tâm!

Với bài toán nhỏ: Tìm kiếm nhị phân có T(n) cao hơn



Các bài toán lớn - Nhận thấy TK nhị phân tốt hơn nhiều

Tìm kiếm

- Tìm kiếm tuần tự
  - Tỷ suất tăng: n
  - Độ phức tạp thuật toán: $O(n)$
  - Được sử dụng tốt cho các dãy chưa sắp xếp
- Tìm kiếm nhị phân
  - Cho các dãy đã sắp xếp
  - Tỷ suất tăng: $\log_2 n$
  - Độ phức tạp: $O(\log n)$

  Trường hợp dãy chưa sắp xếp: phải cộng thêm chi phí sắp xếp dãy, do đó (không nên dùng tìm kiếm nhị phân)

SẮP XẾP

Mục tiêu

Sau khi hoàn tất bài học này bạn cần phải:

- Hiểu các giải thuật sắp xếp.
- Vận dụng được giải thuật để minh họa việc sắp xếp.
- Hiểu các lưu đồ của các giải thuật sắp xếp.
- Hiểu các chương trình sắp xếp.
- Hiểu được việc đánh giá các giải thuật.

Tầm quan trọng của bài toán sắp xếp

- Sắp xếp một danh sách các đối tượng theo một thứ tự nào đó là một bài toán thường được vận dụng trong các ứng dụng tin học.
- Sắp xếp là một yêu cầu không thể thiếu trong khi thiết kế các phần mềm.
- Do đó việc nghiên cứu các phương pháp sắp xếp là rất cần thiết để vận dụng trong khi lập trình.

Sắp xếp trong và sắp xếp ngoài

- **Sắp xếp trong** là sự sắp xếp **dữ liệu được tổ chức trong bộ nhớ trong** của máy tính.
- Các đối tượng cần được sắp xếp là các mẩu tin gồm một hoặc nhiều trường. Một trong các trường được gọi là khóa (key), kiểu của nó là một kiểu có quan hệ thứ tự (như các kiểu số nguyên, số thực, chuỗi ký tự...).
- Danh sách các đối tượng cần sắp xếp sẽ là một mảng của các mẩu tin vừa nói ở trên.
- Mục đích của việc sắp xếp là tổ chức lại các mẩu tin sao cho các khóa của chúng được sắp thứ tự tương ứng với quy luật sắp xếp.
- Một cách mặc nhiên, quy luật sắp xếp là thứ tự không giảm. Khi cần sắp xếp theo thứ tự không tăng thì phải nói rõ.
- **Sắp xếp ngoài** là sự sắp xếp được sử dụng khi số lượng đối tượng cần sắp xếp lớn không thể lưu trữ trong bộ nhớ trong mà phải lưu trữ trên **bộ nhớ ngoài**.

Tổ chức dữ liệu và ngôn ngữ cài đặt

- Để trình bày các ví dụ minh họa chúng ta sẽ dùng C (Turbo C++, Version 3.0) làm ngôn ngữ thể hiện và sử dụng khai báo sau.

```
const int n = 10;
typedef  int keytype;
typedef  float othertype;
typedef  struct recordtype {
      keytype key;
      othertype otherfields;
     };
recordtype  a[n];   /* khai bao mang a co n phan tu */
```

Tổ chức dữ liệu và ngôn ngữ cài đặt (tt)

```
void Swap(recordtype *x, recordtype *y)
{
    recordtype temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

- Cần thấy rằng thủ tục Swap lấy O(1) thời gian vì chỉ thực hiện 3 lệnh gán nối tiếp nhau.

Giải thuật sắp xếp chọn (Selection Sort)

- Bước 0, chọn phần tử có khóa nhỏ nhất trong n phần tử từ a[0] đến a[n-1] và hoán vị nó với phần tử a[0].
- Bước 1, chọn phần tử có khóa nhỏ nhất trong n-1 phần tử từ a[1] đến a[n-1] và hoán vị nó với a[1].
- Tổng quát ở bước thứ i, chọn phần tử có khoá nhỏ nhất trong n-i phần tử từ a[i] đến a[n-1] và hoán vị nó với a[i].
- Sau n-1 bước này thì mảng đã được sắp xếp.

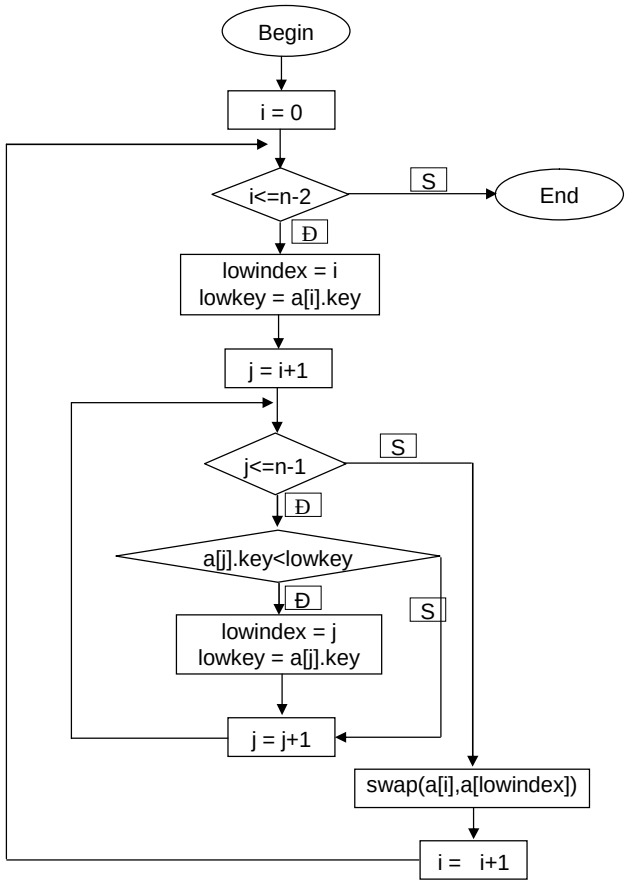
Phương pháp chọn phần tử

- Đầu tiên ta đặt khoá nhỏ nhất là khoá của a[i] (lowkey = a[i].key) và chỉ số của phần tử có khoá nhỏ nhất là i (lowindex = i).
- Xét các phần tử a[j] (với j từ i+1 đến n-1), nếu khoá của a[j] nhỏ hơn khoá nhỏ nhất (a[j].key < lowkey) thì đặt lại lại khoá nhỏ nhất là khoá của a[j] (lowkey = a[j].key) và chỉ số của phần tử có khoá nhỏ nhất là j (lowindex = j).
- Khi đã xét hết các a[j] (j>n-1) thì phần tử có khoá nhỏ nhất là a[lowindex].

Ví dụ sắp xếp chọn

Khóa / Bước	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
Ban đầu	5	6	2	2	10	12	9	10	9	3
Bước 0	2	6	5	2	10	12	9	10	9	3
Bước 1		2	5	6	10	12	9	10	9	3
Bước 2			3	6	10	12	9	10	9	5
Bước 3				5	10	12	9	10	9	6
Bước 4					6	12	9	10	9	10
Bước 5						9	12	10	9	10
Bước 6							9	10	12	10
Bước 7								10	12	10
Bước 8									10	12
Kết quả	2	2	3	5	6	9	9	10	10	12

Lưu đồ sắp xếp chọn



Chương trình sắp xếp chọn

