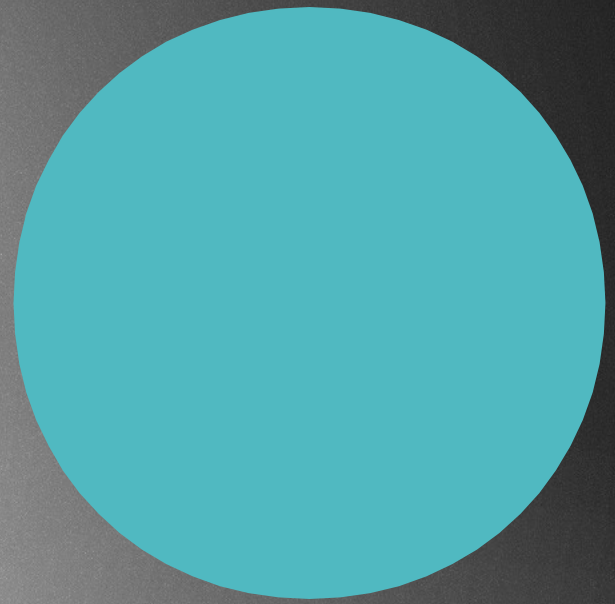
SORTING ALGORITHMS



Sorting

- A sorting algorithm is an algorithm that puts elements of an array in a certain order
- Numbers → numerical ordering !!!
- Strings, characters → alphabetical ordering !!!
- **Comparison based** algorithms
  ~ bubble sort, insertion sort, selection sort, merge sort, quicksort
- **Non-comparison based** sorting
  ~ radix sort, bucket sort

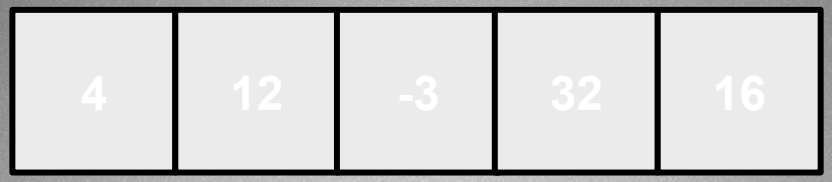
Features

- Time complexity: **O(N ^{2})** or **O(N log N)** or **O(N)**
- In place: strictly an in-place sort needs only **O(1)** memory beyond the items being sorted

  So an in place algorithm does not need any extra memory !!!
- Recursive: some sorting algorithms are implemented in a recursive manner → the divide and conquer ones especially

  // merge sort and quicksort
- Stable: stable sorting algorithms maintain the relative order of records with equal values

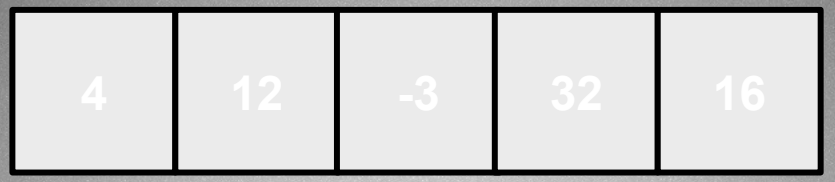
In place feature



An in place algorithm will not allocate any extra memory,
for example a temporary array in order to make the sorting !!!

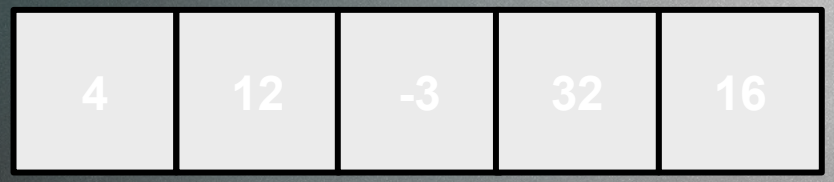
For merge sort → we need some extra memory

In place feature

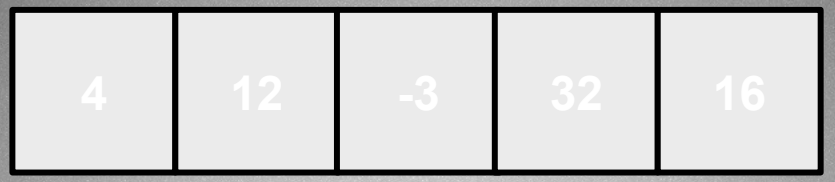


An in place algorithm will not allocate any extra memory,
for example a temporary array in order to make the sorting !!!

For merge sort → we need some extra memory



In place feature



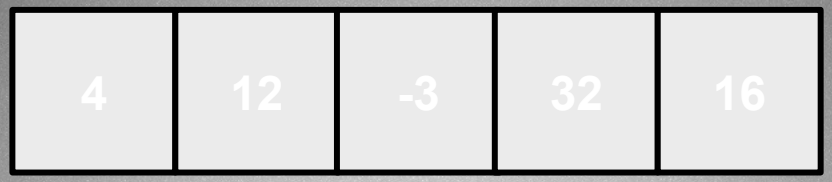
An in place algorithm will not allocate any extra memory,
for example a temporary array in order to make the sorting !!!

For merge sort → we need some extra memory

IN PLACE !!!

