Data Structures and Algorithms

In C++

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
February 16, 2023
United States

Contents

Selection Sort

Concept 1: The selection sort algorithm sorts an array by repeatedly finding the minimum element (if sorting in ascending order) from the unsorted part of the array and putting it at the end of the sorted part of the array. The algorithm maintains two subarrays in a given array:

- A subarray of already sorted elements.
- A subarray of elements that remain to be sorted.

At the start of the algorithm, the first subarray is empty. In each pass through the outer loop of the selection sort, the minimum element from the unsorted subarray is selected and moved to the end of the sorted subarray.

1.1 Psuedocode

```
procedure selection_sort(array : list of sortable items, n :
↪  length of list)
    i := 0
    while i < n - 1
        min_index ← i
        j := i + 1
        while j < n
            if array[j] < array[min_index]
                min_index ← j
            end if
            j = j + 1
        end while
        swap array[i] and array[min_index]
        i = i + 1
    end while
end procedure
```

1.2 Example

```
int main(int argc, const char* argv[]) {
    int arr[] = {2,4,1,3,5}; int n = 5;

    for (int j=0; j <n-1; ++j) {
        int min = j;
        for (int k=j+1; k <n-1; ++k) {
            if (arr[k] < arr[min]) {
                min = k;
            }
        }
        std::swap(arr[j], arr[min]);
    }
}
```

1.3 Complexity

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$

Note:-

The primary advantage of selection sort is that it never makes more than $O(n)$ swaps, which can be useful if the array elements are large and copying them is a costly operation.

Insertion Sort

Concept 2: The insertion sort algorithm sorts a list by repeatedly inserting an unsorted element into the correct position in a sorted sublist. The algorithm maintains two sublists in a given array:

- A sorted sublist. This sublist initially contains a single element (an array of one element is always sorted).
- A sublist of elements to be inserted one at a time into the sorted sublist.

2.1 Psuedocode

```
procedure insertion_sort(array : list of sortable items, n :
↪  length of list)
    i ← 1
    while i < n
        j ← i
        while j > 0 and array[j - 1] > array[j]
            swap array[j - 1] and array[j]
            j ← j - 1
        end while
        i ← i + 1
    end while
end procedure
```

2.2 Example

```
int main(int argc, const char* argv[]) {
    int arr[] = {2,4,1,3,5};
    int n = 5;

    for (int j=1; j<n; ++j) {
        for (int k=j; k>0; --k) {
            if (arr[k-1] > arr[k]) {
                std::swap(arr[k-1], arr[k]);
            }
        }
    }
}
```

2.3 Optimizing Insertion Sort

Performing a full swap of the array elements in each inner for loop iteration is not necessary. Instead, we save the value that we want to insert into the sorted subarray in temporary storage. In place of performing a full swap, we simply copy elements to the right. The saved value can then be inserted into its proper position once that has been located.

This alternative approach can potentially save a considerable number of assignment statements. If N swaps are performed by the inner loop, the original version of insertion sort requires $N \cdot 3$ assignment statements to perform those swaps. The improved version listed below only requires $N + 2$ assignment statements to accomplish the same task.

2.3.1 Psuedocode

```
procedure insertion_sort(array : list of sortable items, n :
↪  length of list)
    i ← 1
    while i < n
        temp ← array[i]
        j ← i
        while j > 0 and array[j - 1] > temp
            array[j] ← array[j - 1]
            j ← j - 1
        end while
        array[j] ← temp
        i ← i + 1
    end while
end procedure
```

2.3.2 Example

```
int arr[] = {5,6,4,3,1};
int n = 5;

for (int j=1; j<n; ++j) {
    int tmp = arr[j];
    int k=j;
    for (; k>0; --k) {
        if (arr[k-1] > tmp) {
            arr[k] = arr[k-1];
        } else {
            break;
        }
    }
    arr[k] = tmp;
}
```

2.4 Complexity

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$

Note:-

The primary advantage of insertion sort over selection sort is that selection sort must always scan all remaining unsorted elements to find the minimum element in the unsorted portion of the list, while insertion sort requires only a single comparison when the element to be inserted is greater than the last element of the sorted sublist. When this is frequently true (such as if the input list is already sorted or partially sorted), insertion sort is considerably more efficient than selection sort. The best case input is a list that is already correctly sorted. In this case, insertion sort has O(n) complexity.

Bubble Sort

Concept 3: Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

3.1 Psuedocode

```
procedure bubble_sort(array : list of sortable items, n : length
↪ of list)
    do
        swapped ← false
        i ← 1
        while i < n
            if array[i - 1] > array[i]
                swap array[i - 1] and array[i]
                swapped ← true
            end if
            i ← i + 1
        end while
    while swapped
end procedure
```

Note:-

If no items are swapped during a pass through the outer loop (i.e., the variable swapped remains false), then the array is already sorted and the algorithm can terminate.

3.2 Example

```
int arr[] = {5,6,4,3,1};
int n = 5;

bool swapped;
do {
    swapped = 0;

    for (int i=0; i<n; ++i) {
        if (arr[i-1] > arr[i]) {
            std::swap(arr[i-1], arr[i]);
            swapped = 1;
        }
    }

} while (swapped);

```

3.3 Optimizing Bubble Sort

The bubble sort algorithm can be optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. Therefore the inner loop can avoid looking at the last $n - 1$ items when running for the n-th time:

3.3.1 Psuedocode

```
procedure bubble_sort(array : list of sortable items, n : length
↪  of list)
    do
        swapped ← false
        i ← 1
        while i < n
            if array[i - 1] > array[i]
                swap array[i - 1] and array[i]
                swapped ← true
            end if
            i ← i + 1
        end while
        n ← n - 1
    while swapped
end procedure
```

It is common for multiple elements to be placed in their final positions on a single pass. In particular, after every pass through the outer loop, all elements after the position of the last swap are sorted and do not need to be checked again. Taking this into account makes it possible to skip over many elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the swapped variable:

```
 do
    last ← 0
    i ← 1
    while i < n
        if array[i - 1] > array[i]
            swap array[i - 1] and array[i]
            last ← i
        end if
        i ← i + 1
    end while
    n ← last
while n > 1
```

3.3.2 Example

```
int last;
do  {
    last = 0;
    int j=1;

    for (; j<n; ++j) {
        if (arr[j-1] > arr[j]) {
            std::swap(arr[j-1], arr[j]);
            last = j;
        }
    }
    n = last;

} while (n > 0);
```

3.4 Complexity

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$

Note:-

Other $O(n^2)$ sorting algorithms, such as insertion sort, generally run faster than bubble sort (even with optimizations) and are no more complex. Therefore, bubble sort is not a practical sorting algorithm. The only significant advantage that bubble sort has over most other sorting algorithms (but not insertion sort), is that the ability to detect that the list is sorted is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$.