```
void SelectionSort(void)
{       int i,j,lowindex;
        keytype lowkey;
/*1*/     for (i=0; i<=n-2; i++) {
/*2*/       lowindex = i;
/*3*/       lowkey = a[i].key;
/*4*/       for (j = i+1; j <= n-1; j++)
/*5*/         if (a[j].key < lowkey) {
/*6*/           lowkey = a[j].key;
/*7*/           lowindex = j; }
/*8*/       Swap(&a[i],&a[lowindex]);
    }
 }
```

Đánh giá sắp xếp chọn

- Hàm Swap tốn O(1).
- Toàn bộ chương trình chỉ bao gồm lệnh /*1*/. Lệnh /*1*/ chứa các lệnh "đồng cấp" /*2*/, /*3*/, /*4*/ và /*8*/, trong đó các lệnh /*2*/, /*3*/ và /*8*/ đều tốn thời gian O(1).
- Lệnh /*6*/ và /*7*/ đều tốn O(1) nên lệnh /*5*/ tốn O(1).
- Vòng lặp /*4*/ thực hiện n-i-1 lần, vì j chạy từ i+1 đến n-1, mỗi lần lấy O(1), nên lấy O(n-i-1) thời gian.
- Gọi T(n) là thời gian thực hiện của chương trình, thì T(n) là thời gian thực hiện lệnh /*1*/. Mà lệnh /*1*/ có i chạy từ 0 đến n-2 nên ta có:

$$T(n) = \sum_{i=0}^{n-2}(n - i - 1) = \frac{n(n-1)}{2} = O(n^2)$$

Giải thuật sắp xếp xen (Insertion Sort)

- Trước hết ta xem phần tử a[0] là một dãy đã có thứ tự.
- Bước 1, xen phần tử a[1] vào danh sách đã có thứ tự a[0] sao cho a[0], a[1] là một danh sách có thứ tự.
- Bước 2, xen phần tử a[2] vào danh sách đã có thứ tự a[0], a[1] sao cho a[0], a[1], a[2] là một danh sách có thứ tự.
- Tổng quát, bước i, xen phần tử a[i] vào danh sách đã có thứ tự a[0], a[1], … a[i-1] sao cho a[0], a[1],.. a[i] là một danh sách có thứ tự.
- Sau n-1 bước thì kết thúc.

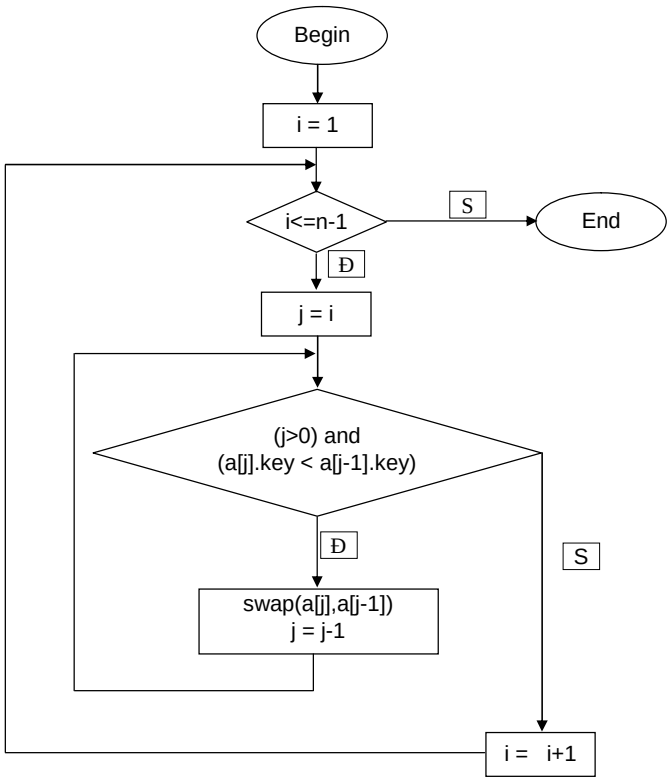
Phương pháp xen

- Phần tử đang xét a[j] sẽ được xen vào vị trí thích hợp trong danh sách các phần tử đã được sắp trước đó a[0],a[1],..a[j-1]:
- So sánh khoá của a[j] với khoá của a[j-1] đứng ngay trước nó.
- Nếu khoá của a[j] nhỏ hơn khoá của a[j-1] thì hoán đổi a[j-1] và a[j] cho nhau và tiếp tục so sánh khoá của a[j-1] (lúc này a[j-1] chứa nội dung của a[j]) với khoá của a[j-2] đứng ngay trước nó...

Ví dụ sắp xếp xen

Khóa / Bước	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	A[7]	a[8]	a[9]
Ban đầu	5	6	2	2	10	12	9	10	9	3
Bước 1	5	6								
Bước 2	2	5	6							
Bước 3	2	2	5	6						
Bước 4	2	2	5	6	10					
Bước 5	2	2	5	6	10	12				
Bước 6	2	2	5	6	9	10	12			
Bước 7	2	2	5	6	9	10	10	12		
Bước 8	2	2	5	6	9	9	10	10	12	
Bước 9	2	2	3	5	6	9	9	10	10	12

Lưu đồ sắp xếp xen



Chương trình sắp xếp xen

```
void InsertionSort(void)
 {
    int i,j;
/*1*/     for (i = 1; i<= n-1; i++) {
/*2*/         j = i;
/*3*/         while ((j>0) && (a[j].key < a[j-1].key)) {
/*4*/                 Swap(&a[j], &a[j-1]);
/*5*/                 j= j-1;
        }
    }
}
```

Đánh giá sắp xếp xen

- Các lệnh /*4*/ và /*5*/ đều lấy O(1). Vòng lặp /*3*/, trong trường hợp xấu nhất, chạy i lần (j giảm từ i đến 1), mỗi lần tốn O(1) nên /*3*/ lấy i thời gian.
- Lệnh /*2*/ và /*3*/ là hai lệnh nối tiếp nhau, lệnh /*2*/ lấy O(1) nên cả hai lệnh này lấy i.
- Vòng lặp /*1*/ có i chạy từ 1 đến n-1 nên ta có:

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

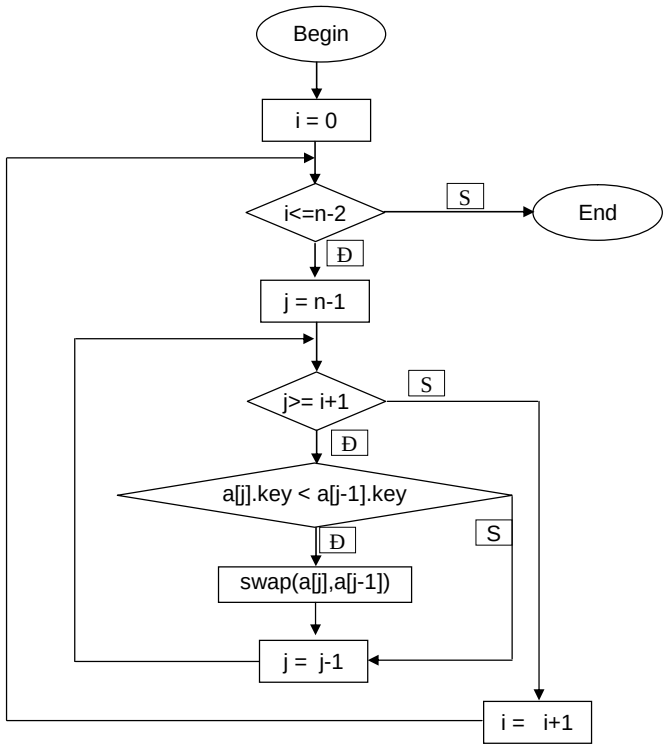
Giải thuật sắp xếp “nổi bọt” (Bubble Sort)

- Bước 1: Xét các phần tử a[j] (j giảm từ n-1 đến 1), so sánh khoá của a[j] với khoá của a[j-1]. Nếu khoá của a[j] nhỏ hơn khoá của a[j-1] thì hoán đổi a[j] và a[j-1] cho nhau. Sau bước này thì a[0] có khoá nhỏ nhất.
- Bước 2: Xét các phần tử a[j] (j giảm từ n-1 đến 2), so sánh khoá của a[j] với khoá của a[j-1]. Nếu khoá của a[j] nhỏ hơn khoá của a[j-1] thì hoán đổi a[j] và a[j-1] cho nhau. Sau bước này thì a[1] có khoá nhỏ thứ 2.
- …
- Sau n-1 bước thì kết thúc.

Ví dụ sắp xếp “nổi bọt”

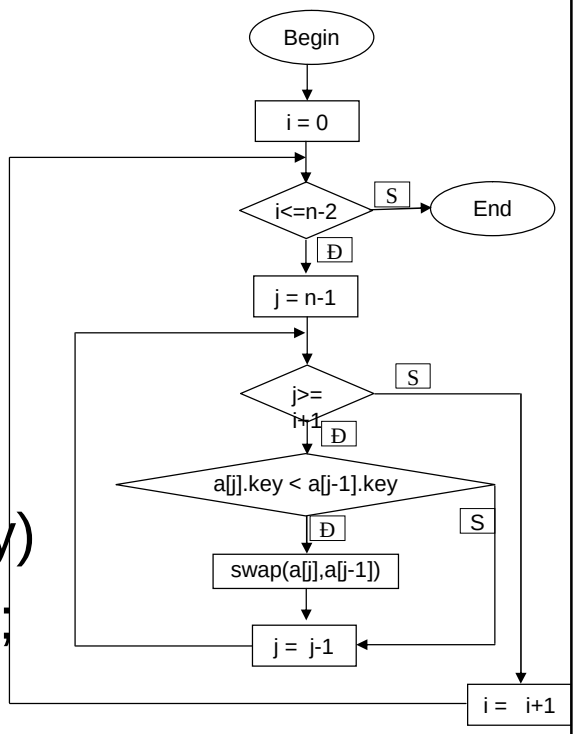
Khóa / Bước	a[0]	a[1]	a[2]	a[3]	a[4]	A[5]	a[6]	a[7]	a[8]	a[9]
Ban đầu	5	6	2	2	10	12	9	10	9	3
Bước 1	2	5	6	2	3	10	12	9	10	9
Bước 2		2	5	6	3	9	10	12	9	10
Bước 3			3	5	6	9	9	10	12	10
Bước 4				5	6	9	9	10	10	12
Bước 5					6	9	9	10	10	12
Bước 6						9	9	10	10	12
Bước 7							9	10	10	12
Bước 8								10	10	12
Bước 9									10	12
Kết quả	2	2	3	5	6	9	9	10	10	12

Lưu đồ sắp xếp nổi bọt



Chương trình sắp xếp "nổi bọt"