For example: quicksort

In place feature



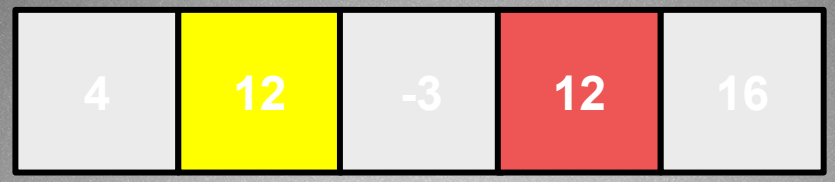
An in place algorithm will not allocate any extra memory,
for example a temporary array in order to make the sorting !!!

For merge sort → we need some extra memory

Sometimes we have some extra space when storing the numbers we want to sort → not going to be in place !!!

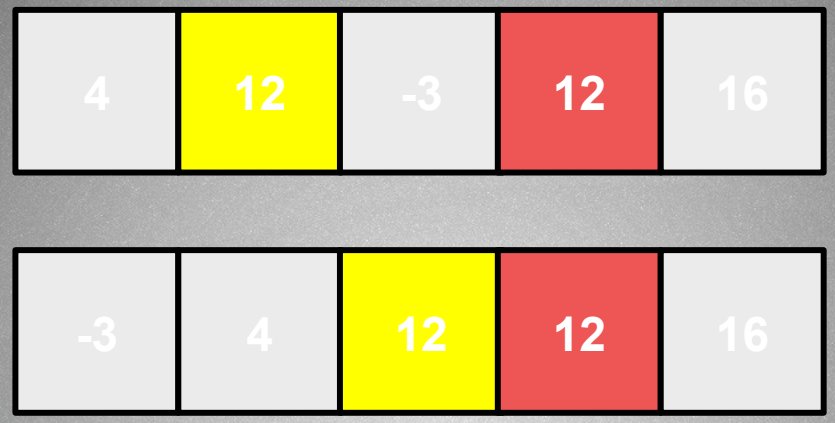
Why is it good to have algorithm that are in-place?
MEMORY EFFICIENT !!!

Stable algorithms

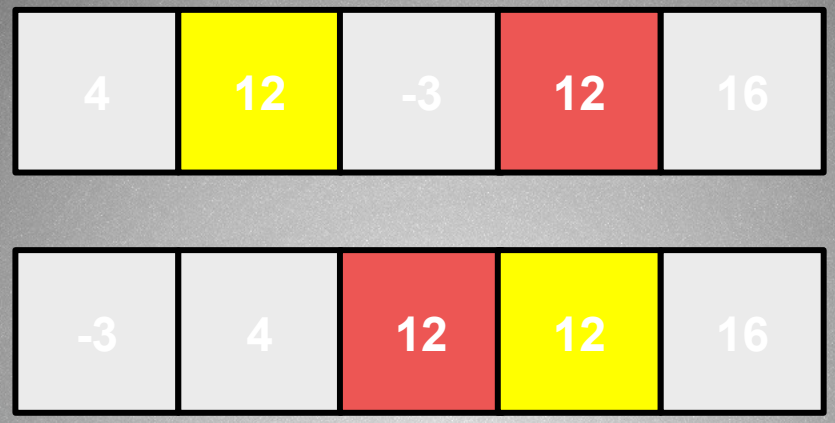


before sorting

Stable algorithms



Stable algorithms



So the relative order of equal items remain the same

The red 12 is after the yellow 12 even after sorting !!!

Merge sort: stable

Quicksort: unstable

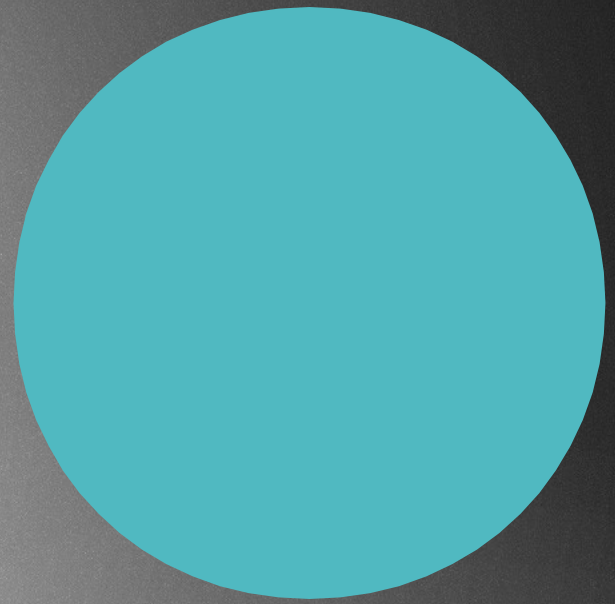
Lower bound

For sorting **N** items → we have to make **log N!** comparisons
With Stringling-formula it can be reduced to **N logN**

- so the **Ω(N logN)** time complexity is the lower bound for **comparison based** sorting algorithms
- ok but we can achieve **O(N)** running time as far as sorting is concernded, such as bucket sort or radix sort

THESE ARE NOT COMPARISON BASED ALGORITHMS !!!

