

## Algorithm (演算法):

In many lessons, what you have learnt are just new syntax and statements.  
This time, you will learn how to solve a problem or perform some tasks.  
That's how to write a program.  
Algorithm is a term for the ways to do some specific tasks like sorting, searching.

This lesson, we will focus on **sorting** (排序) first.  
The task is to sort an array in either ascending or descending order.

First of all, there are mainly 2 kinds of sorting algorithms, **comparison** and **distribution**.

Namely, comparative sorting based on comparison among elements in the array while distribution sorting based on distribution of elements in the array.

Also, sorting can be divided into 2 groups in terms of their performance to handle duplicate keys.

**Stable (穩定的): It always preserves the original order of duplicate keys.**

**Unstable (不穩的): Otherwise, it is called unstable.**

Example: 0, 2, 3, 1<sub>a</sub>, 1<sub>b</sub> (there are two 1's, I use subscript a and b to denote them)

For stable sort: the order of 1<sub>a</sub> and 1<sub>b</sub> is preserved → 0, 1<sub>a</sub>, 1<sub>b</sub>, 2, 3

For unstable sort: the order may not be preserved → 0, 1<sub>b</sub>, 1<sub>a</sub>, 2, 3

We will discuss about sorting based on comparison first.

*In the following, we define array A that we are going to sort.*

*N is the total number of elements in the array.*

*J, K, temp are the control variables of for-loop etc.*

*For fear of the similarity between I and I, no identifier will be named i.*

*Swap(a,b) is a procedure to swap the value of a and b.*

```
Const n=5;
```

```
Var a:array[1..n] of integer;
```

```
Max,d,j,k,temp:integer;
```

```
Procedure swap(var x,y:integer);
```

```
Var temp:integer;
```

```
Begin
```

```
temp:=x;
```

```
x:=y;
```

```
y:=temp;
```

```
End;
```

**Comparison Sorting I:**

The first sorting algorithm is **Bubble Sort (氣泡排序法)**.

Pascal code of **Bubble Sort**

To sort in ascending order:

```
For j:=1 to n - 1 do
For k:=1 to n - k do
If a[k]>a[k+1] then swap(a[k],a[k+1]);
```

To sort in descending order:

```
For j:=1 to n - 1 do
For k:=1 to n - k do
If a[k]<a[k+1] then swap(a[k],a[k+1]);
```

Algorithm:

- (1) For each pair of adjacent elements, starting from first two to last two, compare adjacent elements.  
If the first is greater than the second, swap them.  
At last, the largest element must bubble up to the end. (n)
- (2) Repeat step(1) for all elements except the last one.  
At last, the second largest element will appear at last but one. (n-1)
- (3) Keep repeating for one fewer element each time, until you have no more pairs to compare.  
(Alternatively, keep repeating until no swaps are needed.)

Example: Sort 5 numbers with **bubble sort**. 99, 546, 126, -32, 12.

When j = 1 (for 5 numbers, there is only 4 adjacent pairs)

K=1	99 546 126 32 12	← No Swap is need!.
K=2	99 546 126 32 12	← Swap! Become 99 126 546 -32 12
K=3	99 126 546 32 12	← Swap! Become 99 126 32 546 12
K=4	99 126 32 546 12	← Swap! Become 99 126 32 12 546

*The largest element 546 is bubbled up to the end.*

*Therefore, we need to compare the remaining 4 numbers only.*

When j = 2 (for 4 numbers, there is only 3 adjacent pairs)

K=1	99 126 32 12 546	← No Swap is need!.
K=2	99 126 32 12 546	← Swap! Become 99 32 126 12 546
K=3	99 32 126 12 546	← Swap! Become 99 32 12 126 546

*The second largest element 126 is bubbled up to the last but one.*

*Therefore, we need to compare the remaining 3 numbers only.*

When j = 3 (for 3 numbers, there is only 2 adjacent pairs)

K=1	99 32 12 126 546	← Swap! Become 32 99 12 123 546
K=2	32 99 12 123 546	← Swap! Become 32 12 99 123 546

When j = 4 (for 2 numbers, it is the last pair to be compared)