```
void BubbleSort(void)
{          int i,j;
/*1*/ for(i= 0; i<= n-2; i++)
/*2*/   for(j=n-1;j>=i+1; j--)
/*3*/     if (a[j].key < a[j-1].key)
/*4*/       Swap(&a[j],&a[j-1]);
}
```



Ý tưởng của QuickSort

- Chọn một giá trị khóa v làm chốt (pivot).
- Phân hoạch dãy a[0]..a[n-1] thành hai mảng con "bên trái" và "bên phải". Mảng con "bên trái" bao gồm các phần tử có **khóa nhỏ hơn chốt**, mảng con "bên phải" bao gồm các phần tử có **khóa lớn hơn hoặc bằng chốt**.
- Sắp xếp mảng con “bên trái” và mảng con “bên phải”.
- Sau khi đã sắp xếp được mảng con “bên trái” và mảng con “bên phải” thì mảng đã cho sẽ được sắp bởi vì tất cả các khóa trong mảng con “bên trái” đều nhỏ hơn các khóa trong mảng con “bên phải”.
- Việc sắp xếp các mảng con “bên trái” và “bên phải” cũng được tiến hành bằng phương pháp nói trên.
- Một mảng chỉ gồm một phần tử hoặc gồm nhiều phần tử có khóa bằng nhau thì đã có thứ tự.

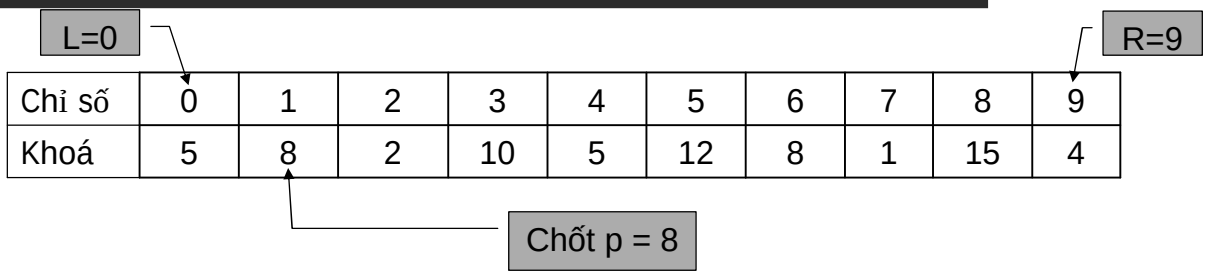
Phương pháp chọn chốt

- Chọn giá trị **khóa lớn nhất** trong hai phần tử có khóa khác nhau đầu tiên kể từ trái qua.
- Nếu mảng chỉ gồm một phần tử hay gồm nhiều phần tử có khóa bằng nhau thì không có chốt**.**
- **Ví dụ:** Chọn chốt trong các mảng sau
  - Cho mảng gồm các phần tử có khoá là 6, 6, 5, 8, 7, 4, ta chọn chốt là 6 (khoá của phần tử đầu tiên).
  - Cho mảng gồm các phần tử có khoá là 6, 6, 7, 5, 7, 4, ta chọn chốt là 7 (khoá của phần tử thứ 3).
  - Cho mảng gồm các phần tử có khoá là 6, 6, 6, 6, 6, 6 thì không có chốt (các phần tử có khoá bằng nhau).
  - Cho mảng gồm một phần tử có khoá là 6 thì không có chốt (do chỉ có một phần tử).

Phương pháp phân hoạch

- Để phân hoạch mảng ta dùng 2 "con nháy" L và R trong đó L từ bên trái và R từ bên phải.
- Ta cho L chạy sang phải cho tới khi gặp phần tử có khóa ≥ chốt
- Cho R chạy sang trái cho tới khi gặp phần tử có khóa < chốt.
- Tại chỗ dừng của L và R nếu L < R thì hoán vị a[L],a[R].
- Lặp lại quá trình dịch sang phải, sang trái của 2 "con nháy" L và R cho đến khi L > R.
- Khi đó L sẽ là điểm phân hoạch, cụ thể là a[L] là phần tử đầu tiên của mảng con “bên phải”.

Ví dụ về phân hoạch



Ví dụ về phân hoạch

L=1

R=9

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	8	2	10	5	12	8	1	15	4

Chốt p = 8

Ví dụ về phân hoạch

L=1

R=9

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	10	5	12	8	1	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=2

R=9

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	10	5	12	8	1	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=3 R=9

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	10	5	12	8	1	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=3 R=8

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	10	5	12	8	1	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=3 R=7

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	10	5	12	8	1	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=3 R=7

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=4 R=7

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=5 R=7

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=5 R=6

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

Ví dụ về phân hoạch

L=5 R=5

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

Ví dụ về phân hoạch

R=4 L=5

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

0	1	2	3	4
5	4	2	1	5

5	6	7	8	9
12	8	10	15	8

Giải thuật QuickSort

- Để sắp xếp mảng a[i]..a[j] ta làm các bước sau:
  - **Xác định chốt**.
  - **Phân hoạch** mảng đã cho thành hai mảng con a[i]..a[k-1] và a[k]..a[j].
  - Sắp xếp mảng a[i]..a[k-1] (Đệ quy).
  - Sắp xếp mảng a[k]..a[j] (Đệ quy).
- Đệ quy sẽ dừng khi không còn tìm thấy chốt.

Ví dụ về QuickSort

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	8	2	10	5	12	8	1	15	4

Chốt p = 8

5	4	2	1	5	12	8	10	15	8

Chốt p = 5

Ví dụ về QuickSort

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

5 1	4	2	1 5	5	12	8	10	15	8

Chốt p = 5

1	4	2	5	5

Chốt p = 4

Ví dụ về QuickSort

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

5 1	4	2	1 5	5	12	8	10	15	8

Chốt p = 5 | Chốt p = 12

1	4 2	2 4	5	5

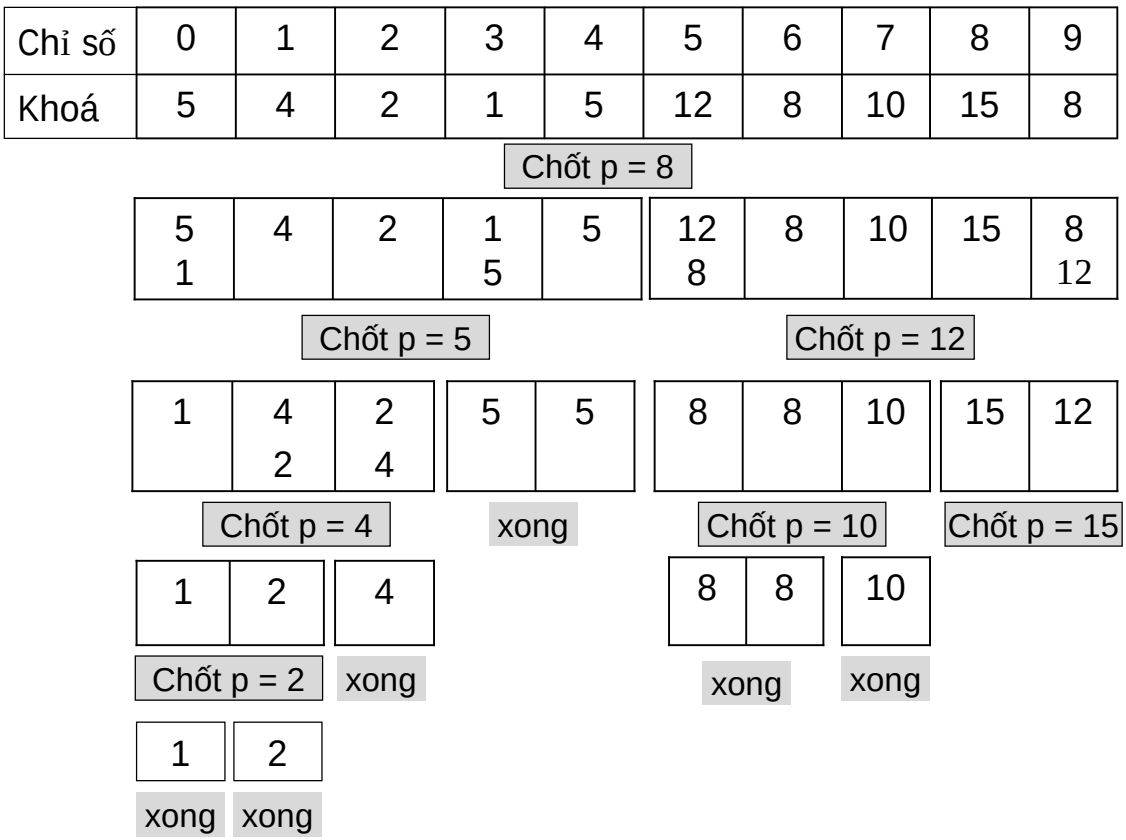
Chốt p = 4 | xong

1	2	4

Chốt p = 2 | xong

1	2
xong	xong

Ví dụ về QuickSort



Ví dụ về QuickSort

Chỉ số	0	1	2	3	4	5	6	7	8	9
Khoá	5	4	2	1	5	12	8	10	15	8

Chốt p = 8

5 1	4	2	1 5	5	12 8	8	10	15	8 12

Chốt p = 5 | Chốt p = 12

1	4 2	2 4	5	5	8	8	10	15 12	12 15

Chốt p = 4 | xong | Chốt p = 10 | Chốt p = 15

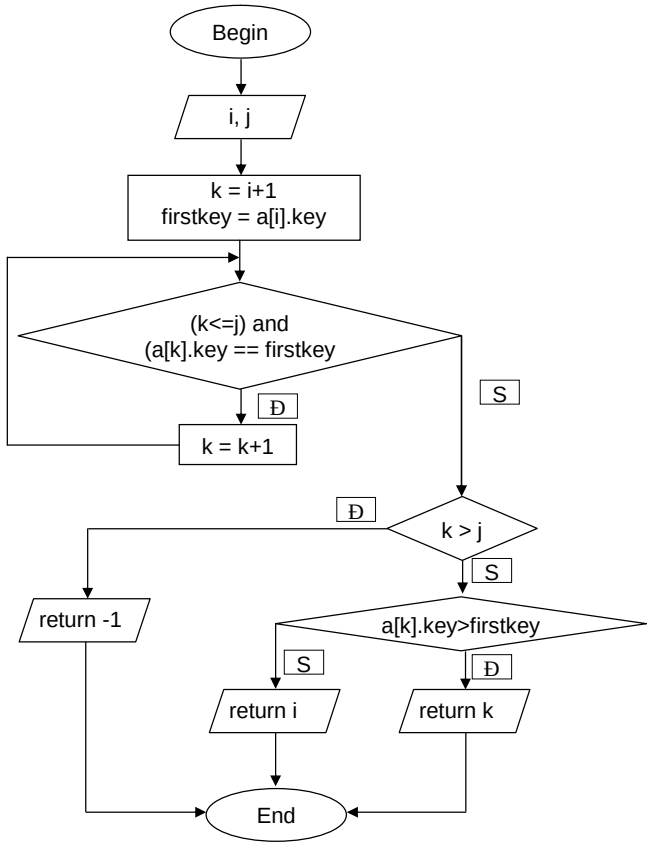
1	2	4	8	8	10	12	15

Chốt p = 2 | xong | xong | xong | xong | xong

1	2

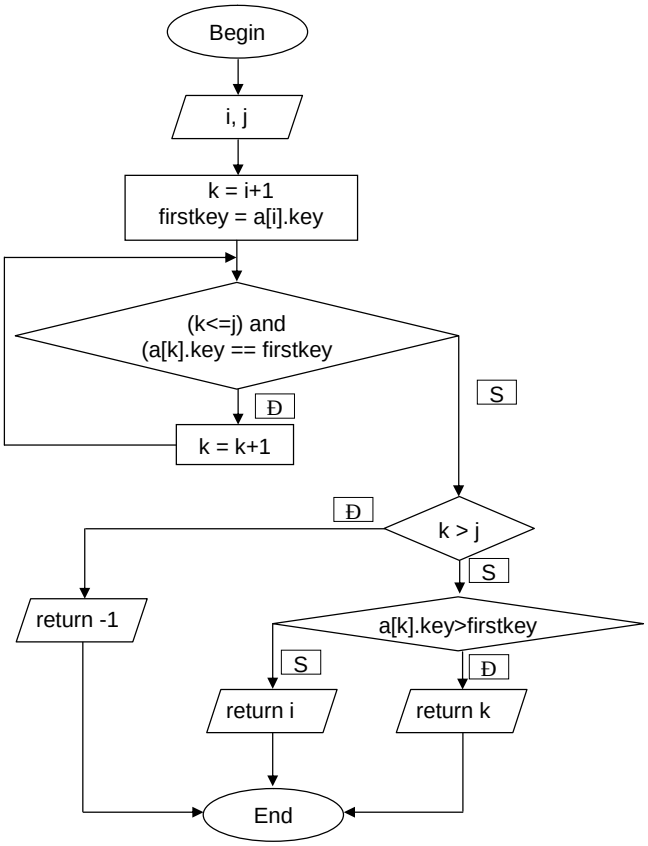
xong | xong

Lưu đồ hàm FindPivot



Chương trình hàm FindPivot