SORTING ALGORITHMS



BOGO SORT

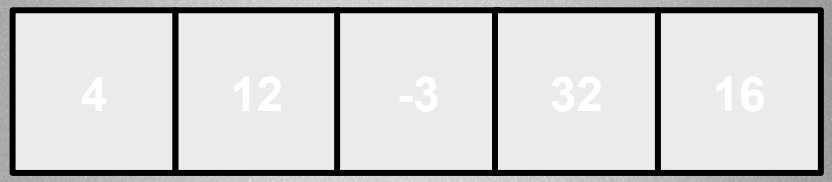
Bogo sort

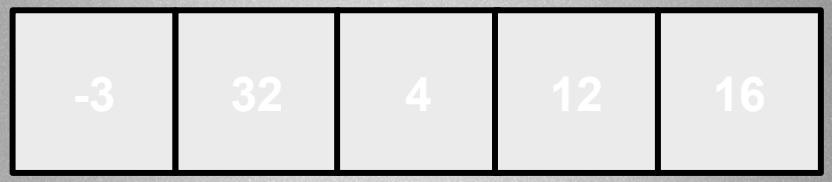
- Also known as **permutation sort** or **shotgun sort**
- A particularly ineffective sorting algorithm
- The algorithm keeps generating permutations of its input until it finds one that is sorted
- **O((n+1)!)** time complexity
- Two variants

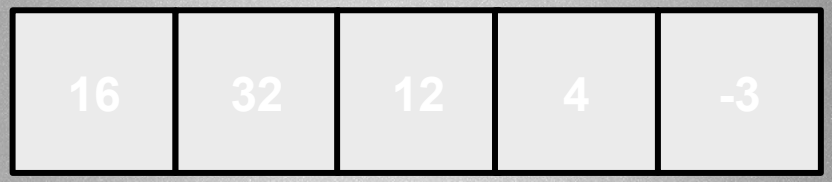
  1.) deterministic version that enumerates all permutations until it hits a sorted one

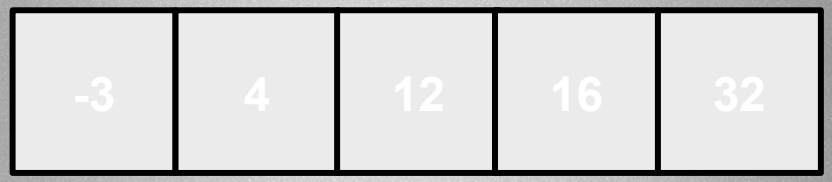
  2.) randomized one: we randomly permutate the input until we find

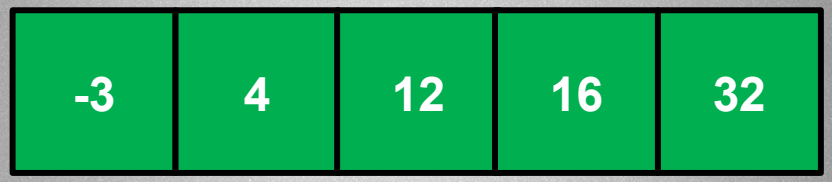
  the solution // the sorted array

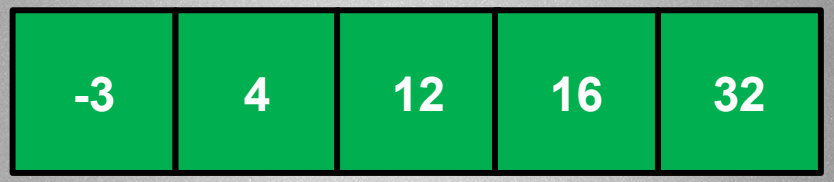

4
12
-3
32
16


-3
32
4
12
16


16
32
12
4
-3


-3
4
12
16
32


-3
4
12
16
32

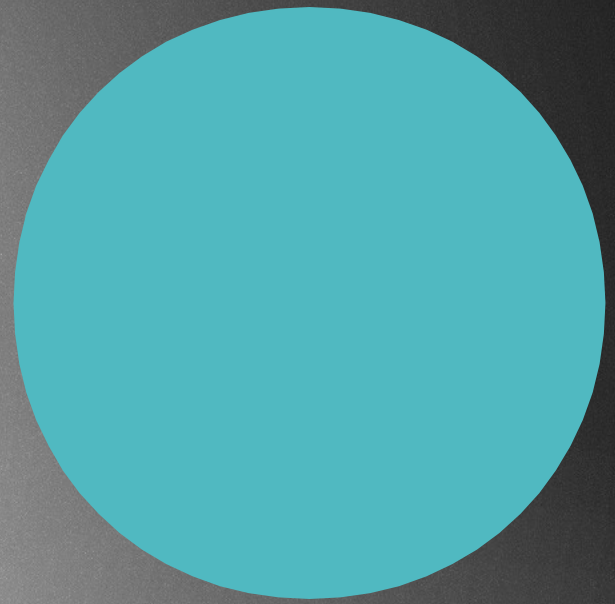


Why are we talking about the most inefficient sorting algorithm?

