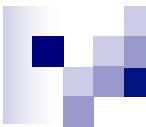




Các thuật toán sắp xếp và tìm kiếm

THAM KHẢO

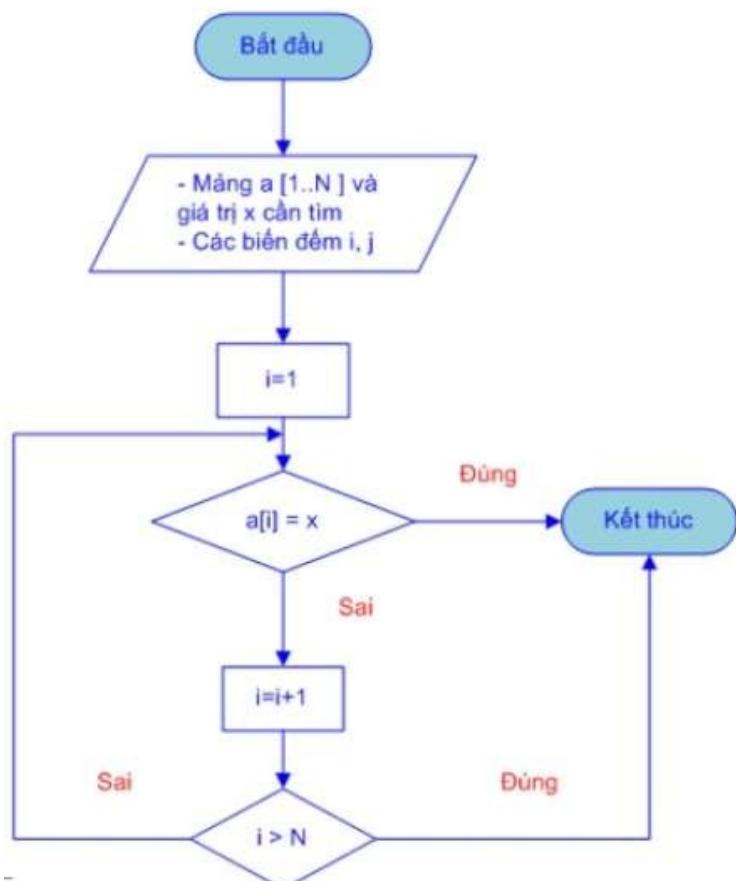


LINEAR SEARCH

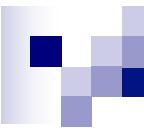
- Thường sử dụng với các mảng chưa được sắp xếp.
- Ý tưởng: tiến hành so sánh số x với các phần tử trong mảng cho đến khi gặp được phần tử có giá trị cần tìm.
- Input:
 - Mảng $A[n]$ và giá trị cần tìm x
- Output:
 - True: nếu tìm thấy;
 - False: nếu không tìm thấy.

LINEAR SEARCH

■ Thuật toán:

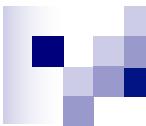


```
bool LinearSearch(int *p, int length, int x)
{
    for (int i = 0; i < length; i++)
        if (*(p + i) == length)
            return true;
    return false;
}
```



BINARY SEARCH

- Thường dùng cho mảng đã sắp xếp thứ tự;
- Ý tưởng:
 - Tìm kiếm kiểu tra “**từ điển**”;
 - Tìm cách giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong mảng đã được sắp xếp;
 - Tại mỗi bước, so sánh x với phần tử nằm ở vị trí giữa của mảng tìm kiếm hiện hành:
 - Nếu x nhỏ hơn thì sẽ tìm kiếm ở nửa mảng trước;
 - Ngược lại, tìm kiếm ở nửa mảng sau.



BINARY SEARCH

Giả sử chúng ta cần tìm vị trí của giá trị 31 trong một mảng bao gồm các giá trị như hình dưới đây bởi sử dụng Binary Search:

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Đầu tiên, chúng ta chia mảng thành hai nửa theo phép toán sau:

$$\text{chỉ-mục-giữa} = \text{ban-đầu} + (\text{cuối} + \text{ban-đầu}) / 2$$

Với ví dụ trên là $0 + (9 - 0) / 2 = 4$ (giá trị là 4.5). Do đó 4 là chỉ mục giữa của mảng.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

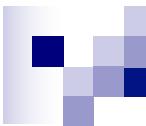
Bây giờ chúng ta so sánh giá trị phần tử giữa với phần tử cần tìm. Giá trị phần tử giữa là 27 và phần tử cần tìm là 31, do đó là không kết nối. Bởi vì giá trị cần tìm là lớn hơn nên phần tử cần tìm sẽ nằm ở mảng con bên phải phần tử giữa.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Chúng ta thay đổi giá trị ban-đầu thành chỉ-mục-giữa + 1 và lại tiếp tục tìm kiếm giá trị chỉ-mục-giữa.

$$\text{ban-đầu} = \text{chỉ-mục-giữa} + 1$$

$$\text{chỉ-mục-giữa} = \text{ban-đầu} + (\text{cuối} + \text{ban-đầu}) / 2$$



BINARY SEARCH

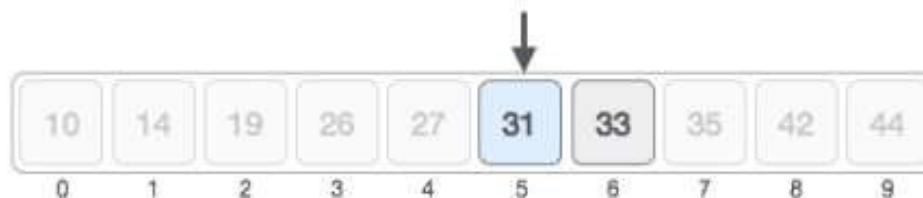
Bây giờ chỉ mục giữa của chúng ta là 7. Chúng ta so sánh giá trị tại chỉ mục này với giá trị cần tìm.