```
int FindPivot(int i,int j)
{  keytype  firstkey;
   int k ;
   k = i+1;
   firstkey = a[i].key;
   while ( (k <= j) && (a[k].key ==
     firstkey) ) k++;
   if (k > j)  return -1;
   else
      if (a[k].key>firstkey) return k;
     else return i;
}
```

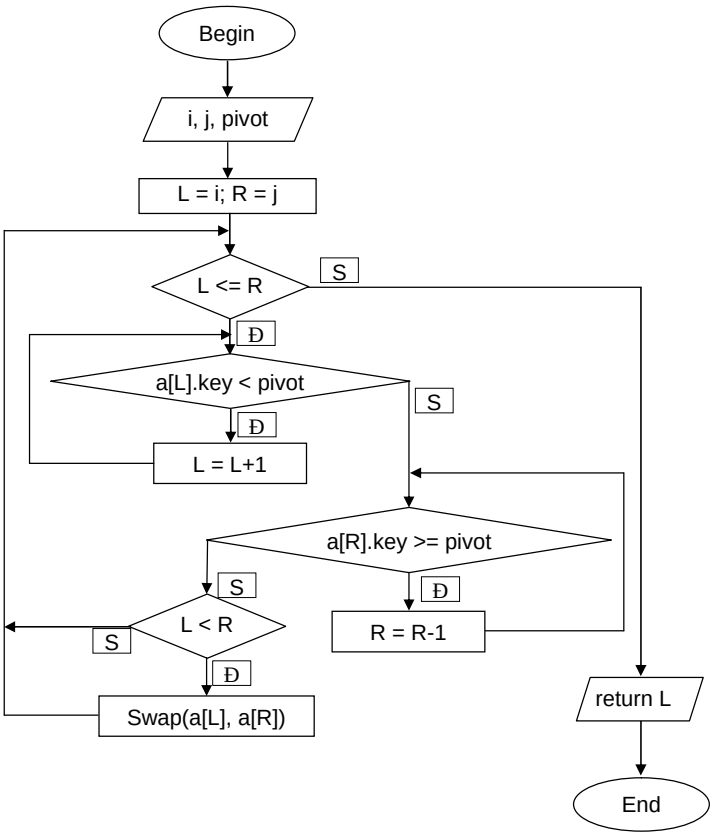


Phân tích hàm FindPivot