For classical computers → it is inefficent

For **quantuum computers** → **O(1)** running time is guaranteed !!!
Because of quantuum entanglement we can „search” for every possible combinations simultaneously

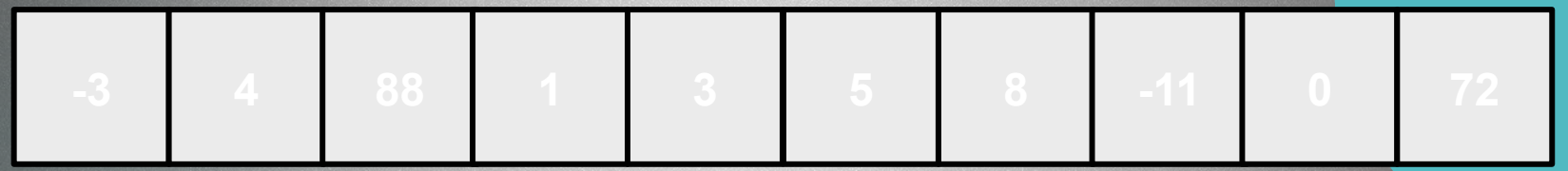
SORTING ALGORITHMS

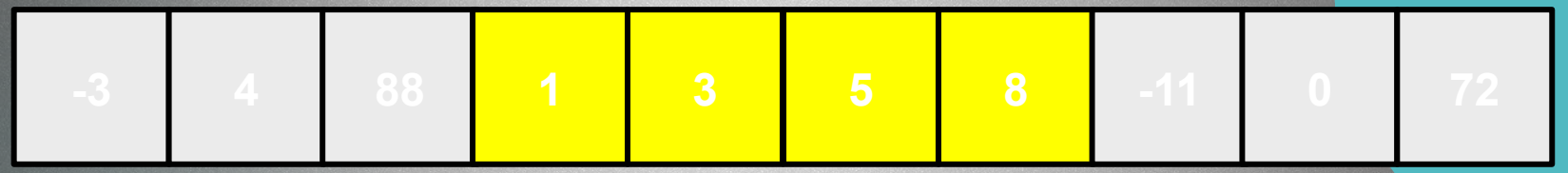


ADAPTIVE SORTING

Adaptive algorithms

- An adaptive algorithm is an algorithm that changes its behavior based on information available at runtime
- Adaptive sort → it takes advantage of existing order in its input
- It benefits from local orders → sometimes an unsorted array contains sequences that are sorted by default → the algorithms will sort faster
- Most of the times: we just have to modify existing sorting algorithms in order to end up with an adaptive one


-3
4
88
1
3
5
8
-11
0
72

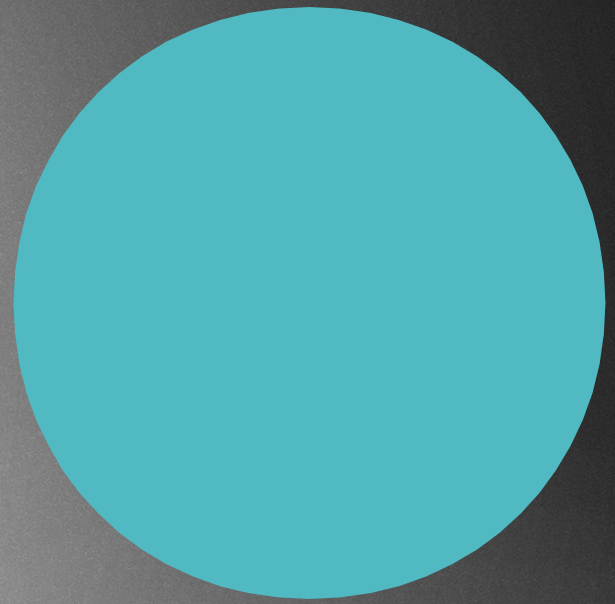


It is a sorted subarray !!!

Adaptive algorithms

- Comparison based algorithms have optimal **O(N logN)** running time complexity
- Adaptive sort takes advantage of the existing order of the input to try to achieve better times: maybe **O(N)** could be reached
- The more presorted the input is, the faster it should be sorted
- **IMPORTANT**: nearly sorted sequences are common in practice !!!
- Heapsort, merge sort: not adaptive algorithms, do not take advantage of presorted sequences
- Shell sort: adaptive algorithm so performs better if the input is partially sorted

SORTING ALGORITHMS



BUBBLE SORT

Bubble sort

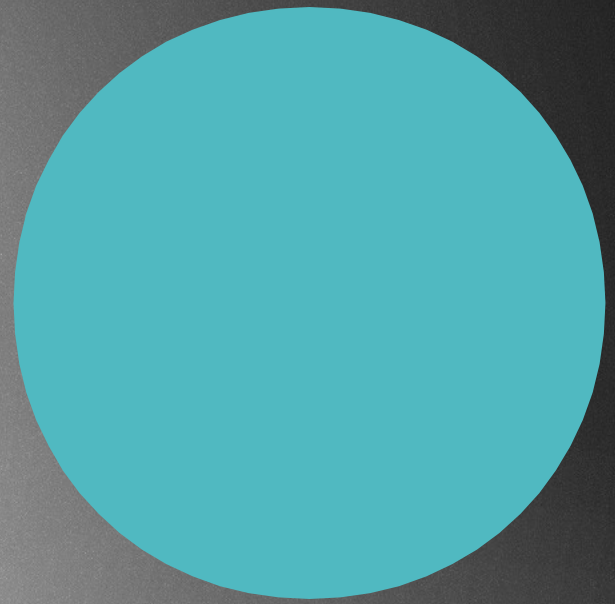
- Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order
- It is too slow and impractical for most problems even when compared to insertion sort
- Bubble sort has worst-case and average complexity both **$O(N^2)$**
- Bubble sort is not a practical sorting algorithm
- It will not be efficient in the case of a reverse-ordered collection
- Stable sorting algorithm
- In place algorithm → does not need any additional memory

