Quadratic Sorting Algorithms - O(n^2)

Say we have an array of values, put these values in ascending or descending order.

A few variations:

1) These could be objects, not numbers. In our examples, we'll make them numbers, put in practical applications they are usually objects
2) For example, events, sort by timestamp

IN PLACE SORT - just change the array you are in
OUT OF PLACE SORT - allocate a new array of the same size, and put your sorted version here.
(We're going to be studying in-place today)

Stable Sort - things that are equal stay in the same order

- If you're dealing with integers - who cares, a 9 is a 9
- But if you're dealing with events that you're ordering with timestamps, maybe if things occurred at the same time you want them in the original order of the array
- Most of our examples happen to be stable (or at least have a stable version)

In order to sort, you need to have a comparison function:
f(a, b) to check if a < b.
If they are integers or doubles, your machine comes with these “<”, “>”
If they are objects, Java has a Comparable interface:

- If a, b are comparable, you can call a.compareTo(b) and get an answer
- Usually it's < 0 if a is smaller, 0 if they are equal > 0 if b is smaller
- So, essentially a - b

Notes on comparison functions. Obvious? Perhaps, but let’s put on our completionist mathematician hat here.

- Needs to be deterministic
    - You shouldn’t get different answers every time you call a.compareTo(b) on the same objects!
- Needs to follow the transitive law: a<b and b < c implies a < c
- Needs to be a total order (either a < b, a = b, or a > b)
    - Partial orders (where some objects don’t compare to each other) lead to a different type of sort, out of scope.

One of the most studied problems in computer science, from the advent of computers. Why?

- Back to UNIVAC, ENIAC
- They discovered that they needed to run sorts a lot; very useful
- There are many different sorting algorithms, still being invented. There’s no one-size-fits all optimum, and it often depends on the situation
- Interesting theoretical properties.

Let’s give away the ending:

- Today we are going to go over quadratic sorts O(n^2). These are simplest sorting algorithms that work well on small sets of data
- End of semester: O(n * log n) algorithms, best you can do in most circumstances, and in special circumstances, where you get close to linear time.

We need another piece, a SWAP FUNCTION

```
swap(arr, i, j):
  temp = arr[i]
  arr[i] = arr[j]
  arr[j] = temp
```

Bubble Sort

```
BUBBLE_SORT(arr):
  sorted = false
  while(!sorted):
    sorted = true
    for(i from 0 to n - 2):
      if (arr[i] > arr[i + 1]):
        swap(arr, i, j)
        sorted = false
```

https://www.youtube.com/watch?v=Cq7SMsQBEUw

SHOW VIDEO

- After each pass, the highest value "bubbles" to the top
- Worst Case, Typical Case = O(n^2)
- List already ordered = O(n)
- Note - if the smallest value is at the top, it will take n passes before it goes to the bottom
- Very inefficient with the swaps, which are more involved than comparisons

https://www.youtube.com/watch?v=8oJS1BMKE64

Insertion Sort

```
  For i = 1 to n - 1:
    //everything less that i is sorted at this point
    // Bubble down arr[i] to find out where it goes in arr[0]..arr[i-1]
    for(j = i; j > 0; j–)
      if (arr[j - 1] > arr[j]):
        swap(j - 1, j)
```

WORST CASE: O(n^2) - reverse sorted

Selection Sort

```
MIN(arr) {
  min = arr[0]
  minIndex = 0
  for(i = 1; i < arr.length; i++):
    if (arr[i] < min):
      min = arr[i]
      minIndex = i
}
```

https://www.youtube.com/watch?v=92BfuxHn2XE

```
SELECTION_SORT
  For i = 0 to N-1:
    // 1) Find the index of the min of all elements from arr[i]..arr[N-1]
    // 2) Swap it with position i
```

Selection sort also O(n^2) but now only O(n) swaps!