```
int FindPivot(int i,int j)
{       keytype  firstkey;
        int k ;
/*1*/   k = i+1;
/*2*/   firstkey = a[i].key;
/*3*/   while ( (k <= j) && (a[k].key == firstkey) ) k++;
/*4*/   if (k > j)  return -1;
        else
/*5*/     if (a[k].key>firstkey) return k; else return i;
}
```

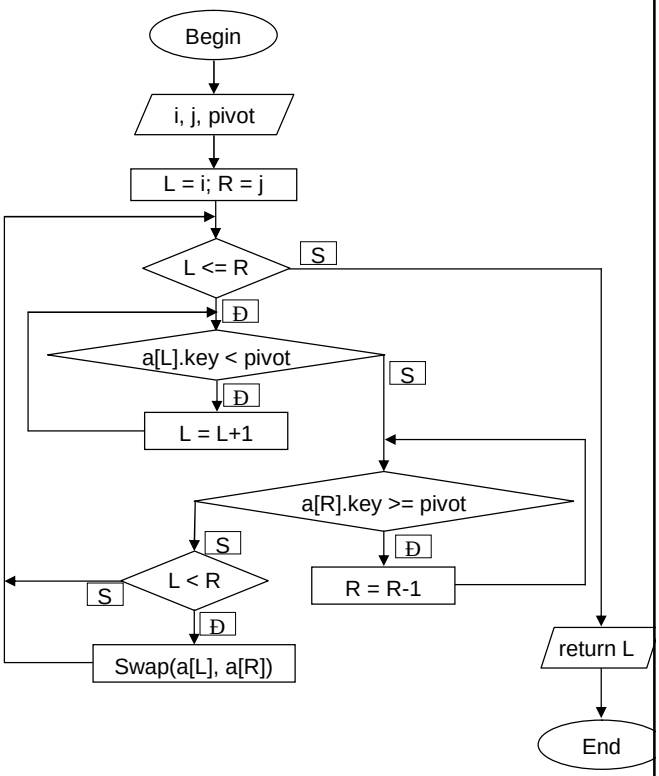
- /*1*/, /*2*/, /*3*/ và /*4*/ nối tiếp nhau.
- Lệnh WHILE là tốn nhiều thời gian nhất.
- Trong trường hợp xấu nhất thì k chạy từ i+1 đến j, tức là vòng lặp thực hiện j-i lần, mỗi lần O(1) do đó tốn j-i
- Đặc biệt khi i=0 và j=n-1, thì thời gian thực hiện là n-1 hay T(n) = O(n).

Lưu đồ hàm Partition



Hàm Partition

```
int Partition(int i,int j, keytype pivot)
{           int L,R;
/*1*/       L = i;
/*2*/       R = j;
/*3*/       while (L <= R) {
/*4*/         while (a[L].key < pivot)  L++;
/*5*/         while (a[R].key >= pivot) R--;
/*6*/         if (L<R) Swap(&a[L],&a[R]);
            }
/*7*/       return L;  /*Tra ve diem phan
    hoach*/
}
```



Phân tích hàm Partition

```
int Partition(int i,int j, keytype pivot)
{          int L,R;
/*1*/      L = i;
/*2*/      R = j;
/*3*/      while (L <= R) {
/*4*/        while (a[L].key < pivot)  L++;
/*5*/        while (a[R].key >= pivot) R--;
/*6*/        if (L<R) Swap(&a[L],&a[R]);
           }
/*7*/      return L;
}
```

- /*1*/, /*2*/, /*3*/ và /*7*/ nối tiếp nhau
- Thời gian thực hiện của /*3*/ là lớn nhất.
- Các lệnh /*4*/, /*5*/ và /*6*/ là thân của lệnh /*3*/, trong đó lệnh /*6*/ lấy O(1).
- Lệnh /*4*/ và lệnh /*5*/ thực hiện việc di chuyển L sang phải và R sang trái cho đến khi L và R gặp nhau, thực chất là duyệt các phần tử mảng, mỗi phần tử một lần, mỗi lần tốn O(1) thời gian. Tổng cộng việc duyệt này tốn j-i thời gian.
- Vòng lặp /*3*/ thực chất là để xét xem khi nào thì duyệt xong, do đó thời gian thực hiện của lệnh /*3*/ chính là thời gian thực hiện của hai lệnh /*4*/ và /*5*/ và do đó là j-i.
- Đặc biệt khi i=0 và j=n-1 ta có T(n) = O(n).

Hàm QuickSort

