جامعة آل البيت
AL al-BAYT UNIVERSITY

Faculty of Prince Al-Hussien
Bin Abdullah II for IT

An Explanation of the Selection Sort Algorithm

Sulaiman J. Al-Asad
SID 2100901005

Table of Contents

1. Introduction

1.1. History

Ever since the beginning of Computer Science, *The Sorting Problem* has attracted a large amount of attention, with some of the earliest sorting algorithms dating back as far as 1887, when Herman Hollerith invented *Radix Sort* for his tabulating machine [1].

Another early example of a sorting algorithm is *Bubble Sort*, of which the earliest description was analysis was in a 1956 paper by Edward H. Friend [2].

1.2. Classification

Sorting algorithms can be classified into several different categories:

- Asymptotic runtime complexity:
  Best-case, Average-case, Worst-case performance, usually only interested in Best-case and Worst-case performances for a given algorithm.
- Memory usage (Space complexity)
  Some algorithms are *in-place* (see next point), while others require additional memory resources to complete their sorting.
- Technique
  There are *In-place* sorting algorithms, that work within the list itself, Bubble Sort for example. While some others use *Divide & Conquer* techniques like Merge Sort. Those are all *Comparison Based*, while there are algorithms like Radix Sort which work based on individual digits.

1.3. Stability

Stability is a Sorting Algorithm's ability to maintain the order of equal elements.

Stable sorting algorithms sort equal elements in the same order they appear in the input list. For example, presume you are sorting the following list of 5 elements: **[8, 5, 3_B, 1, 3_A]** (where **3_B** is equal to **3_A**), a stable sorting algorithm would sort it as **[1, 3_B, 3_A, 5, 8]**, notice the order of elements **3_A** & **3_B** is unchanged in the sorted list compared to the input, while an unstable algorithm may sort it as **[1, 3_A, 3_B, 5, 8]**, where the order of the equal elements was not maintained.

When equal elements are indistinguishable, like say, a number, stability is usually not an issue worth considering.

2. Basics of Selection Sort

2.1. What is Selection Sort?

Selection Sort is an in-place, comparison based, ***O(n²)*** time complexity sorting algorithm.

It works by dividing the input into two parts, a “sorted” part and an “unsorted” part, while repeatedly searching for the smallest element in the “unsorted” part and placing it at the end of the “sorted” part.[4]

2.2. Implementation in Code

Selection Sort is a very simple and straightforward algorithm to implement in any programming language, below is the pseudocode for the algorithm.

```
func SelectionSort(a[]):
   n = a.len;
   for i, 0 → n - 1:
      min = i;
      for j, i + 1 → n:
         if a[j] < arr[min]:
            min = j;
      swap(a[i], a[min]);
end func
```

2.3. Space & Time Complexity

Since Selection Sort is an In-place sorting algorithm, the space complexity stays constant at ***O(1)***, as there are no additional memory resources required to complete the sort.

However, Selection Sort’s time complexity grows very quickly with input size, as it has a quadratic time complexity ***O(n²)***, which we calculate in the following equation.

$$\frac{(n-1)+(n-2)+(n-3)+\cdots+1}{2} \cong \frac{n(n-1)}{2} = \frac{n^2+n}{2} = n^2$$

Each iteration is one step less than the previous, hence why we write a sequence of ***(n-1)* + *(n-2)* + *…* +** *1*, with the last being only a single step.

However, Selection Sort’s time complexity is the same across the board for all cases, with worst, average, and best-case performances all of ***O(n²)***, which means that no matter what data is input, the algorithm will take the same amount of time, which makes it inefficient when the input data set is already somewhat sorted.

One notable property of Selection Sort is the fact that it only performs a total of ***O(n)*** swaps in the worst-case, while it is possible to modify the algorithm to not swap if it is not necessary, bringing the number of swaps in the best case to ***O(1)***.

2.4. When to Use

As Selection Sort is a quadratic time-complexity algorithm, it should not be used when the data set is relatively large, as it will significantly increase in running time with a small increase

However, Selection Sort still has its use cases, most notably in embedded systems, where memory write speeds are very slow, and minimizing the number of swaps is of utmost importance.

It's also efficient at Sorting only a set number of elements out of a larger data set, for example finding the highest or lowest ***m*** values in the input set ***n***, which brings its time complexity to ***O(n . m)***.

One thing to note is that Selection Sort is not stable by default. However, a stable implementation is possible with relatively little code modification.

3. Bidirectional Selection Sort

3.1. What's Bidirectional?

Also known as double Selection Sort, double-ended Selection Sort, or even cocktail sort, due to its similarity to a cocktail shaker. This modification of the original Selection Sort aims to improve the performance by performing two swaps per iteration as opposed to one in traditional Selection Sort.

The sort begins from two elements and searches the entire list until it finds the minimum value and maximum value. The sort will swap the value of the first element with the minimum value and the last element with the maximum value. It will then select the second and the last element and searches for the second minimum and maximum element. The process will continue until the list is sorted.

