**Insertion sort**

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

The numbers, which are needed to be sorted, are known as **keys**. Here is the algorithm of the insertion sort method.

**Algorithm: Insertion-Sort(A)**

for j = 2 to A.length

key = A[j]

i = j – 1

while i > 0 and A[i] > key

A[i + 1] = A[i]

i = i -1

A[i + 1] = key

Analysis

Run time of this algorithm is very much dependent on the given input.

If the given numbers are sorted, this algorithm runs in ***O(n)*** time. If the given numbers are in reverse order, the algorithm runs in ***O(n2)*** time.

Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Unsorted list:** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 13 | 5 | 18 | 14 | |

**1st iteration:**

Key = a[2] = 13

a[1] = 2 < 13

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Swap, no swap | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 13 | 5 | 18 | 14 | |

**2nd iteration:**

Key = a[3] = 5

a[2] = 13 > 5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Swap 5 and 13 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 5 | 13 | 18 | 14 | |

Next, a[1] = 2 < 13

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Swap, no swap | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 5 | 13 | 18 | 14 | |

**3rd iteration:**

Key = a[4] = 18

a[3] = 13 < 18,

a[2] = 5 < 18,

a[1] = 2 < 18

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Swap, no swap | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 5 | 13 | 18 | 14 | |

**4th iteration:**

Key = a[5] = 14

a[4] = 18 > 14

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Swap 18 and 14 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 5 | 13 | 14 | 18 | |

Next, a[3] = 13 < 14,

a[2] = 5 < 14,

a[1] = 2 < 14

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| So, no swap | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 5 | 13 | 14 | 18 | |

Finally,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **the sorted list is** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 2 | 5 | 13 | 14 | 18 | |

**SELECTION SORT**

This type of sorting is called **Selection Sort** as it works by repeatedly sorting elements. It works as follows: first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

**Algorithm: Selection-Sort (A)**

fori ← 1 to n-1 do

min j ← i;

min x ← A[i]

for j ←i + 1 to n do

if A[j] < min x then

min j ← j

min x ← A[j]

A[min j] ← A [i]

A[i] ← min x

Selection sort is among the simplest of sorting techniques and it works very well for small files. It has a quite important application as each item is actually moved at the most once.

Section sort is a method of choice for sorting files with very large objects (records) and small keys. The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.

Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test if ***A[j]*** < ***min x*** is executed exactly the same number of times in every case.

Selection sort spends most of its time trying to find the minimum element in the unsorted part of the array. It clearly shows the similarity between Selection sort and Bubble sort.

* Bubble sort selects the maximum remaining elements at each stage, but wastes some effort imparting some order to an unsorted part of the array.
* Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

For each ***i*** from ***1*** to ***n - 1***, there is one exchange and ***n - i*** comparisons, so there is a total of ***n - 1*** exchanges and

***(n − 1) + (n − 2) + ...+ 2 + 1 = n(n − 1)/2*** comparisons.

These observations hold, no matter what the input data is.

In the worst case, this could be quadratic, but in the average case, this quantity is ***O(n log n)***. It implies that the **running time of Selection sort is quite insensitive to the input**.

## Implementation

Void Selection-Sort(int numbers[], int array\_size) {

int i, j;

int min, temp;

for (i = 0; I < array\_size-1; i++) {

min = i;

for (j = i+1; j < array\_size; j++)

if (numbers[j] < numbers[min])

min = j;

temp = numbers[i];

numbers[i] = numbers[min];

numbers[min] = temp;

}

}

## Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Unsorted list:** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 5 | 2 | 1 | 4 | 3 | |

### 1st iteration:

Smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1, smallest = 1

3 > 1, smallest = 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Swap 5 and 1 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1 | 2 | 5 | 4 | 3 | |

### 2nd iteration:

Smallest = 2

2 < 5, smallest = 2

2 < 4, smallest = 2

2 < 3, smallest = 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No Swap | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1 | 2 | 5 | 4 | 3 | |

### 3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Swap 5 and 3 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | |

### 4th iteration:

Smallest = 4

4 < 5, smallest = 4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No Swap | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | |

Finally,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **the sorted list is** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | |

**Bubble Sort**

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

This is the simplest technique among all sorting algorithms.