Giá trị tại chỉ mục 7 là không kết nối, và ngoài ra giá trị cần tìm là nhỏ hơn giá trị tại chỉ mục 7 do đó chúng ta cần tìm trong mảng con bên trái của chỉ mục giữa này.



Tiếp tục tìm chỉ-mục-giữa lần nữa. Lần này nó có giá trị là 5.



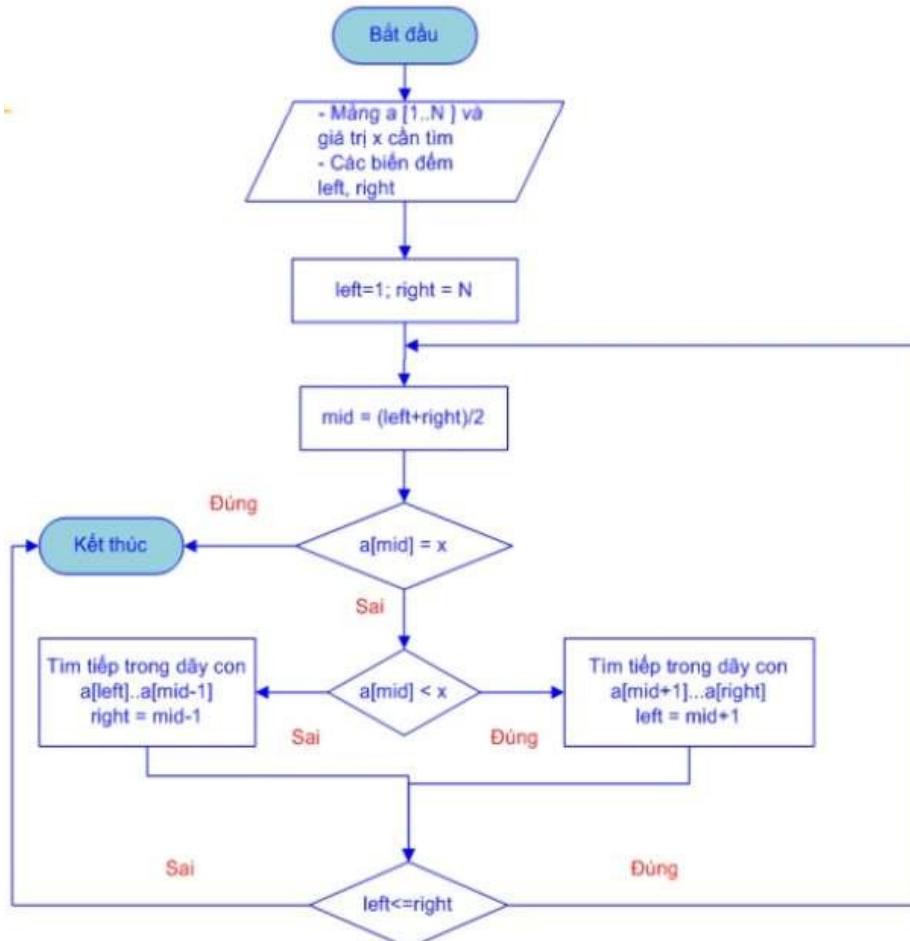
So sánh giá trị tại chỉ mục 5 với giá trị cần tìm và thấy rằng nó kết nối.



Do đó chúng ta kết luận rằng giá trị cần tìm 31 được lưu giữ tại vị trí chỉ mục 5.

BINARY SEARCH

■ Thuật toán:



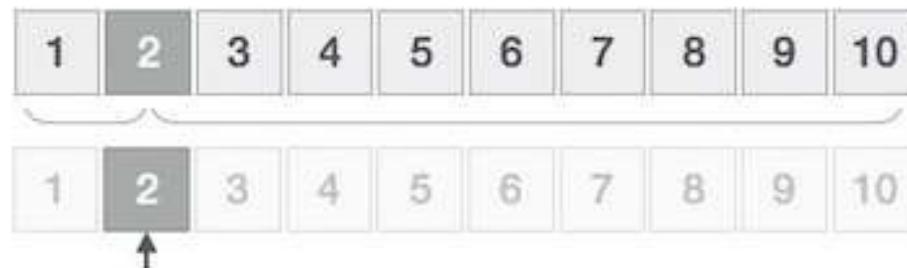
```
bool BinarySearch(int *p, int length, int x)
{
    int left = 0, right = length - 1;
    while (left <= right)
    {
        int k = (left + right) / 2;
        if (*(p + k) == x)
            return true;
        if (*(p + k) > x)
            right = k - 1;
        else
            left = k + 1;
    }
    return false;
}
```

INTERPOLATION SEARCH

- Biến thể của Binary Search, sử dụng với mảng đã được sắp xếp;
- Ý tưởng:
 - Step 1:
 - Binary Search: $pos = left + (right - left)/2;$
 - Cải tiến:
$$pos = left + (X - T[left]) * (right - left)/(T[right] - T[left])$$
 - Step 2:
 - Kiểm $A[pos]$ nếu bằng X thì pos là vị trí cần tìm;
 - Nếu nhỏ hơn X thì ta tăng $left$ lên một đơn vị và tiếp tục thực hiện lại bước 1;
 - Nếu lớn hơn X thì ta giảm $right$ một đơn vị và tiếp tục thực hiện lại bước 1.