Bubble sort

- In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity **O(N)**
- For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their **x** coordinates at a specific scan line (a line parallel to **x** axis) and with incrementing **y** their order change (two elements are swapped) only at intersections of two lines

Bubble sort

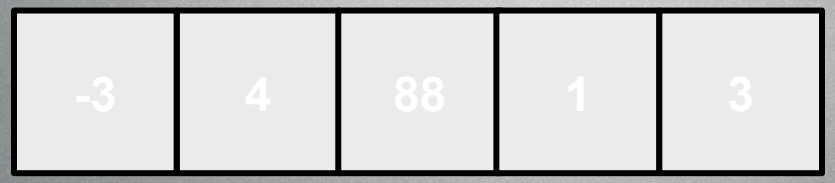
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



Bubble sort

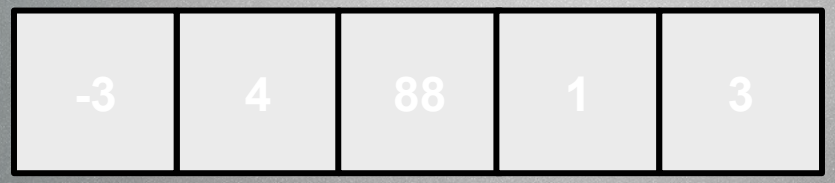
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```

We iterate through
all the items in the array !!!



Bubble sort

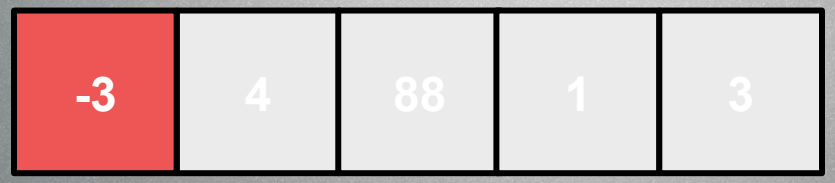
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



We keep considering fewer and fewer items, because on every iteration we consider one more item to be sorted !!!

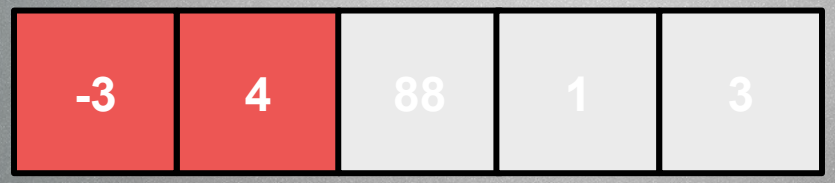
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



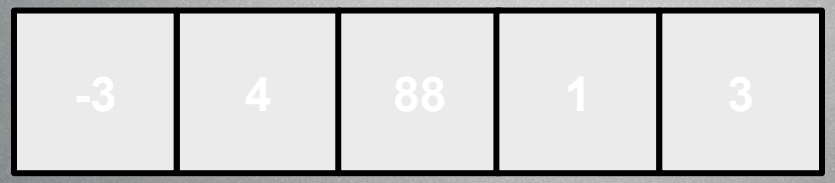
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



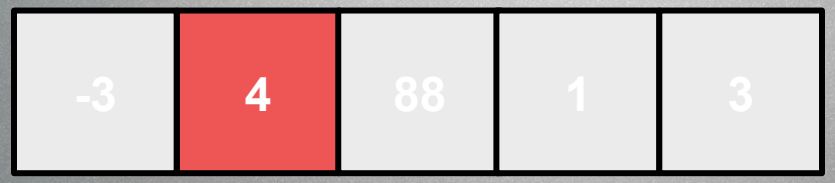
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



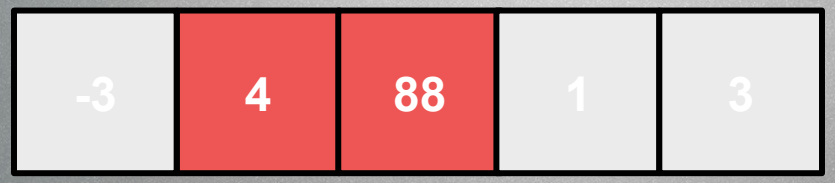
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