```
void QuickSort(int i,int j)
{ keytype pivot;
  int pivotindex, k;
  pivotindex = FindPivot(i,j);
  if (pivotindex != -1) {
      pivot = a[pivotindex].key;
      k = Partition(i,j,pivot);
      QuickSort(i,k-1);
      QuickSort(k,j);
  }
}
```

Đánh giá QuickSort (Trường hợp xấu nhất)

- Giả sử các giá trị khóa của mảng khác nhau nên hàm FindPivot luôn tìm được chốt và đệ quy chỉ dừng khi kích thước bài toán bằng 1.
- Gọi T(n) là thời gian thức hiện việc QuickSort mảng có n phần tử.
- Thời gian tìm chốt và phân hoạch mảng là O(n) = n.
- Khi n = 1, thủ tục QuickSort chỉ làm một nhiệm vụ duy nhất là gọi hàm Findpivot với kích thước bằng 1, hàm này tốn thời gian O(1) =1.
- Trong trường hợp xấu nhất, phân hoạch lệch.
- Khi đó ta có thể thành lập phương trình đệ quy như sau:

$$T(n) = \begin{cases} 1 & \text{nêu } n = 1 \\ T(n-1) + T(1) + n & \text{nêu } n > 1 \end{cases}$$

Giải PT này ta được T(n) =O(n^2)

Đánh giá QuickSort (Trường hợp tốt nhất)

- Trong trường hợp tốt nhất khi ta chọn được chốt sao cho hai mảng con có kích thước bằng nhau và bằng n/2.
- Lúc đó ta có phương trình đệ quy như sau:

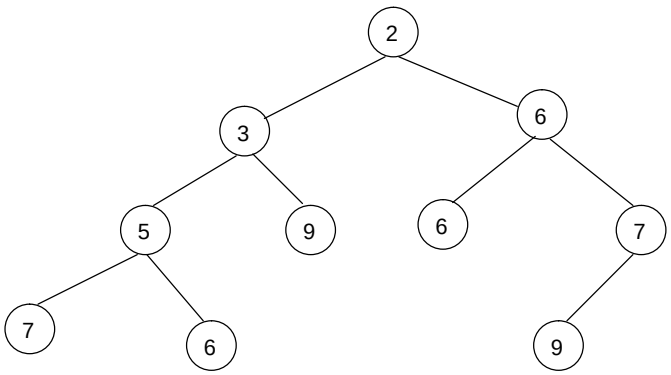
$$T(n) = \begin{cases} 1 & \text{nêu } n = 1 \\ 2T(\frac{n}{2}) + n & \text{nêu } n > 1 \end{cases}$$

Giải PT này ta được T(n) =O(nlogn)

HeapSort: Định nghĩa Heap

- Cây sắp thứ tự bộ phận hay còn gọi là heap là cây nhị phân mà giá trị tại mỗi nút (khác nút lá) đều không lớn hơn giá trị của các con của nó.
- Ta có nhận xét rằng nút gốc của cây sắp thứ tự bộ phận có giá trị nhỏ nhất.

Ví dụ về heap



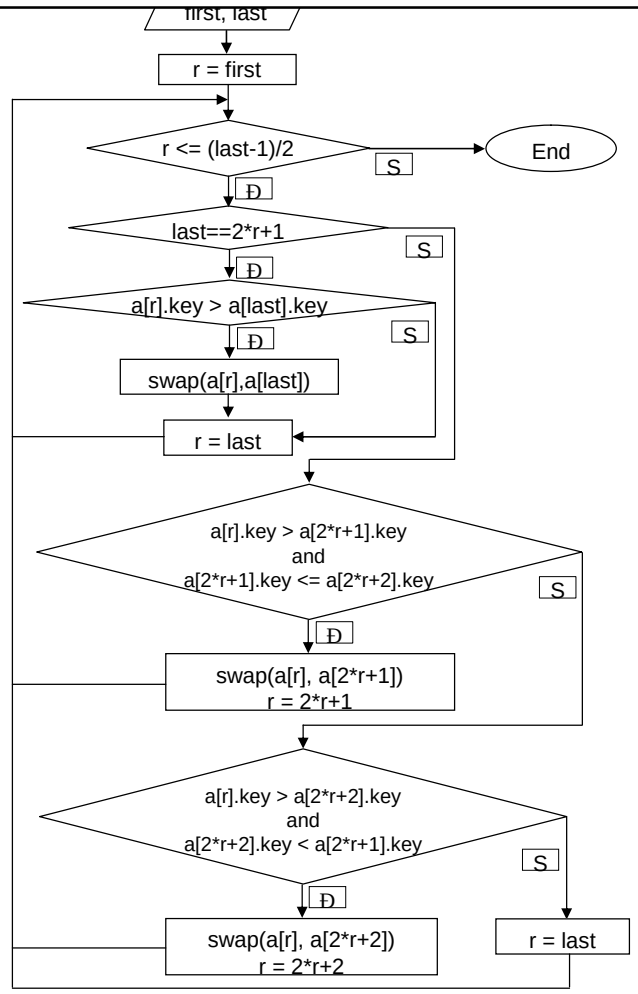
HeapSort : Ý tưởng giải thuật

- (1) Xem mảng ban đầu là một cây nhị phân. Mỗi nút trên cây lưu trữ một phần tử mảng, trong đó a[0] là nút gốc và mỗi nút không là nút lá a[i] có con trái là a[2i+1] và con phải là a[2i+2]. Với cách tổ chức này thì cây nhị phân thu được sẽ có các nút trong là các nút a[0], …, a[(n-2)/2]. Tất cả các nút trong đều có 2 con, ngoại trừ nút a[(n-2)/2] có thể chỉ có một con trái (trong trường hợp n là một số chẵn).
- (2) Sắp xếp cây ban đầu thành một heap căn cứ vào giá trị khoá của các nút.
- (3) Hoán đổi nút gốc a[0] cho cho nút lá cuối cùng.
- (4) Sắp lại cây sau khi đã bỏ đi nút lá cuối cùng để nó trở thành một heap mới.
- Lặp lại quá trình (3) và (4) cho tới khi cây chỉ còn một nút. Nút này cùng với các nút lá đã bỏ đi tạo thành một mảng sắp theo thứ tự giảm.

Thiết kế hàm PushDown

- PushDown nhận vào 2 tham số first và last để đẩy nút first xuống.
- Giả sử a[first],..,a[last] đã đúng vị trí của một heap, ngoại trừ a[first]. PushDown dùng để đẩy phần tử a[first] xuống đúng vị trí của nó trong cây.
- Xét a[first], có các khả năng có thể xẩy ra:
  - Nếu a[first] chỉ có một con trái và nếu khoá của nó lớn hơn khoá của con trái (a[first].key > a[2*first+1].key) thì hoán đổi a[first] cho con trái của nó và kết thúc.
  - Nếu a[first] có khoá lớn hơn con trái của nó và khoá của con trái không lớn hơn khoá của con phải thì hoán đổi a[first] cho con trái của nó, việc này có thể gây ra tình trạng con trái sẽ không đúng vị trí nên phải xem xét lại con trái để có thể đẩy xuống.
  - Ngược lại, nếu a[first] có khoá lớn hơn khoá của con phải của nó và khoá của con phải nhỏ hơn khoá của con trái thì hoán đổi a[first] cho con phải của nó, việc này có thể gây ra tình trạng con phải sẽ không đúng vị trí nên phải tiếp tục xem xét con phải để có thể đẩy xuống.
  - Nếu tất cả các trường hợp trên đều không xẩy ra thì a[first] đã đúng vị trí.

Lưu đồ hàm PushDown



Chương trình hàm Pushdown

```
void PushDown(int first,int last)
{ int r;
  r= first;
  while (r <= (last-1)/2)
   if (last == 2*r+1) {
     if (a[r].key > a[last].key) Swap(&a[r],&a[last]);
     r = last;
   } else
   if ((a[r].key>a[2*r+1].key) && (a[2*r+1].key<=a[2*r+2].key))
   {
      Swap(&a[r],&a[2*r+1]);
      r = 2*r+1 ;
   } else
      if ((a[r].key>a[2*r+2].key) && (a[2*r+2].key<a[2*r+1].key))
      {
       Swap(&a[r],&a[2*r+2]);
       r = 2*r+2 ;
      }
      else
       r = last;
}
```

Phân tích hàm PushDown

- Ta xét PushDown(0,n-1), tức là PushDown trên cây có n nút.
- PushDown chỉ duyệt trên một nhánh nào đó của cây nhị phân, tức là sau mỗi lần lặp thì số nút còn lại một nửa. Một cách cụ thể, trước hết PushDown trên cây có n nút; Sau lần lặp thứ nhất, PushDown trên cây có n/2 nút; Sau lần lặp thứ hai, PushDown trên cây có n/4 nút;… Tổng quát, Sau lần lặp thứ i, PushDown trên cây có $n/2^i$ nút.
- Như vậy, trong trường hợp xấu nhất (luôn phải thực hiện việc đẩy xuống) thì lệnh lặp while phải thực hiện i lần sao chon/2^i = 1 tức là i=logn (i=logn là số lần lặp của lệnh while, trong trường hợp xấu nhất). Mà mỗi lần lặp chỉ thực hiện một lệnh IF với thân lệnh IF là lời gọi Swap và lệnh gán, do đó tốn O(1) = 1 đơn vị thời gian.
- Từ đó ta thấy PushDown lấy O(logn) để đẩy xuống một nút trong cây có n nút.

Chương trình hàm HeapSort