INTERPOLATION SEARCH

Tìm kiếm nội suy tìm kiếm một phần tử cụ thể bằng việc tính toán vị trí dò (Probe Position). Ban đầu thì vị trí dò là vị trí của phần tử nằm ở giữa nhất của tập dữ liệu.



Nếu tìm thấy kết nối thì chỉ mục của phần tử được trả về. Để chia danh sách thành hai phần, chúng ta sử dụng phương thức sau:

$$mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])$$

Trong đó:

A = danh sách

Lo = chỉ mục thấp nhất của danh sách

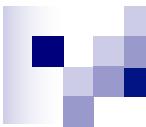
Hi = chỉ mục cao nhất của danh sách

A[n] = giá trị được lưu giữ tại chỉ mục n trong danh sách

Nếu phần tử cần tìm có giá trị lớn hơn phần tử ở giữa thì phần tử cần tìm sẽ ở mảng con bên phải phần tử ở giữa và chúng ta lại tiếp tục tính vị trí dò; nếu không phần tử cần tìm sẽ ở mảng con bên trái phần tử ở giữa. Tiến trình này tiếp tục diễn ra trên các mảng con cho tới khi kích cỡ của mảng con giảm về 0.

INTERPOLATION SEARCH

```
bool InterpolationSearch(int *p, int length, int x)
{
    int left = 0;
    int right = length - 1;
    while (left <= right && x >= *(p + left) && x <= *(p + right))
    {
        int val1 = (x - *(p + left)) / (*(p + right) - *(p + left));
        int val2 = right - left;
        int pos = left + val1 * val2;
        if (*(p + pos) == x)
            return true;
        if (*(p + pos) < x)
            left = pos + 1;
        else
            right = pos - 1;
    }
    return false;
}
```



BUBBLE SORT

- Sắp xếp nổi bọt;
- Ý tưởng: xuất phát từ đầu mảng, so sánh 2 phần tử cạnh nhau để đưa phần tử nhỏ hơn lên trước; sau đó lại xét cặp tiếp theo cho đến khi tiến về đầu mảng. Nhờ vậy, ở lần xử lý thứ i sẽ tìm được phần tử ở vị trí đầu mảng là i.
 - Step 1: $i = 0$
 - Step 2: $j = n - 1$
 - Trong khi $j > i$ thực hiện: nếu $a[j] < a[j - 1]$: hoán vị $a[j]$ & $a[j - 1]$, $j++$;
 - Step 3: $i++$
 - Nếu $i > n - 1$ thì dừng, ngược lại, lặp lại step 2.

BUBBLE SORT

Giả sử chúng ta có một mảng không có thứ tự gồm các phần tử như dưới đây. Bây giờ chúng ta sử dụng giải thuật sắp xếp nổi bọt để sắp xếp mảng này.

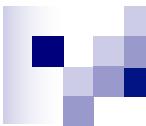


Giải thuật sắp xếp nổi bọt bắt đầu với hai phần tử đầu tiên, so sánh chúng để kiểm tra xem phần tử nào lớn hơn.



Trong trường hợp này, 33 lớn hơn 14, do đó hai phần tử này đã theo thứ tự. Tiếp đó chúng ta so sánh 33 và 27.





BUBBLE SORT

Chúng ta thấy rằng 33 lớn hơn 27, do đó hai giá trị này cần được tráo đổi thứ tự.



Mảng mới thu được sẽ như sau:



Tiếp đó chúng ta so sánh 33 và 35. Hai giá trị này đã theo thứ tự.



Sau đó chúng ta so sánh hai giá trị kế tiếp là 35 và 10.



Vì 10 nhỏ hơn 35 nên hai giá trị này chưa theo thứ tự.



BUBBLE SORT

Tráo đổi thứ tự hai giá trị. Chúng ta đã tiến tới cuối mảng. Vậy là sau một vòng lặp, mảng sẽ trông như sau:



Để đơn giản, tiếp theo mình sẽ hiển thị hình ảnh của mảng sau từng vòng lặp. Sau lần lặp thứ hai, mảng sẽ trông giống như:



Sau mỗi vòng lặp, ít nhất một giá trị sẽ di chuyển tới vị trí cuối. Sau vòng lặp thứ 3, mảng sẽ trông giống như:

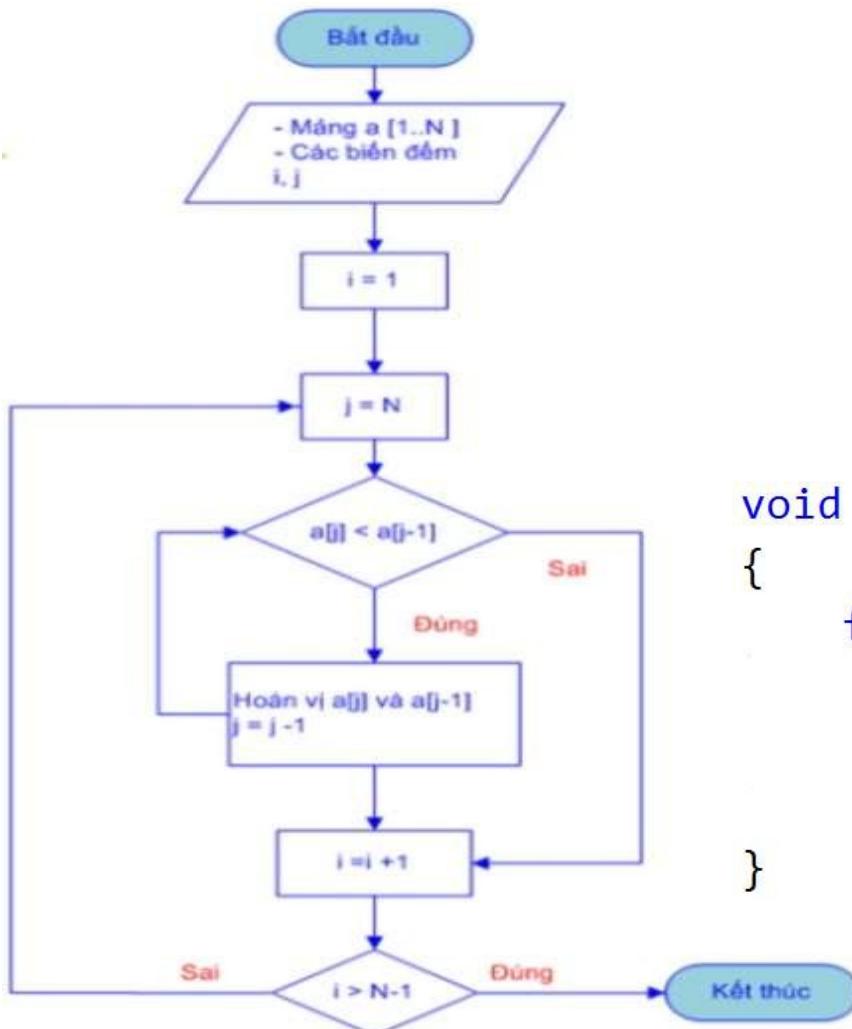


Và khi không cần tráo đổi thứ tự phần tử nào nữa, giải thuật sắp xếp nổi bọt thấy rằng mảng đã được sắp xếp xong.



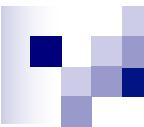
BUBBLE SORT

■ Lưu đồ giải thuật



	3	10	4	6	2	6	15	3	9	7
Bước 1	2	3	10	4	6	3	6	15	7	9
Bước 2		3	3	10	4	6	6	7	15	9
Bước 3			3	4	10	6	6	7	9	15
Bước 4				4	6	10	6	7	9	15
Bước 5					6	6	10	7	9	15
Bước 6						6	7	10	9	15
Bước 7							7	9	10	15
Bước 8								9	10	15
Bước 9									10	15
Kết quả	2	3	3	4	6	6	7	9	10	15

```
void BubbleSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = length - 1; j > i; j--)
            if (*(p + j) < *(p + j - 1))
                Swap(*(p + j - 1), *(p + j));
}
```



SELECTION SORT

- Sắp xếp chọn;
- Ý tưởng:
- Chọn phần tử nhỏ nhất trong n phần tử ban đầu của mảng, đưa phần tử này về vị trí đầu tiên của mảng; sau đó loại nó ra khỏi danh sách sắp xếp;
- Mảng hiện hành còn $n - 1$ phần tử của mảng ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho mảng hiện hành đến khi mảng hiện hành còn 1 phần tử.

SELECTION SORT



Từ vị trí đầu tiên trong danh sách đã được sắp xếp, toàn bộ danh sách được duyệt một cách liên tục. Vị trí đầu tiên có giá trị 14, chúng ta tìm toàn bộ danh sách và thấy rằng 10 là giá trị nhỏ nhất.



Do đó, chúng ta thay thế 14 với 10. Sau một vòng lặp, giá trị 10 thay thế cho giá trị 14 tại vị trí đầu tiên trong danh sách đã được sắp xếp. Chúng ta tráo đổi hai giá trị này.



Tại vị trí thứ hai, giá trị 33, chúng ta tiếp tục quét phần còn lại của danh sách theo thứ tự từng phần tử.



Chúng ta thấy rằng 14 là giá trị nhỏ nhất thứ hai trong danh sách và nó nên xuất hiện ở vị trí thứ hai. Chúng ta tráo đổi hai giá trị này.

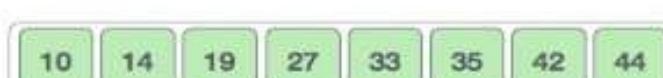
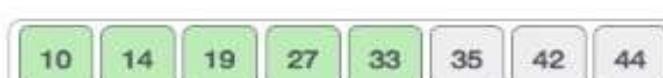
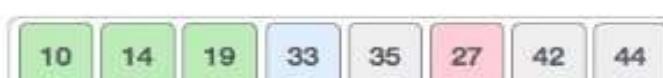
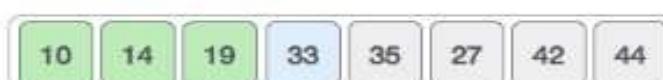


SELECTION SORT

Sau hai vòng lặp, hai giá trị nhỏ nhất đã được đặt tại phần đầu của danh sách đã được sắp xếp.