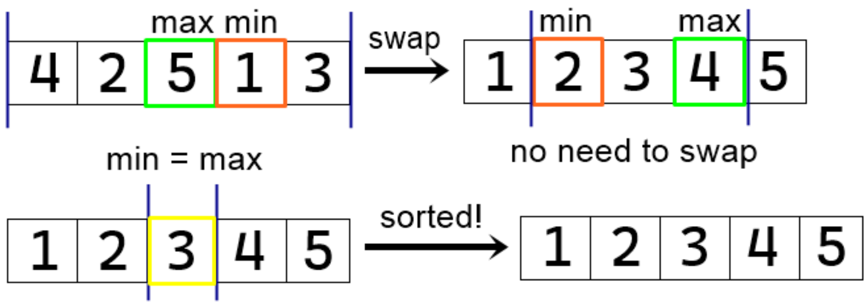


Figure 3.1.1 – How Bidirectional Selection Sort works

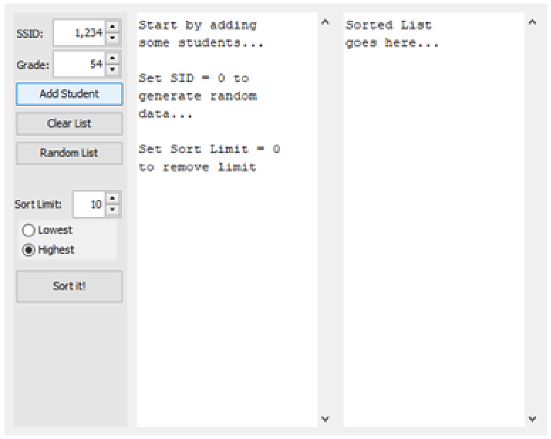
3.2. Why it’s better in practice

While the Time-Complexity of Bidirectional Selection sort stays the same at ***O(n²)***, its real-world performance is anywhere between *25-35%* more efficient than traditional Selection Sort. [5]

4. Selection Sort Interactive Example

In the provided interactive example, you have the ability to either sort the entire list of students by grades in the desired order or extract the desired number of highest or lowest grades in the list.

The application is demonstrated in the figures below.



SSID: 1,234
Grade: 54
Sort Limit: 10
[ ] Lowest [✓ Highest]

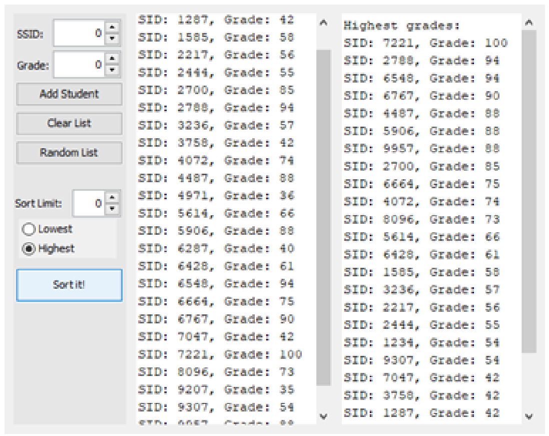
```
Start by adding
some students...

Set SID = 0 to
generate random
data...

Set Sort Limit = 0
to remove limit
```

```
Sorted List
goes here...
```