**Algorithm: Sequential-Bubble-Sort (A)**

fori← 1 to length [A] do

for j ← length [A] down-to i +1 do

if A[A] < A[j - 1] then

Exchange A[j] ↔ A[j-1]

## Implementation

voidbubbleSort(int numbers[], intarray\_size) {

inti, j, temp;

for (i = (array\_size - 1); i >= 0; i--)

for (j = 1; j <= i; j++)

if (numbers[j - 1] > numbers[j]) {

temp = numbers[j-1];

numbers[j - 1] = numbers[j];

numbers[j] = temp;

}

}

## Analysis

Here, the number of comparisons are

**1 + 2 + 3 +...+ (*n* - 1) = *n*(*n* - 1)/2 = O(*n*2)**

Clearly, the graph shows the ***n2*** nature of the bubble sort.

In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

## Memory Requirement

From the algorithm stated above, it is clear that bubble sort does not require extra memory.

## Example

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Unsorted list:** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 5 | 2 | 1 | 4 | 3 | 7 | 6 | |

### 1st iteration:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **5 > 2 swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 2 | 5 | 1 | 4 | 3 | 7 | 6 | |
| **5 > 1 swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 2 | 1 | 5 | 4 | 3 | 7 | 6 | |
| **5 > 4 swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 2 | 1 | 4 | 5 | 3 | 7 | 6 | |
| **5 > 3 swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 2 | 1 | 4 | 3 | 5 | 7 | 6 | |
| **5 < 7 no swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 2 | 1 | 4 | 3 | 5 | 7 | 6 | |
| **7 > 6 swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 2 | 1 | 4 | 3 | 5 | 6 | 7 | |

### 2nd iteration:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **2 > 1 swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 4 | 3 | 5 | 6 | 7 | |
| **2 < 4 no swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 4 | 3 | 5 | 6 | 7 | |
| **4 > 3 swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| **4 < 5 no swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| **5 < 6 no swap** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

There is no change in 3rd, 4th, 5th and 6th iteration.

Finally,

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **the sorted list is** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

**Shell sort**

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as −

### Knuth's Formula

h = h \* 3 + 1

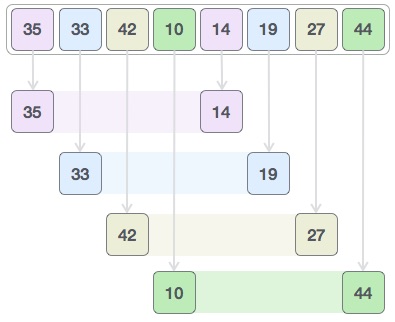
where −

h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is Ο(n), where n is the number of items. And the worst case space complexity is O(n).

## How Shell Sort Works?

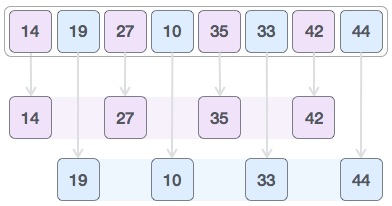
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −

Shell Sort

Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −

Shell Sort

Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −



We see that it required only four swaps to sort the rest of the array.

### Algorithm

Following is the algorithm for shell sort.

**Step 1** − Initialize the value of *h*

**Step 2** − Divide the list into smaller sub-list of equal interval *h*

**Step 3** − Sort these sub-lists using **insertion sort**

**Step 3** − Repeat until complete list is sorted

## Pseudocode

Following is the pseudocode for shell sort.

procedure shellSort()

A : array of items

/\* calculate interval\*/

while interval < A.length /3 do:

interval = interval \* 3 + 1

end while

while interval > 0 do:

for outer = interval; outer < A.length; outer ++ do:

/\* select value to be inserted \*/

valueToInsert = A[outer]

inner = outer;

/\*shift element towards right\*/

while inner > interval -1 && A[inner - interval] >= valueToInsert do:

A[inner] = A[inner - interval]

inner = inner - interval

end while

/\* insert the number at hole position \*/

A[inner] = valueToInsert

end for

/\* calculate interval\*/

interval = (interval -1) /3;

end while

end procedure