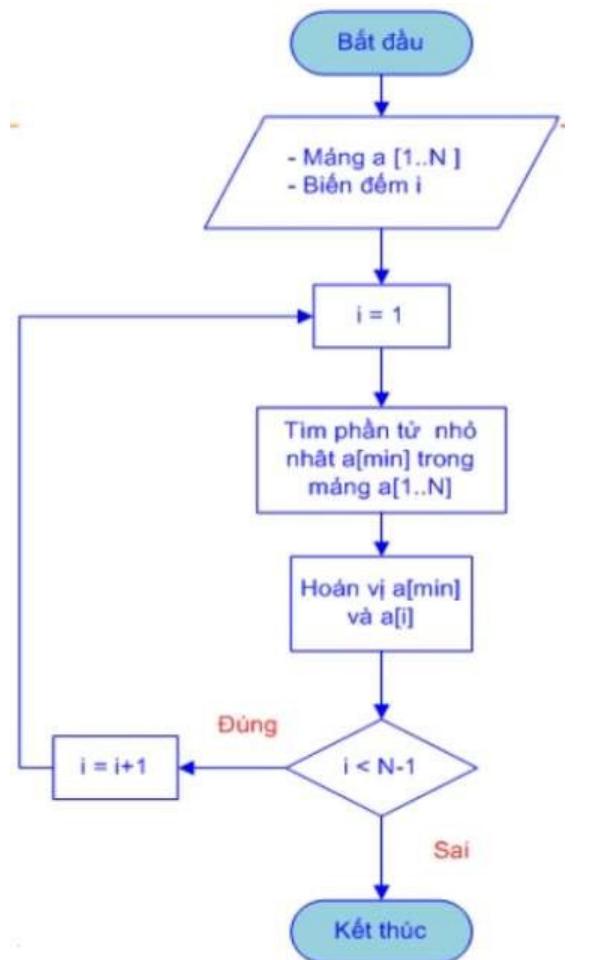


Tiến trình tương tự sẽ được áp dụng cho phần còn lại của danh sách. Các hình dưới minh họa cho các tiến trình này.

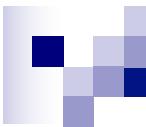


SELECTION SORT

■ Giải thuật:



```
void SelectionSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (*(p + i) > *(p + j))
                Swap(*(p + i), *(p + j));
}
```



INSERTION SORT

■ Sắp xếp chèn

Ý tưởng:

Ví dụ chúng ta có một mảng gồm các phần tử không có thứ tự:



Giải thuật sắp xếp chèn so sánh hai phần tử đầu tiên:

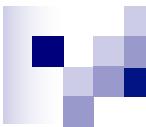


Giải thuật tìm ra rằng cả 14 và 33 đều đã trong thứ tự tăng dần. Nay 14 là trong danh sách con đã qua sắp xếp.



Giải thuật sắp xếp chèn tiếp tục di chuyển tới phần tử kế tiếp và so sánh 33 và 27.





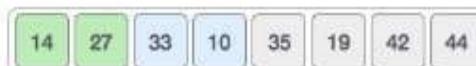
INSERTION SORT



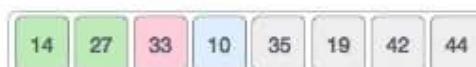
Giải thuật sắp xếp chèn trao đổi vị trí của 33 và 27. Đồng thời cũng kiểm tra tất cả phần tử trong danh sách con đã sắp xếp. Tại đây, chúng ta thấy rằng trong danh sách con này chỉ có một phần tử 14 và 27 là lớn hơn 14. Do vậy danh sách con vẫn giữ nguyên sau khi đã trao đổi.



Bây giờ trong danh sách con chúng ta có hai giá trị 14 và 27. Tiếp tục so sánh 33 với 10.



Hai giá trị này không theo thứ tự.



Vì thế chúng ta trao đổi chúng.



Việc trao đổi dẫn đến 27 và 10 không theo thứ tự.



Vì thế chúng ta cũng trao đổi chúng.



Chúng ta lại thấy rằng 14 và 10 không theo thứ tự.

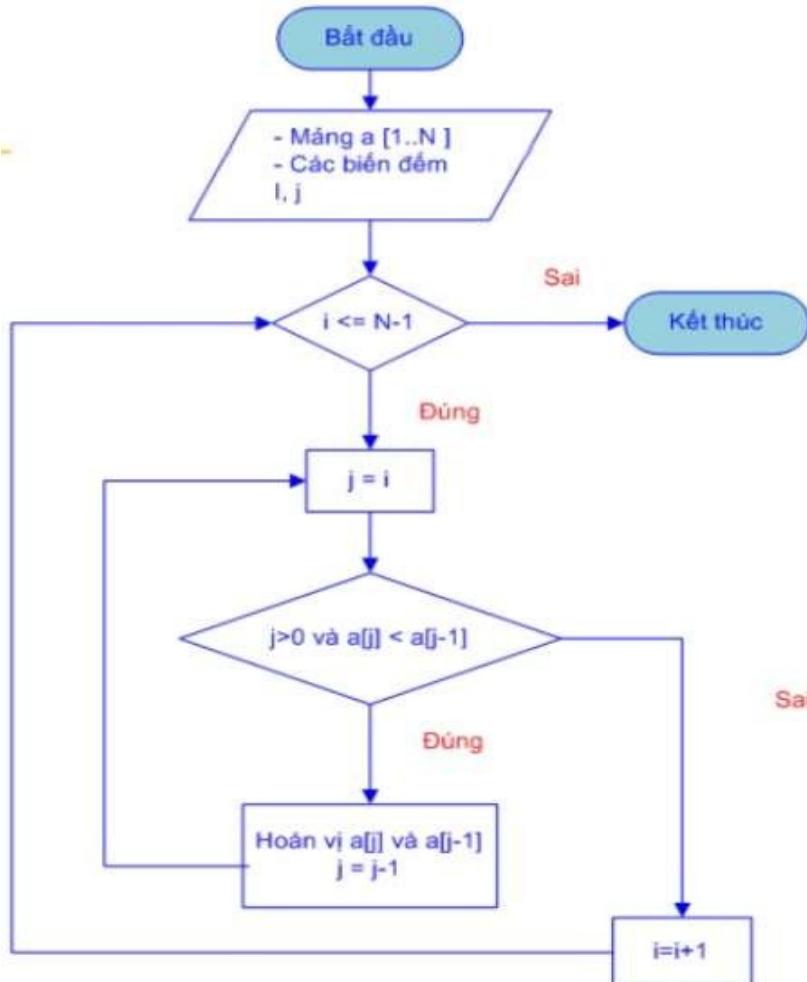


Và chúng ta tiếp tục trao đổi hai số này. Cuối cùng, sau vòng lặp thứ 3 chúng ta có 4 phần tử.