Fig. 4.1 - Adding some students

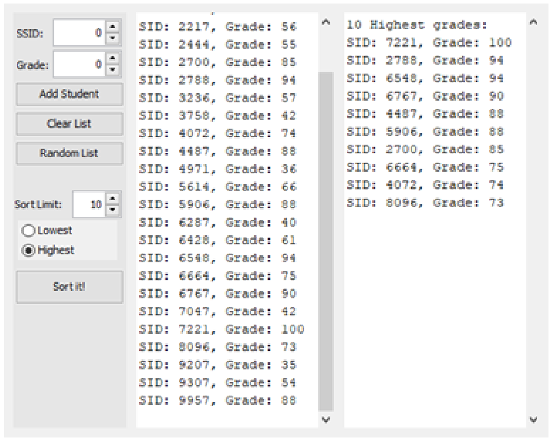


SSID: 0
Grade: 0
Sort Limit: 0
[ ] Lowest [✓ Highest]

```
SID: 1287, Grade: 42
SID: 1585, Grade: 58
SID: 2217, Grade: 56
SID: 2444, Grade: 55
SID: 2700, Grade: 85
SID: 2788, Grade: 94
SID: 3236, Grade: 57
SID: 3758, Grade: 42
SID: 4072, Grade: 74
SID: 4487, Grade: 88
SID: 4971, Grade: 36
SID: 5614, Grade: 66
SID: 5906, Grade: 88
SID: 6287, Grade: 40
SID: 6428, Grade: 61
SID: 6548, Grade: 94
SID: 6664, Grade: 75
SID: 6767, Grade: 90
SID: 7047, Grade: 42
SID: 7221, Grade: 100
SID: 8096, Grade: 73
SID: 9207, Grade: 35
SID: 9307, Grade: 54
```

```
Highest grades:
SID: 7221, Grade: 100
SID: 2788, Grade: 94
SID: 6548, Grade: 94
SID: 6767, Grade: 90
SID: 4487, Grade: 88
SID: 5906, Grade: 88
SID: 9957, Grade: 88
SID: 2700, Grade: 85
SID: 6664, Grade: 75
SID: 4072, Grade: 74
SID: 8096, Grade: 73
SID: 5614, Grade: 66
SID: 6428, Grade: 61
SID: 1585, Grade: 58
SID: 3236, Grade: 57
SID: 2217, Grade: 56
SID: 2444, Grade: 55
SID: 1234, Grade: 54
SID: 9307, Grade: 54
SID: 7047, Grade: 42
SID: 3758, Grade: 42
SID: 1287, Grade: 42
```

Fig. 4.2 - Sorting the entire list in descending order

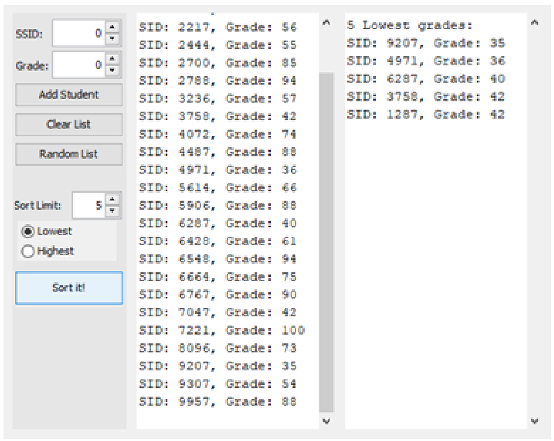


SSID: 0
Grade: 0
Sort Limit: 10
[ ] Lowest [✓ Highest]

```
SID: 2217, Grade: 56
SID: 2444, Grade: 55
SID: 2700, Grade: 85
SID: 2788, Grade: 94
SID: 3236, Grade: 57
SID: 3758, Grade: 42
SID: 4072, Grade: 74
SID: 4487, Grade: 88
SID: 4971, Grade: 36
SID: 5614, Grade: 66
SID: 5906, Grade: 88
SID: 6287, Grade: 40
SID: 6428, Grade: 61
SID: 6548, Grade: 94
SID: 6664, Grade: 75
SID: 6767, Grade: 90
SID: 7047, Grade: 42
SID: 7221, Grade: 100
SID: 8096, Grade: 73
SID: 9207, Grade: 35
SID: 9307, Grade: 54
SID: 9957, Grade: 88
```

```
10 Highest grades:
SID: 7221, Grade: 100
SID: 2788, Grade: 94
SID: 6548, Grade: 94
SID: 6767, Grade: 90
SID: 4487, Grade: 88
SID: 5906, Grade: 88
SID: 2700, Grade: 85
SID: 6664, Grade: 75
SID: 4072, Grade: 74
SID: 8096, Grade: 73
```

Fig. 4.3 - Sorting for the 10 highest grades in the list



SSID: 0
Grade: 0
Sort Limit: 5
[✓ Lowest] [ ] Highest

```
SID: 2217, Grade: 56
SID: 2444, Grade: 55
SID: 2700, Grade: 85
SID: 2788, Grade: 94
SID: 3236, Grade: 57
SID: 3758, Grade: 42
SID: 4072, Grade: 74
SID: 4487, Grade: 88
SID: 4971, Grade: 36
SID: 5614, Grade: 66
SID: 5906, Grade: 88
SID: 6287, Grade: 40
SID: 6428, Grade: 61
SID: 6548, Grade: 94
SID: 6664, Grade: 75
SID: 6767, Grade: 90
SID: 7047, Grade: 42
SID: 7221, Grade: 100
SID: 8096, Grade: 73
SID: 9207, Grade: 35
SID: 9307, Grade: 54
SID: 9957, Grade: 88
```

```
5 Lowest grades:
SID: 9207, Grade: 35
SID: 4971, Grade: 36
SID: 6287, Grade: 40
SID: 3758, Grade: 42
SID: 1287, Grade: 42
```

Fig. 4.4 - Sorting for the 5 lowest grades in the list

5. Sources & References

1. Patent, US395781.
2. Friend, E.; Sorting on electronic computer systems.
   J. ACM 3 (1956), 134–168.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest,
   Clifford Stein, **Introduction to Algorithms (3rd edition)**,
   Cambridge - Massachusetts, *The MIT Press (2009)*
   ISBN 978-0-262-03384-8.
4. Skiena, Steven S.; **The Algorithm Design Manual (2nd edition)**
   New York, NY State university, ISBN 978-1-84800-069-8. (p109)
5. Vilchez, Ramcis N.;
   Bidirectional Enhanced Selection Sort Algorithm Technique
   Technological Institute of the Philippines (TIP),
   Quezon City, Philippines (p29)