Quadratic sorting (2 lectures?)

* The SORTING PROBLEM is to start with an array or vector of items that are in any order, and put them in sorted order.
* This can of course be done with any data type where we can compare two items X, and Y, from that data type and say either X < Y, X > Y, or X=Y.
  + All that means is that you can sort objects by any key field. For instance, if we have Dog objects that have string name and int age, we can sort them either by name or by age.

Sorting is such a fundamental problem in computer science because it enables efficient access to things we’re looking for.

* That’s why practically everything electronic these days is sorted, because we naturally know where to find things.
* Think about workday (sorts classes by dept name), dictionaries (sorts things alphabetically)
* Even things which don’t seem to be sorted by something easy to quantify probably are.
  + Facebook – sorts posts kind of chronologically, but there’s also clearly some other criteria for prioritizing them.
  + Twitter – how does this sort things?

So the point is everything that has to be sorted there’s always some key field, usually numeric or a string, that we use to do the sorting.

Most sorting algorithms assume we’re putting things into ASCENDING order, but we can use the same algorithms to sort in DESCENDING order, all we do is switch the direction of the comparison from X < Y to X > Y.

Today we’re going to look at 3 quadratic sorting algorithms. Called quadratic because they are all O(n^2) algorithms.

**SELECTION SORT**

High level algorithm:

Finding smallest element in the array. Move this element to the front of the list.

Find the second smallest element. Move it to pos #2.

Find the third-smallest element. Move it to pos #3.

Repeat until we find the second-smallest element, and move to 2nd-to-last position.

* Illustrate with 67 33 21 84 49 50 75
* Find 21.
* Swap with 67.
* Scan sublist from index 1 to end.
* Find 33.
* Swap with itself.

Look at code. Ask some questions:

* why outer loop to n-2 (not n-1)? b/c last item is already in place at end
* why j=i+1, not j=i? no need to compare a[j] against itself in the next if statement.

Analysis of selection sort:

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the minimum requires scanning {\displaystyle n}n elements (taking {\displaystyle n-1}n-1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining {\displaystyle n-1}n-1 elements and so on. Therefore, the total number of comparisons is O(n^2).

Can this get any better?

What happens if the array is already sorted? Still must make n^2 comparisons. All swaps still happen, but with self.

What if sorted in reverse order? Same thing.

What if all same element? Same thing.

|  |  |
| --- | --- |
| [**Worst-case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | О(*n*2) comparisons, О(*n*) swaps |
| [**Best-case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | О(*n*2) comparisons, О(*n*) swaps |
| [**Average performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | О(*n*2) comparisons, О(*n*) swaps |

^^^ Performance never changes. Finding the min element can never go faster or slower, because the inner loop must always run from i+1 to n-1. And we always have to run it n times.

**BUBBLE SORT**

* 67 33 21 84 49 50 75
* Swap 67/33
* Swap 67/21
* Swap 84/49
* Swap 84/50
* Swap 84/75

Notice how 84 has bubbled up to the end!

Analysis:

* Already sorted (or all identical) -> best case = n comparisons, zero swaps.
* Reverse sorted -> worst case.
* Invariant: after k passes, the last k elements of the array (largest) are in their final spots (so an optimization is not looping over those last k elements on each pass).

|  |  |
| --- | --- |
| [**Worst-case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | {\displaystyle O(n^{2})} n^2 comparisons, {\displaystyle O(n^{2})} n^2 swaps |
| [**Best-case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | {\displaystyle O(n)} n comparisons, {\displaystyle O(1)} O(1) swaps |
| [**Average performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | {\displaystyle O(n^{2})} n^2 comparisons, {\displaystyle O(n^{2})} n^ 2 swaps |

^^^ Performance: Bubble sort does have different big-oh depending on the starting array. In the best case, we never make any swaps, because the array is already sorted, and therefore the outer loop only runs once.

**INSERTION SORT**

* Basic algorithm: In your head, maintain a separator between the SORTED portion of the array (starts as one element) and the unsorted portion (starts as everything but first element).
* Take the first element in the UNSORTED portion, and find where it goes in the SORTED PORTION.
* INSERT it into place (slide everything over).

Run on

* 67 33 21 84 49 50 75
* Start: 67 | 33 21 ...
  + Put 33 into place in sorted portion
  + 67 slides over to take 33's spot
  + 33 goes at beginning.
* Now: 33 67 | 21 84...
  + Put 21 into place
  + slide 67 over
  + slide 33 over
  + 21 goes at beginning
* Now: 21 33 67 | 84 49...
  + 84 stays where it is.
* Now: 21 33 67 84 | 49 50...

|  |
| --- |
|  |
| [**Worst-case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | О(*n*2) comparisons and swaps |
| [**Best-case performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n*) comparisons, O(*1*) swaps |
| [**Average performance**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) | О(*n*2) comparisons and swaps |

**INSERTION analysis**

* Insertion sort is very similar to [selection sort](https://en.wikipedia.org/wiki/Selection_sort).
* As in selection sort, after *k* passes through the array, the first *k* elements are in sorted order.
* However, the fundamental difference between the two algorithms is that for selection sort these are the *k* smallest elements of the unsorted input, while in insertion sort they are simply the first *k* elements of the input.
* The primary advantage of insertion sort over selection sort is that selection sort must always scan all remaining elements to find the absolute smallest element in the unsorted portion of the list, while insertion sort requires only a single comparison when the *k*+1th element is greater than the *k*th element; when this is frequently true (such as if the input array is already sorted or partially sorted), insertion sort is distinctly more efficient compared to selection sort.

Already sorted (or all same) -> best case -> linear run time. During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

Reverse order sorted = worst case.

COMPARE

* all 3 are n^2
* Primary virtue of selection sort is simplicity.
* Sel sort: no speed up for lists that are already sorted.
  + many applications involve lists that are already partially sorted
* bubble sort is good for lists that are already sorted but in general is awful
  + makes way too many swaps
  + least efficient. Never use!
* Insertion sort is really the most widely used of the 3 in practice (though there are better sorts)
  + best for lists <= 20 elements.

<https://www.youtube.com/watch?v=m4yVlPqeZwo> 🡨 Obama at google/start at 22:50

In the lead up to the 2008 presidential election, which eventually pitted Barack Obama against John McCain, a number of candidates, including both Obama and McCain, visited Google’s headquarters in California, to talk about their technology policy positions. Obama

Have students run all 3 on

3 7 4 9 5 2 6 1

INSERTION EXAMPLE

3 7 4 9 5 2 6 1

3\* 7 4 9 5 2 6 1

3 7\* 4 9 5 2 6 1

3 4\* 7 9 5 2 6 1

3 4 7 9\* 5 2 6 1

3 4 5\* 7 9 2 6 1

2\* 3 4 5 7 9 6 1

2 3 4 5 6\* 7 9 1

1\* 2 3 4 5 6 7 9