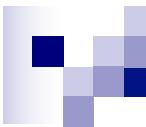
INSERTION SORT

■ Giải thuật:



```
void InsertionSort(int *p, int length)
{
    for (int i = 0; i < length; i++)
        for (int j = i; j > 0; j--)
            if (*(p + j) < *(p + j - 1))
                Swap(*(p + j), *(p + j - 1));
}
```

	3	7	22	3	1	5	8	4	3	9
Bước 0	3									
Bước 1	3	7								
Bước 2	3	7	22							
Bước 3	3	3	7	22						
Bước 4	1	3	3	7	22					
Bước 5	1	3	3	5	7	22				
Bước 6	1	3	3	5	7	8	22			
Bước 7	1	3	3	4	5	7	8	22		
Bước 8	1	3	3	3	4	5	7	8	22	
Bước 9	1	3	3	3	4	5	7	8	9	22

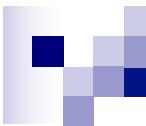


MEGRE SORT

- Sắp xếp trộn;

- Ý tưởng:

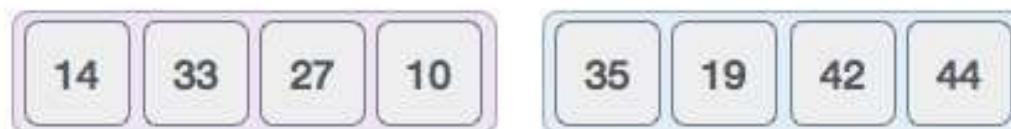
- Chia mảng lớn thành những mảng con nhỏ hơn;
 - Tiếp tục chia cho đến khi mảng con nhỏ nhất là 1 phần tử;
 - Tiến hành so sánh 2 mảng con có cùng mảng cơ sở (vừa so sánh, vừa sắp xếp & ghép) cho đến khi gộp thành 1 mảng duy nhất



MEGRE SORT



Đầu tiên, giải thuật sắp xếp trộn chia toàn bộ mảng thành hai nửa. Tiến trình chia này tiếp tục diễn ra cho đến khi không còn chia được nữa và chúng ta thu được các giá trị tương ứng biểu diễn các phần tử trong mảng. Trong hình dưới, đầu tiên chúng ta chia mảng kích cỡ 8 thành hai mảng kích cỡ 4.



Tiến trình chia này không làm thay đổi thứ tự các phần tử trong mảng ban đầu. Bây giờ chúng ta tiếp tục chia các mảng này thành 2 nửa.

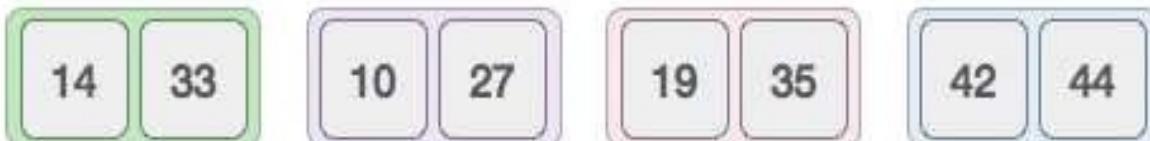


Tiến hành chia tiếp cho tới khi không còn chia được nữa.



MEGRE SORT

Đầu tiên chúng ta so sánh hai phần tử trong mỗi list và sau đó tổ hợp chúng vào trong một list khác theo cách thức đã được sắp xếp. Ví dụ, 14 và 33 là trong các vị trí đã được sắp xếp. Chúng ta so sánh 27 và 10 và trong list khác chúng ta đặt 10 ở đầu và sau đó là 27. Tương tự, chúng ta thay đổi vị trí của 19 và 35. 42 và 44 được đặt tương ứng.



Vòng lặp tiếp theo là để kết hợp từng cặp list một ở trên. Chúng ta so sánh các giá trị và sau đó hợp nhất chúng lại vào trong một list chứa 4 giá trị, và 4 giá trị này đều đã được sắp thứ tự.



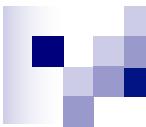
Sau bước kết hợp cuối cùng, danh sách sẽ trông giống như sau:



MEGRE SORT

```
void MergeSort(int *p, int left, int right)
{
    if (right > left)
    {
        int mid;
        mid = (left + right) / 2;
        MergeSort(p, left, mid);
        MergeSort(p, mid + 1, right);
        Merge(p, left, mid, right);
    }
}
```

```
void Merge(int *p, int left, int mid, int right)
{
    int *temp, i = left, j = mid + 1;
    temp = new int[right - left + 1];
    for (int k = 0; k <= right - left; k++)
    {
        if (*(p + i) < *(p + j))
        {
            *(temp + k) = *(p + i);
            i++;
        }
        else
        {
            *(temp + k) = *(p + j);
            j++;
        }
        if (i == mid + 1)
        {
            while (j <= right)
            {
                k++;
                *(temp + k) = *(p + j);
                j++;
            }
            break;
        }
        if (j == right + 1)
        {
            while (i <= mid)
            {
                k++;
                *(temp + k) = *(p + i);
                i++;
            }
            break;
        }
    }
    for (int k = 0; k <= right - left; k++)
        *(p + left + k) = *(temp + k);
    delete temp;
}
```