```
void HeapSort(void)
     {  int i;
/*1*/  for(i = (n-2)/2; i>=0; i--)
/*2*/            PushDown(i,n-1);
/*3*/  for(i = n-1; i>=2; i--) {
/*4*/            Swap(&a[0],&a[i]);
/*5*/            PushDown(0,i-1);
         }
/*6*/  Swap(&a[0],&a[1]);
     }
```

Phân tích HeapSort

- Hàm PushDown lấy O(logn).
- Trong HeapSort,
  - Vòng lặp /*1*/-/*2*/ lặp (n-2)/2+1 lần mà mỗi lần lấy O(logn) nên thời gian thực hiện /*1*/-/*2*/ là O(n logn).
  - Vòng lặp /*3*/-/*5*/ lặp n-2 lần, mỗi lần lấy O(logn) nên thời gian thực hiện của /*3*/-/*5*/ là O(n logn).
- Thời gian thực hiện HeapSort là O(nlogn).

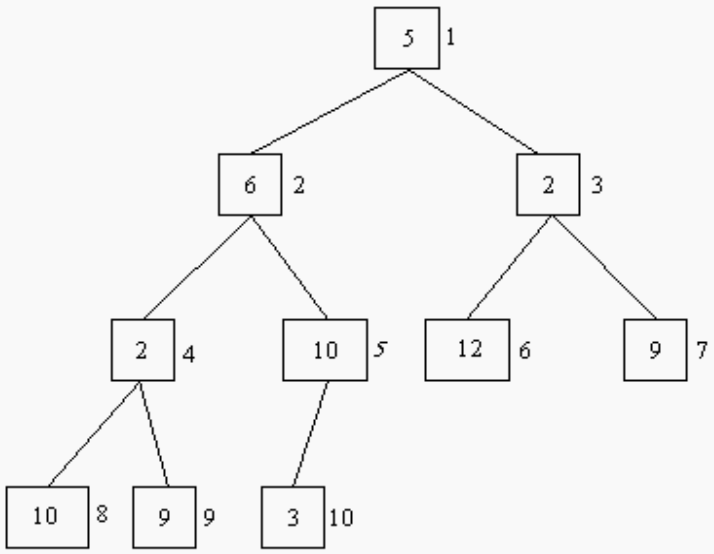
HeapSort: Trình bày bằng bảng

Khóa / Bước	a[0]	a[1]	a[2]	a[3]	a[4]	A[5]	a[6]	a[7]	a[8]	a[9]
Ban đầu	5	6	2	2	10	12	9	10	9	3
Tạo Heap										

Ví dụ 2-6: Sắp xếp mảng bao gồm 10 phần tử có khoá là các số nguyên như trong các ví dụ trước:

Khoá	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
Ban đầu	5	6	2	2	10	12	9	10	9	3

Mảng này được xem như là một cây nhị phân ban đầu như sau:

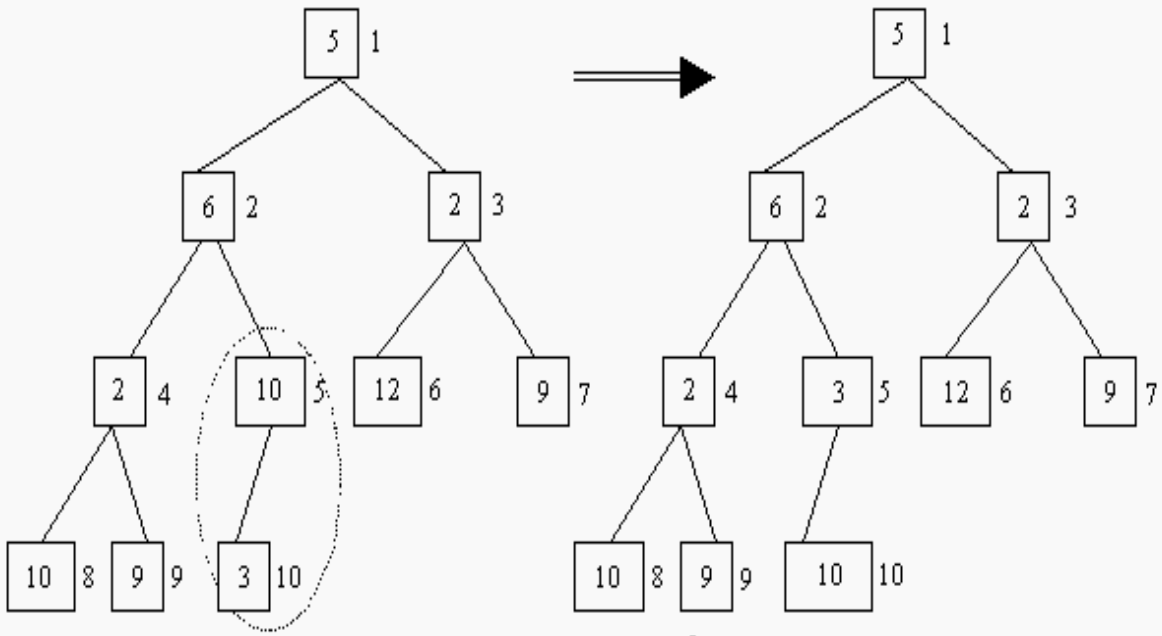


Hình 2-6: Cây ban đầu

Trong cây trên, giá trị ghi trong các nút là khoá của các phần tử mảng, giá trị ghi bên ngoài các nút là chỉ số của các phần tử mảng.

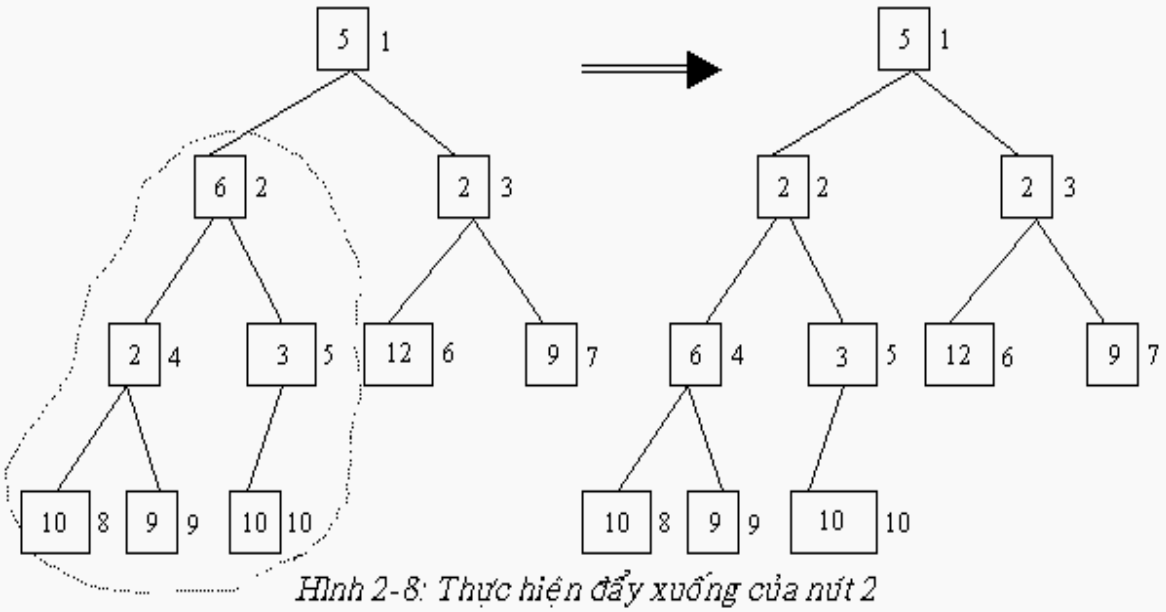
Việc sắp xếp cây này thành một heap sẽ bắt đầu từ việc đẩy xuống nút a[5] (vì 5 = 10 DIV 2)

Xét nút 5 ta thấy a[5] chỉ có một con trái và giá trị khóa tương ứng của nó lớn hơn con trái của nó nên ta đổi hai nút này cho nhau và do con trái của a[5] là a[10] là một nút lá nên việc đẩy xuống của a[5] kết thúc.



Hình 2-7: Thực hiện đẩy xuống của nút 5

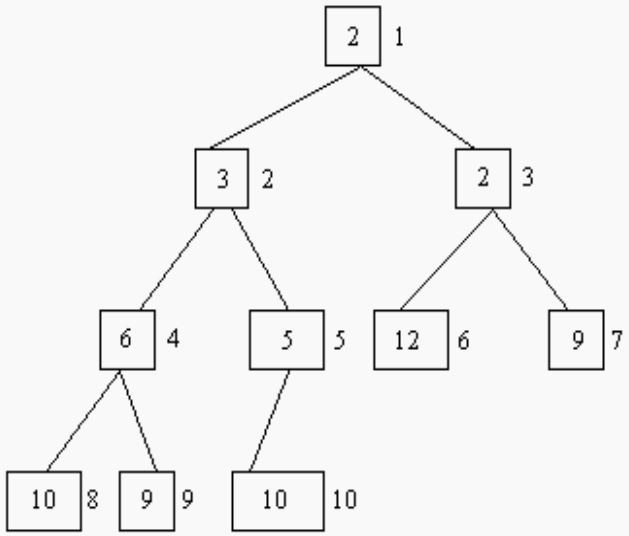