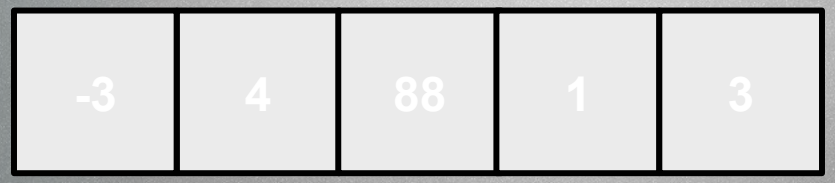
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



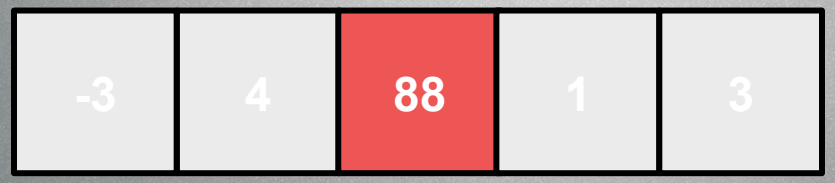
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



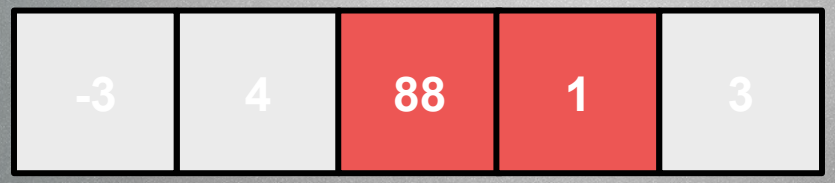
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



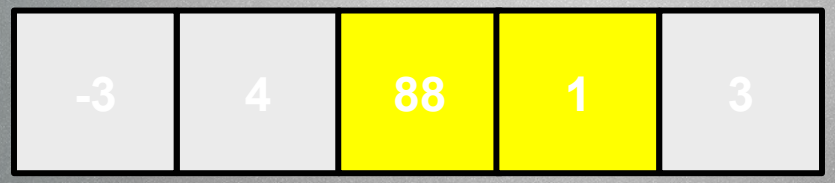
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



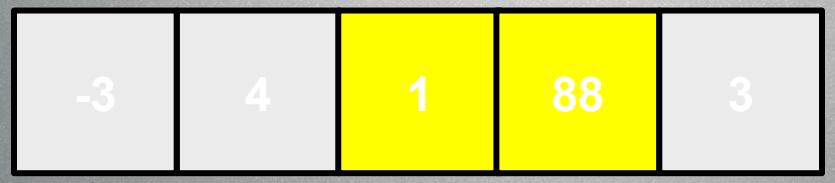
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



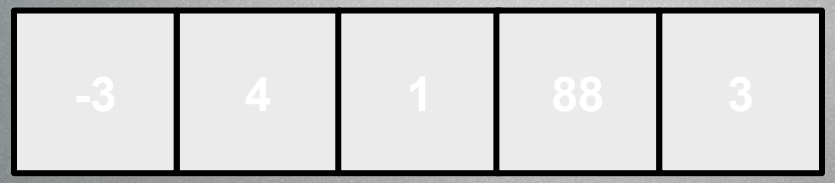
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



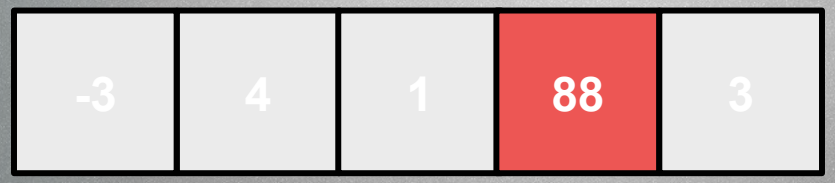
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



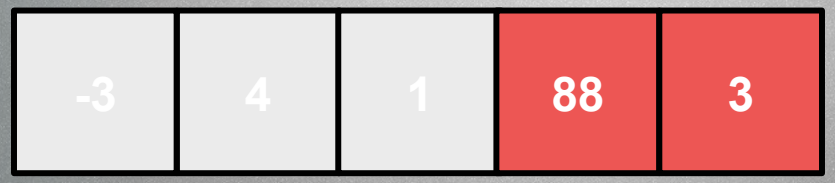
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



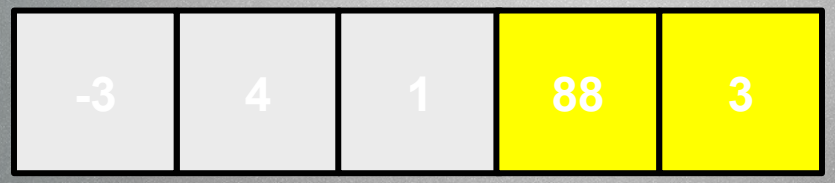
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



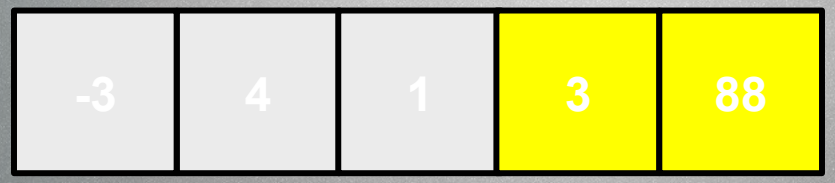
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



