

# Inefficient Sorting Algorithms

Sorting - the process of ordering a bunch of items based on one or more rules, subject to one or more constraints...

Items - what are we sorting, and how many are there?

Rules - how do we order them?

Constraints - is the data in an array or a linked list? etc.

**Rule #1:** Don't choose a sorting algorithm until you understand the requirements of your problem.

**Rule #2:** Always choose the simplest sorting algorithm possible that meets your requirements.

## Selection Sort

1. Look at all N books and select the shortest book
2. Swap this with the first book
3. Look at the remaining N-1 books, and select the shortest
4. Swap this book with the second book
5. Look at the remaining N-2 books, and select the shortest
6. Swap this book with the third book, and so on...

### O(N^2)

Notes: if all the books are mostly in order before our sort starts, selection sort still takes just as many steps either way. Selection sort is **unstable**. It doesn't adapt to the data in any way, so its runtime is always quadratic. However, selection sort minimizes the number of swaps, so in applications where the cost of swapping items is high, selection sort may very well be the algorithm of choice.

```
void selectionSort(int A[], int n) //shelf of N books

{
    for(int i = 0; i < n; i++) //for i = 1 to N
    {
        int minIndex = i;

        for(int j = i + 1; j < n; j++) //find the smallest book between slots i and N
        {
            if(A[j] < A[minIndex])
                minIndex = j;
        }

        swap(A[i], A[minIndex]); //swap this smallest book with book i
    }
}
```

Side note: an "unstable" sorting algorithm re-orders the items without taking into account their initial ordering. A "stable" sorting algorithm does take into account the initial ordering when sorting, maintaining the order of similar-valued items.

## Insertion Sort

(the most common way to sort playing cards?)

1. Let's focus on the first two books - ignore the rest.
2. If the last book in this set is in the wrong order
3. If the last book in this set is in the wrong order, remove it from the shelf
  1. Shift the book before it to the right
  2. Insert our book into the proper slot
4. Now focus on the first three books.
5. If the last book in this set is in the wrong order, remove it from the shelf
  1. Shift the books before it to the right, as necessary
  2. Insert our book into the proper slot
6. Now our first three books are in sorted order!
7. Now focus on the first four books - ignore the rest.
8. If the last book is in this set is in the wrong order, remove it from the shelf
  1. Shift the books before it to the right, as necessary
  2. Insert our book into the proper slot
9. Just keep repeating this process until the entire shelf is sorted

### **O(N^2)**

Notes: if all the books are in the proper order, then insertion sort never needs to do any shifting! In this case, it just takes roughly  $\sim N$  steps to sort the array! **O(N)**

Insertion sort is **stable**. It's usually used when the data is nearly sorted (because insertion sort is an adaptive algorithm) or when the problem size is small. For these reasons, insertion sort is often used as the recursive base case for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

```
void insertionSort(int A[], int n)

{
    for(int s = 2; s <= n; s++) //focus on successively larger prefixes of the array

    {
        int sortMe = A[s - 1]; //make a copy of the last value in the current set

        int i = s - 2;

        while(i >= 0 && sortMe < A[i]) //shift the values in the focus region right until we find the proper slot for sortMe

        {
            A[i + 1] = A[i];

            i--;
        }

        A[i + 1] = sortMe; //store the sortMe value into the vacated slot
    }
}
```

## Bubble Sort (the stupidest f\*cking algorithm ever)

1. Start at the top element of your array
2. Compare the first two elements: A[0] and A[1]
3. If they're out of order, then swap them
4. Then advance one element in your array
5. Compare these two elements A[1] and A[2]
6. If they're out of order, swap them
7. Repeat this process until you hit the end of the array
8. When you hit the end, if you made at least one swap, then repeat the whole process again!!!

During each pass, we compare every element with its successor (and possibly swap each). That requires about N steps. If we did even one swap, we need to repeat the whole process again. What the worst case? We might have to repeat this entire process N times.

**O(N^2)**

Notes: Bubble sort is **stable**. Just like insertion sort, bubble sort is really efficient on pre-sorted arrays and linked lists.

```
void bubbleSort(int Arr[], int n)

{
    bool atLeastOneSwap;

    do
    {
        atLeastOneSwap = false; //start by assuming we won't do any swaps

        for(int j = 0; j < n - 1; j++)
        {
            if(Arr[j] > Arr[j + 1]) //compare each element with its neighbor and swap them if they're out-of-order
            {
                Swap(arr[j], Arr[j + 1]);

                atLeastOneSwap = true; //don't forget we swapped!
            }
        }

        } while(atLeastOneSwap == true); //if we swapped at least once, then start back at the top and repeat the whole process
    }
```