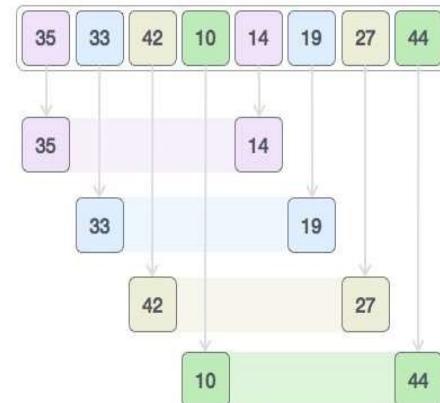


SHELL SORT

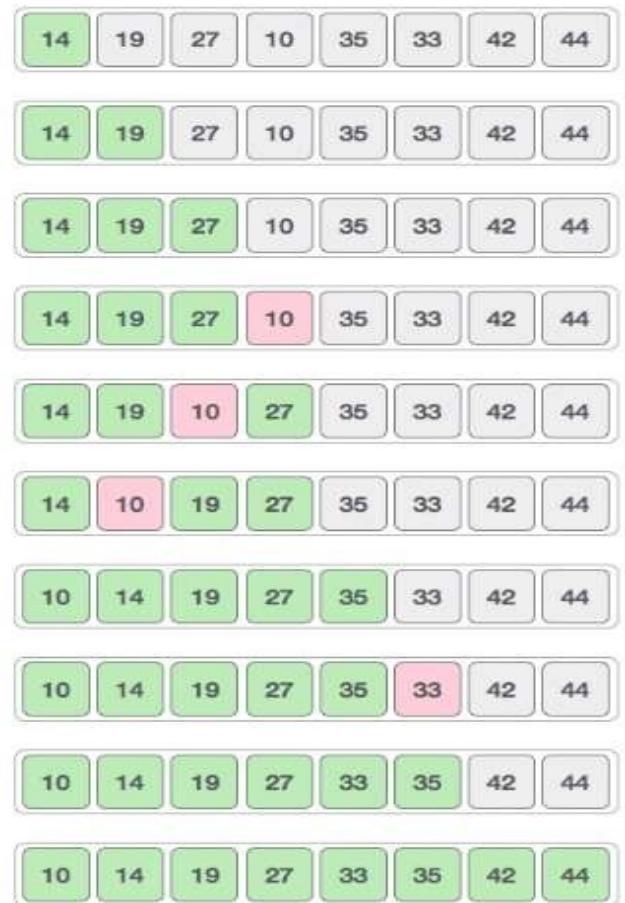
- Ý tưởng:
- Sử dụng Selection Sort trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn;
- Khoảng cách này là interval: là số vị trí từ phần tử này đến phần tử khác.
- $h = h * 3 + 1$ (h là interval với giá trị ban đầu là 1).

SHELL SORT

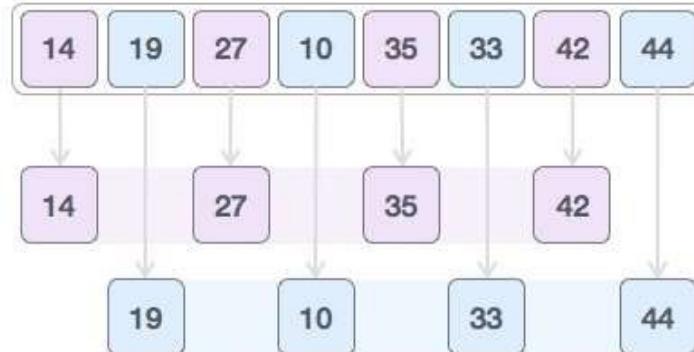
■ $h = 4$



• $h = 1$



■ $h = 2$



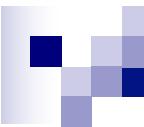
Tiếp tục so sánh và tráo đổi các giá trị (nếu cần) trong mảng ban đầu. Sau bước này, mảng sẽ
trống như sau:



SHELL SORT

■ Ví dụ:

```
void ShellSort(int *p, int length)
{
    for (int gap = length / 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < length; i += 1)
        {
            int temp = *(p + i);
            int j;
            for (j = i; j >= gap && *(p + j - gap) > temp; j -= gap)
                *(p + j) = *(p + j - gap);
            *(p + j) = temp;
        }
    }
}
```



QUICK SORT

- Sắp xếp nhanh;