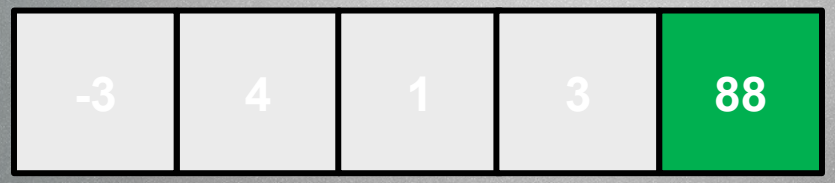
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



Bubble sort

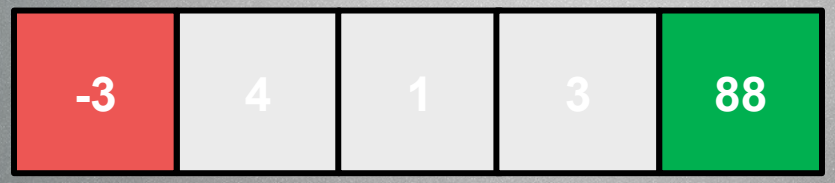
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

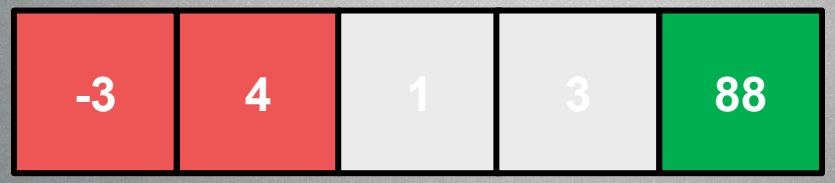
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

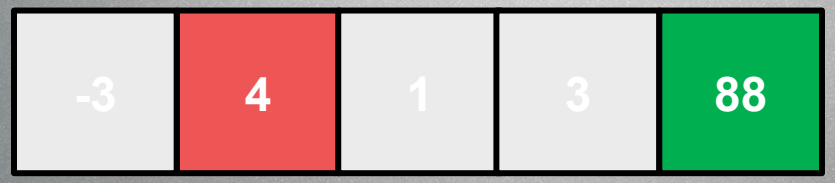
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

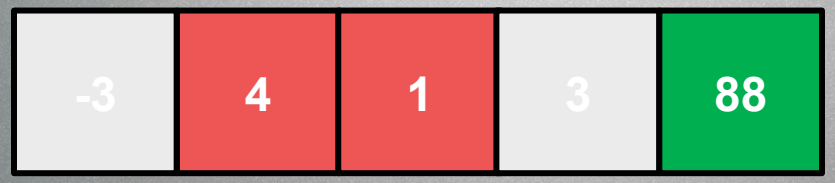
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

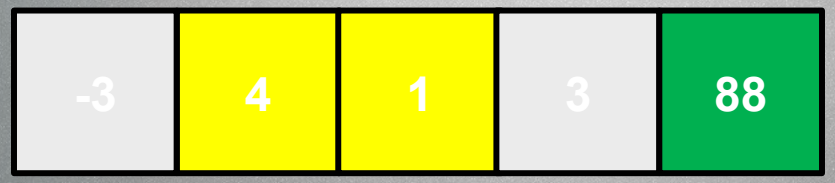
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

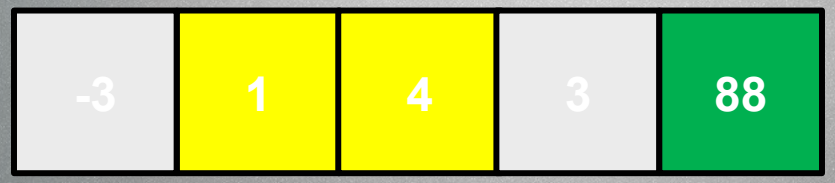
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

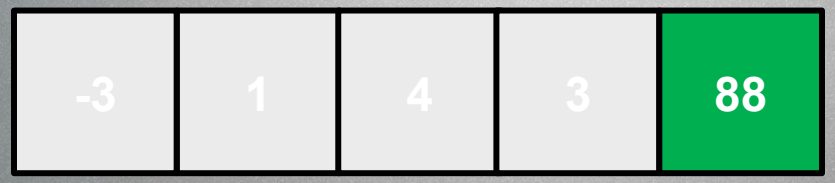
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

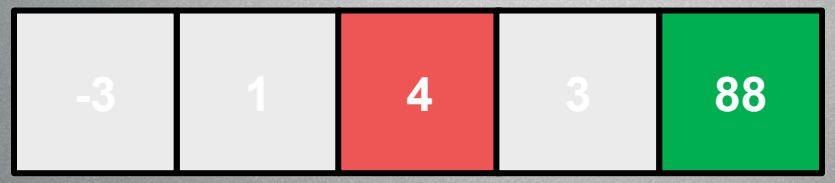
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

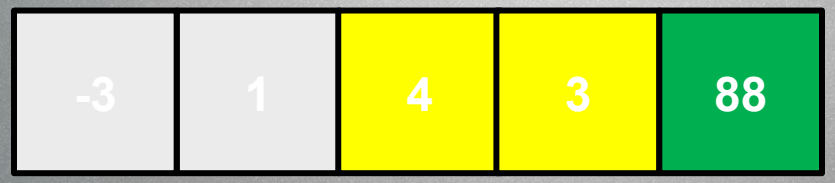
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```

On every iteration we bubble up the largest item !!!