K=1	32 12 99 123 546	← Swap! Become 12 32 99 123 546
-----	------------------	---------------------------------

*Obviously, it is a **stable sort**, as two duplicate keys won't be swapped.*

*Also number of comparisons is approximately equal to  $n^2$  for large n.*

*We denote it with the notation Big-O,  $O(n^2)$ . It stands for **time complexity**.*

*That means the time requires to sort the array is nearly  $n^2$ .*

### Comparison Sorting II:

The second one is called **Selection Sort** (選擇排序法)

```
For j:=1 to n – 1 do
Begin
min:=j;
For k:=j to n do
If a[k]<a[min] then min:=k;
Swap(a[j],a[min]);
End;

{This code is for ascending order}
{Change the < to >, for descending}
```

Find the min. element within the array, and swap it with the 1<sup>st</sup> element.  
(or put it to the first position)

Then, find the second min. element, and swap it with the 2<sup>nd</sup> element.  
(or put it to the second position)

....

Find the i<sup>th</sup> min. element, and swap it with the i<sup>th</sup> element.

Finally, look for the (n-1)<sup>th</sup> min. element and swap it with the (n-1)<sup>th</sup> element.

Example: Sort 5 numbers with **selection sort**. 99,126, -32, 12, 546.

When j = 1 (find the largest number)  
The largest number is 546.

Swap it with the first element.  
Become: 546, 126, -32, 12, 99.

When j =2, find the second largest number. (126)

Swap it with 2<sup>nd</sup> element.  
Become 546 126 -32 12 99

When j = 3, find the 3<sup>rd</sup> max. ( 99 )  
Become: 546 126 99 12 -32

When j = 4, find the 4<sup>th</sup> max (12)  
Become: 546 126 99 12 -32.

Finally, as the first (n-1) numbers are sorted, the last one must be the smallest.  
We finished sorting them.

*Obviously, it is an **unstable sort**, as we swap nonadjacent elements.*

*For each searching of k<sup>th</sup> max, we take k steps.*

*We have to searching for n-1 times.*

*That is  $1+2+3+\dots+n-1 = \frac{1}{2}n(n-1)$ , which is approximately equal to  $n^2$  for large n.*

*This is liked **Bubble Sort**, which compares  $\frac{1}{2}n(n-1)$  times.*

*So, it is also an **O(n<sup>2</sup>)** sorting.*

**Comparison Sorting III:**

The third one is called **Insertion Sort (插入排序法)**, which is the fastest sorting among three.

<pre> For k:=2 to n do Begin temp:=a[k]; j:=k-1; while (j&gt;=1) and (temp&lt;a[j]) do begin a[j+1]:=a[j]; dec(j); end; a[j+1]:=temp; End;                 </pre>	<p>Insertion sort aimed at keeping the first <math>k^{\text{th}}</math> elements are sorted, and insert the <math>(k+1)^{\text{th}}</math> element at suitable position such that the first <math>(k+1)^{\text{th}}</math> elements are also sorted.</p> <p>First, draw a[2], compare a[2] and a[1], if <math>a[2]&lt;a[1]</math> then move a[1] to a[2] and a[2] to a[1].</p> <p>...</p> <p>Finally, pick up the last element, and repeat to move the previous element until a suitable position is found.</p>
---	---

Example: Sort 5 numbers with **selection sort**. 126, 99, -32, 12, 546.

When  $k = 2$ . (2<sup>nd</sup> = **99**)

J=1	<u>126</u> 99 -32 12 546	← Move! Become 126 126 -32 12 546
-----	--------------------------	-----------------------------------

Insert 99 at the first position. Become : 99 126 -32 12 546

When  $k = 3$ . (3<sup>rd</sup> = **-32**)

J=2	99 <u>126</u> -32 12 546	← Move! Become 99 126 126 12 546
-----	--------------------------	----------------------------------

J=1	99 <u>126</u> 126 12 546	← Move! Become 99 99 126 12 546
-----	--------------------------	---------------------------------

Insert -32 at the first position. Become : -32 99 126 12 546

When  $k = 4$ . (**12**)