Nút 4 và nút 3 đã đúng vị trí nên không phải thực hiện sự hoán đổi. Tại nút 2, giá trị khóa của nó lớn hơn khoá con trái và khoá của con trái nhỏ hơn khoá của con phải nên ta hóan đổi nút 2 cho con trái của nó (nút 4), sau khi hoán đổi, ta xét lại nút 4, thấy nó vẫn đúng vị trí nên kết thúc việc đẩy xuống của nút 2.



Hình 2-8: Thực hiện đẩy xuống của nút 2

Cuối cùng ta xét nút 1, ta thấy giá trị khoá của nút 1 lớn hơn khoá của con trái và con trái có khoá bằng khoá của con phải nên hóan đổi nút 1 cho con trái của nó (nút 2).

Sau khi thực hiện phép hóan đổi nút 1 cho nút 2, ta thấy nút 2 có giá trị khoá lớn hơn khoá của con phải của nó (nút 5) và con phải có khoá nhỏ hơn khoá của con trái nên phải thực hiện phép hoán đổi nút 2 cho nút 5. Xét lại nút 5 thì nó vẫn đúng vị trí nên ta được cây mới trong hình 2-9.



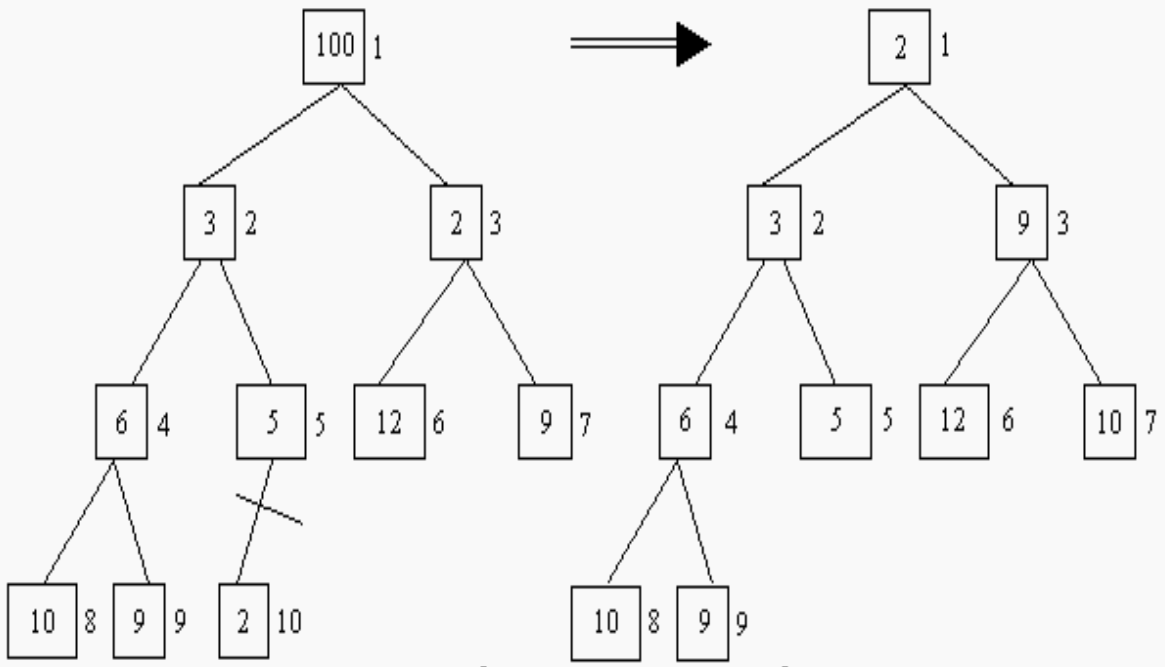
Hình 2-9: Cây ban đầu đã được tạo thành heap

Cây này là một heap tương ứng với mảng

Khoá	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
Heap	2	3	2	6	5	12	9	10	9	10

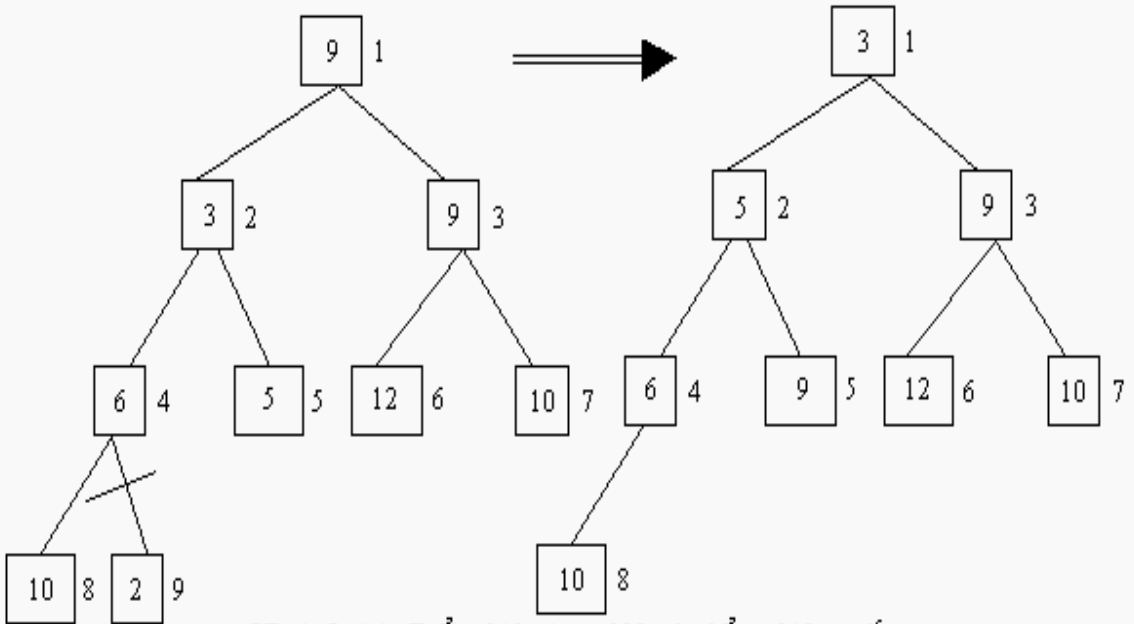
Từ heap đã có ở trên, hoán đổi a[1] cho a[10] ta có a[10] là nút có khóa nhỏ nhất, cắt bỏ nút a[10] ra khỏi cây. Như vậy phần cuối mảng chỉ gồm một phần tử a[10] đã được sắp.

Thực hiện việc đẩy a[1] xuống đúng vị trí của nó trong cây a[1]..a[9] ta được cây:



Hình 2-10: Hoán đổi a[1] cho a[10] và đẩy a[1] xuống

Hoán đổi a[1] cho a[9] và cắt a[9] ra khỏi cây. Ta được phần cuối mảng bao gồm hai phần tử a[9]..a[10] đã được sắp. Thực hiện việc đẩy a[1] xuống đúng vị trí của nó trong cây a[1]..a[8] ta được cây



Hình 2-11: Đổi a[1] cho a[9] và đẩy a[1] xuống

Tiếp tục quá trình trên ta sẽ được một mảng có thứ tự giảm.