**TITLE**

SUBTITLE

Text

**ELEMENTARY SORTING ALGORITHMS**

**Bubble Sort**

is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array moves to the end in each iteration. Therefore, it is called a bubble sort.

**HOW IT WORKS**

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

**ALGORITHM**

void bubbleSort(int array[], int size) {

// loop to access each array element

for (int step = 0; step < size - 1; ++step) {

// loop to compare array elements

for (int i = 0; i < size - step - 1; ++i) {

// compare two adjacent elements

// change > to < to sort in descending order

if (array[i] > array[i + 1]) {

// swapping occurs if elements

// are not in the intended order

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

}

}

}

}

**OPTIMIZED ALGORITHM**

// Optimized Bubble sort in C

#include

// perform the bubble sort

void bubbleSort(int array[], int size) {

// loop to access each array element

for (int step = 0; step < size - 1; ++step) {

// check if swapping occurs

int swapped = 0;

// loop to compare array elements

for (int i = 0; i < size - step - 1; ++i) {

// compare two array elements

// change > to < to sort in descending order

if (array[i] > array[i + 1]) {

// swapping occurs if elements

// are not in the intended order

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

swapped = 1;

}

}

// no swapping means the array is already sorted

// so no need for further comparison

if (swapped == 0) {

break;

}

}

}

**TIME COMPLEXITY**

|  |  |
| --- | --- |
| **Worst Case Complexity** | O(n2) |
| If we want to sort in ascending order and the array is in descending order then the worst case occurs. |
| **Best Case Complexity** | O(n) |
| If the array is already sorted, then there is no need for sorting. |
| **Average Case Complexity** | O(n2) |
| It occurs when the elements of the array are in jumbled order (neither ascending nor descending). |

**SPACE COMPLEXITY**

Space complexity is **O(1)** because an extra variable is used for swapping. In the optimized bubble sort algorithm, two extra variables are used. Hence, the space complexity will be **O(2**).

**CONDITIONS**

* complexity does not matter
* short and simple code is preferred

**Selection Sort**

is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

This algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

**The subarray which is already sorted.**

**Remaining subarray which is unsorted.**

**HOW IT WORKS**

1. Set the first element as minimum.
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

1. After each iteration, minimum is placed in the front of the unsorted list.
2. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

**ALGORITHM**

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void selectionSort(int array[], int size) {

for (int step = 0; step < size - 1; step++) {

int min\_idx = step;

for (int i = step + 1; i < size; i++) {

// To sort in descending order, change > to < in this line.

// Select the minimum element in each loop.

if (array[i] < array[min\_idx])

min\_idx = i;

}

// put min at the correct position

swap(&array[min\_idx], &array[step]);

}

}

**STABLE ALGORITHM**

**void** stableSelectionSort(**int** a[], **int** n)

{

    // Iterate through array elements

**for** (**int** i = 0; i < n - 1; i++)

    {

        // Loop invariant : Elements till a[i - 1]

        // are already sorted.

        // Find minimum element from

        // arr[i] to arr[n - 1].

**int** min = i;

**for** (**int** j = i + 1; j < n; j++)

**if** (a[min] > a[j])

                min = j;

        // Move minimum element at current i.

**int** key = a[min];

**while** (min > i)

        {

            a[min] = a[min - 1];

            min--;

        }

        a[i] = key;

    }

}

**TIME COMPLEXITY**

|  |  |
| --- | --- |
| **Worst Case Complexity** | O(n2) |
| If we want to sort in ascending order and the array is in descending order then the worst case occurs. |
| **Best Case Complexity** | O(n2) |
| If the array is already sorted, then there is no need for sorting. |
| **Average Case Complexity** | O(n2) |
| It occurs when the elements of the array are in jumbled order (neither ascending nor descending). |

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.

**SPACE COMPLEXITY**

Space complexity is **O(1)** because an extra variable is used for swapping.

**CONDITIONS FOR USE**

* a small list is to be sorted
* cost of swapping does not matter
* checking of all the elements is compulsory
* cost of writing to memory matters like in flash memory (number of writes/swaps is O(n) as compared to O(n2) of bubble sort)

**Insertion Sort**

is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

**HOW IT WORKS**

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

1. Now, the first two elements are sorted.

Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

1. Similarly, place every unsorted element at its correct position.

**ALGORITHM**

void insertionSort(int array[], int size) {

for (int step = 1; step < size; step++) {

int key = array[step];

int j = step - 1;

// Compare key with each element on the left of it until an element smaller than

// it is found.

// For descending order, change key<array[j] to key>array[j].

while (key < array[j] && j >= 0) {

array[j + 1] = array[j];

--j;

}

array[j + 1] = key;

}

}

**TIME COMPLEXITY**

|  |  |
| --- | --- |
| **Worst Case Complexity** | O(n2) |
| Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.  Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.  Thus, the total number of comparisons = n\*(n-1) ~ n2 |
| **Best Case Complexity** | O(n) |
| When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear. |
| **Average Case Complexity** | O(n2) |
| It occurs when the elements of the array are in jumbled order (neither ascending nor descending). |

**SPACE COMPLEXITY**

Space complexity is **O(1)** because an extra variable is used for swapping.

**CONDITIONS FOR USE**

* the array is having a small number of elements
* there are only a few elements left to be sorted