J=3	-32 99 <u>126</u> 12 546	← Move! Become -32 99 126 126 546
-----	--------------------------	-----------------------------------

J=2	-32 <u>99</u> 126 126 546	← Move! Become -32 99 99 126 546
-----	---------------------------	----------------------------------

J=1	<u>-32</u> 99 99 126 546	← No need to move!
-----	--------------------------	--------------------

Insert 12 at the second position. Become : -32 12 99 126 546

When  $k = 5$ . (**546**)

J=4	-32 12 99 <u>126</u> 546	← No need to move!
-----	--------------------------	--------------------

Finish.

*This is a **stable sort**.*

*Although it is also an  $O(n^2)$  sorting, it is the fastest compared with **Bubble Sort** and **Selection Sort**.*

*It is very efficient when the array is almost sorted.*

*It is about  $O(kn)$ , where  $k$  is number of unsorted elements.*

### Distribution Sort I:

In this section, I will use two more arrays *c* having the same data type as *a*,

**Const** *m* = 1000; {*m* is the range of number}

**var** *b*:array [1..*m*] of integer;

The simplest one is **Pigeonhole Sort**.

```
For k:=1 to m do b[k]:=0;
For k:=1 to n do inc(b[a[k]]);
J:=0;
For k:=1 to m do
While b[k]<>0 do
Begin
Inc(j);
A[j]:=k;
Dec(b[k]);
End;
```

Count the occurrences of each numbers and store to *b*.  
Then, starting from 1 to *m* put each occurred number to the array.

This is a **stable sort**.

It runs in **linear time**  $O(n+m)$ . But it also use  $O(n+m)$  memory.

Notes: if  $m \sim n$ , then it becomes  $O(2n) = O(n)$

If  $m \gg n$ , it becomes  $O(m)$ .

If  $n \gg m$ , it becomes  $O(n)$ .

The second one is called **Counting Sort**. It is modified from Pigeonhole Sort.

```
For k:=1 to m do b[k]:=0;
For k:=1 to n do inc(b[a[k]]);
For k:=2 to m do inc(b[k],b[k-1]);
For k:=n downto 1 do
Begin
C[b[a[k]]]:=a[k];
Dec(B[a[k]]);
End;
A:=C;
```

Count the occurrences of each numbers and store to *b*.  
Calculated the accumulated sum of  $b[1] \dots b[k]$  and store to  $b[k]$ .  
The value of  $b[k]$  indirectly represent at where the last value *k* shall be inserted.  
Therefore, from *n* to 1, for each element  $a[k]$  we insert them to array *C* at position  $b[a[k]]$  and decrease  $b[a[k]]$  by one.  
Finally, store *C* to *A*.  
This is also **stable**.  
It runs in **linear time**  $O(n)$  as well. But it uses more memory,  $O(n+m)$ .  
Generally, it is faster than **pigeonhole sort**.

The final one to be taught is called **Radix Sort**. It is a modification of **Counting Sort**.

This time, we declare array *b* with the following: **var** *b* : array [-9..9] of integer;

**function** *dig*(*m*,*n*:integer):integer;

**var** *j*:integer;

**begin**

**for** *j*:=1 to *n* do *m*:=*m* div 10; *digit*:=*m* mod 10;

**end**;

```
Max:=0;
For k:=1 to n do
if max<abs(a[k]) then max:=abs(a[k]);
for j:=0 to trunc(ln(max)/ln(10)) do
begin
For k:=-9 to 9 do b[k]:=0;
For k:=1 to n do inc(b[dig(a[k],j)]);
For k:=-8 to 9 do inc(b[k],b[k-1]);
For k:=n downto 1 do
Begin
C[b[dig(a[k],j)]]:=a[k];
Dec(B[dig(a[k],j)]);
End;
A:=C;
End;
```

First sort all numbers according to unit digit.  
Then to tenth digit.  
And to hundredth digit.  
And so on.

It is a **stable sort**.

Its time complexity is  $O(dn)$ .

*D* is the maximum number of digits.

It uses only  $O(n)$  memory.