Insertion, Selection, and Bubble Sort

Josh Wilcox (jw14g24@soton.ac.uk)

February 25, 2025

Contents

1 Properties of Sorting Aglorithm

1.1 Stability

A sorting algorithm is **stable** if it does not change the order of elements that have the same value

- For example, if you have a list of tuples where the first element is the key to sort by, a stable sort will maintain the relative order of tuples with equal keys.
- Stability is important in scenarios where the order of equal elements carries additional information that should be preserved.

1.2 In-Place

A sorting algorithm is **in-place** if the memory use is $\mathcal{O}(1)$

- This means that the algorithm sorts the list without requiring extra space proportional to the size of the input list.
- In-place algorithms are generally more space-efficient and can be crucial in environments with limited memory.

2 Insertion Sort

Algorithm 1 Insertion Sort

```
1:  procedure InsertionSort(A)
2:      for j ← 2 to length(a) do
3:          key ← a[j]
4:          i ← j − 1
5:          while u ≥ 0 and a[i] > key do
6:              a[i + 1] ← a[i]
7:              i ← i − 1
8:          end while
9:          a[i + 1] ← key
10:     end for
11: end procedure
```

2.1 Properties

2.1.1 Worst Case Comlexity

- Occurs when array is in **reverse order**
- Every element has to be moved to the front
- Number of comparisions:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- $\Theta(n^2)$

2.1.2 Average Case Complexity

- Each new element moves half way down the sorted subsequence
- Gives the average time complexity of **half** the worst case
- Therefore still $\Theta(n^2)$

2.1.3 Best case complexity

- Occurs when array is already sorted
- Only $n-1$ comparisons as you just move through the array
- $\Theta(n)$

2.2 Uses

- A good choice for sorting **small arrays**
  - Done as it is **stable** and **inplace**
  - Also is very fast when arrays are already almost sorted

3 Selection Sort

- More direct brute force method
- Searches the smallest element and puts it in the correct final position
  - Then continue with the second smallest etc
- Can be made **in-place** by only **swapping** elements
  - Removes the need for creating a new temp array

Algorithm 2 Selection Sort

```
1:  procedure SelectionSort(A)
2:      for j ← 1 to length(a) − 1 do
3:          min ← j
4:          for i ← j + 1 to length(a) do
5:              if a[i] < a[min] then
6:                  min ← i
7:              end if
8:          end for
9:          swap a[j] and a[min]
10:     end for
11:     return a                                   ▷ Sorted sequence
12: end procedure
```

3.1 Not Stable

- Selection sort is not stable because it swaps elements that are not adjacent.
- Consider the following example:
  - Suppose we have the array $[4_a, 3, 4_b, 1]$, where 4_a and 4_b are two different elements with the same value.
  - In the first pass, the smallest element 1 is found and swapped with 4_a, resulting in $[1, 3, 4_b, 4_a]$.
  - The relative order of 4_a and 4_b has changed, demonstrating that selection sort is not stable.

3.2 Time Complexity

Always Requires $\frac{n(n-1)}{2}$ comparisons therefore:

$$\Theta(n^2)$$

- Requires more comparisons than insertion sort
- However expected time can be more precisely calculated

- Selection sort also has **inferior** average case and best case complexity

3.2.1 Swaps

- Selection sort only performs at most $n - 1$ swaps
  - This is fewer than the Insertion Sort
  - Makes it attractive when physical swapping is expensive

4 Bubble Sort

- Requires lots and lots of comparisons and swapping
- Time complexity is $\mathcal{O}(n^2)$

Algorithm 3 Bubble Sort

```
1: procedure BubbleSort(A)
2:     for j ← length(a) to 2 do
3:         for i ← 1 to j − 1 do
4:             if a[i] > a[i + 1] then
5:                 swap a[i] and a[i + 1]          ▷ Exchange adjacent elements
6:             end if
7:         end for
8:     end for
9:     return a                                    ▷ Sorted sequence
10: end procedure
```