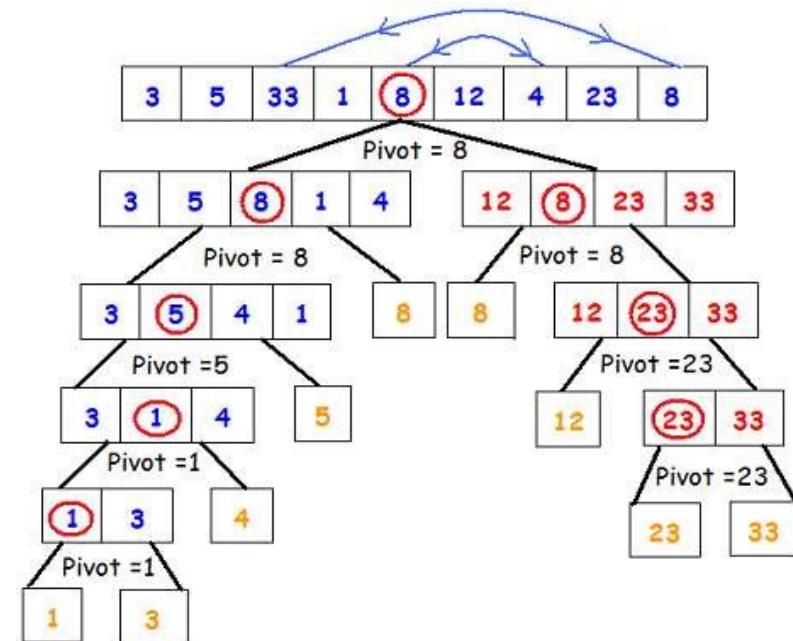
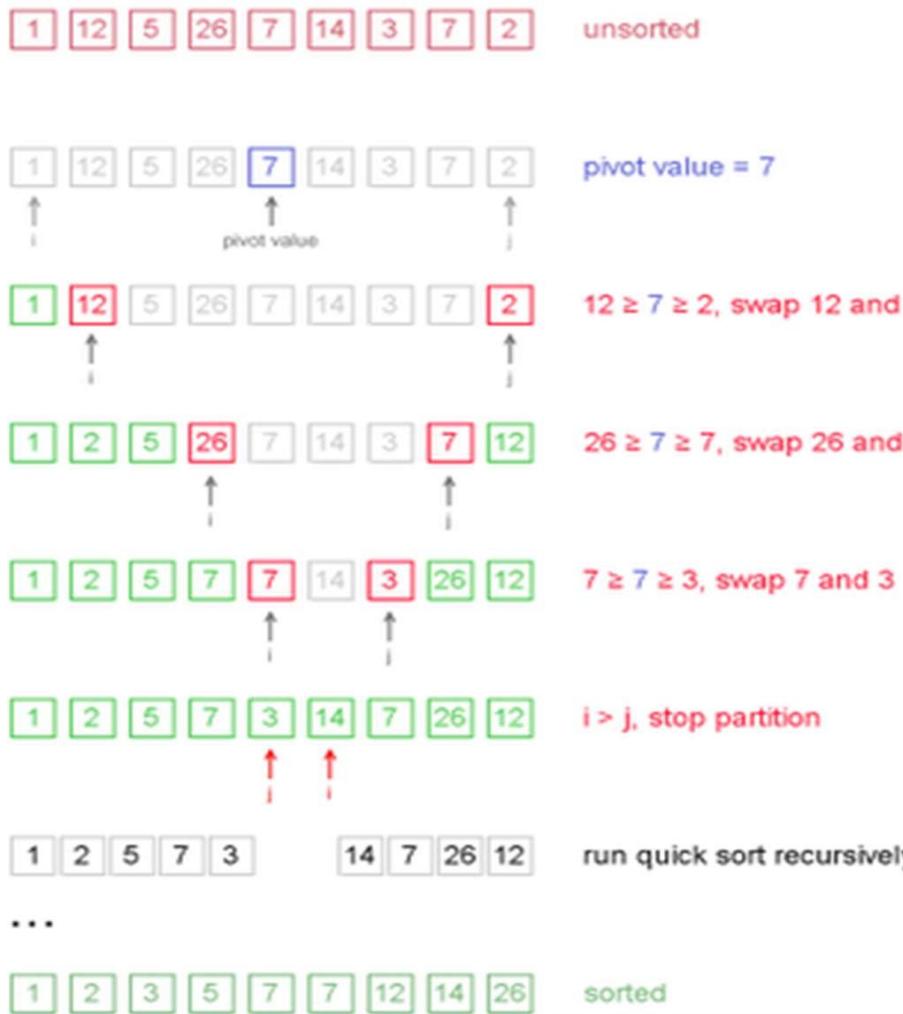
- Ý tưởng:

- QuickSort chia mảng thành hai danh sách bằng cách so sánh từng phần tử của danh sách với một phần tử được chọn được gọi là phần tử chốt. Những phần tử nhỏ hơn hoặc bằng phần tử chốt được đưa về phía trước và nằm trong danh sách con thứ nhất, các phần tử lớn hơn chốt được đưa về phía sau và thuộc danh sách con thứ hai. Tiếp tục chia như vậy tới khi các danh sách con đều có độ dài bằng 1.

- Cách chọn phần tử chốt:

- Chọn phần tử đứng đầu hoặc đứng cuối;
 - Chọn phần tử đứng giữa mảng;
 - Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối;
 - Chọn phần tử ngẫu nhiên.

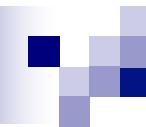
QUICK SORT



QUICK SORT

■ Ví dụ:

```
void QuickSort(int *p, int left, int right)
{
    srand(time(NULL));
    int key = *(p + (left + rand() % (right - left + 1)));
    int i = left, j = right;
    while (i <= j)
    {
        while (*(p + i) < key) i++;
        while (*(p + j) > key) j--;
        if (i <= j)
        {
            if (i < j)
                Swap(*(p + i), *(p + j));
            i++; j--;
        }
    }
    if (left < j)
        QuickSort(p, left, j);
    if (i < right)
        QuickSort(p, i, right);
}
```



HEAP SORT

- Sắp xếp vun đống;
- Ý tưởng:
- Phiên bản cải tiến của Selection Sort khi chia các phần tử thành 2 mảng con, 1 mảng các phần tử đã được sắp xếp và mảng còn lại các phần tử chưa được sắp xếp. Trong mảng chưa được sắp xếp, các phần tử lớn nhất sẽ được tách ra và đưa vào mảng đã được sắp xếp. Điều cải tiến ở Heapsort so với Selection sort ở việc sử dụng cấu trúc dữ liệu heap thay vì tìm kiếm tuyến tính (linear-time search) như Selection sort để tìm ra phần tử lớn nhất.

HEAP SORT

■ Ví dụ:

```
void Heapify(int *p, int length, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < length && *(p + left) > *(p + largest))
        largest = left;
    if (right < length && *(p + right) > *(p + largest))
        largest = right;
    if (largest != i)
    {
        Swap(*(p + i), *(p + largest));
        Heapify(p, length, largest);
    }
}
void HeapSort(int *p, int length)
{
    for (int i = length / 2 - 1; i >= 0; i--)
        Heapify(p, length, i);
    for (int i = length - 1; i >= 0; i--)
    {
        Swap(*p, *(p + i));
        Heapify(p, i, 0);
    }
}
```