Bubble sort

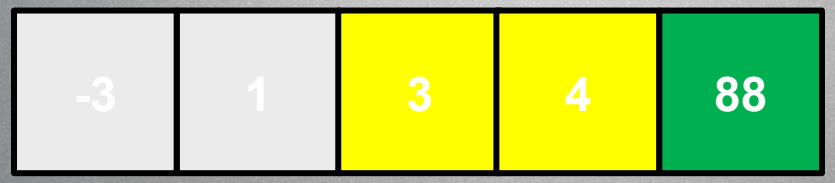
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

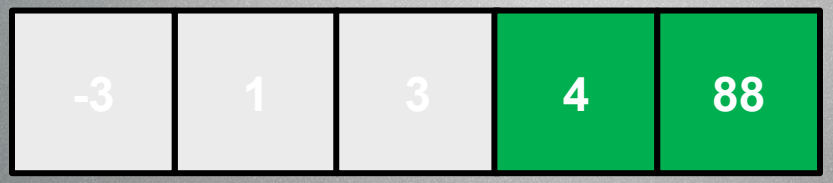
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

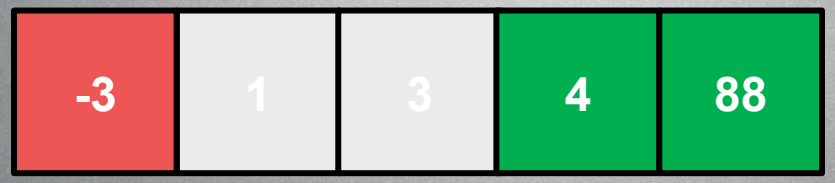
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

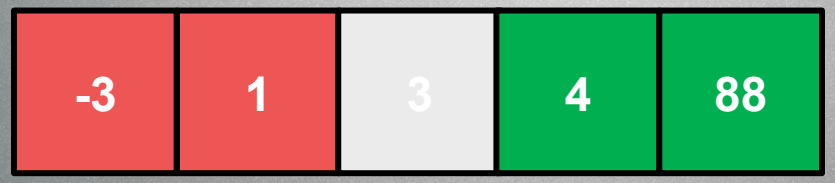
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

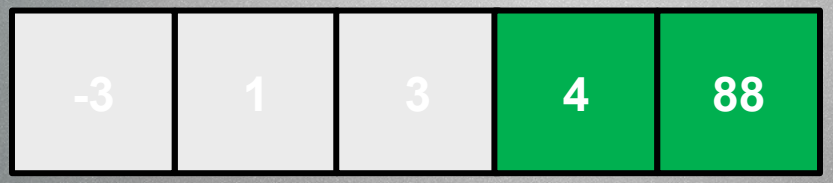
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

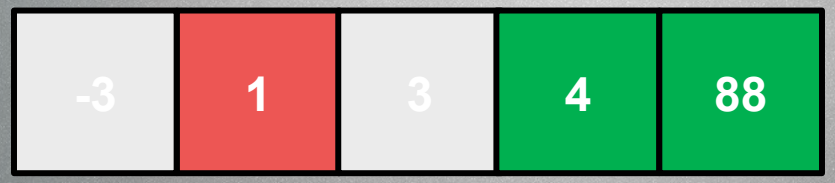
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

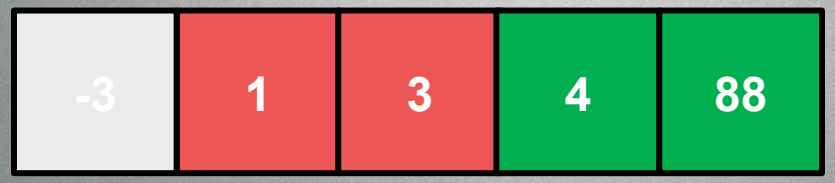
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

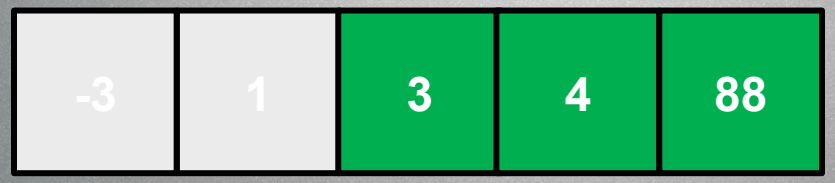
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

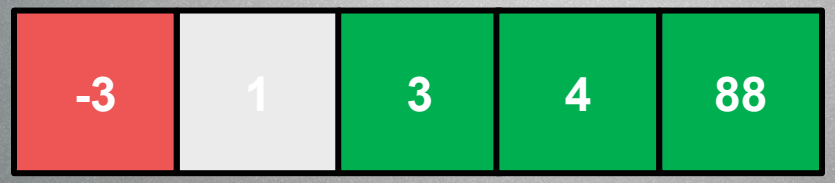
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

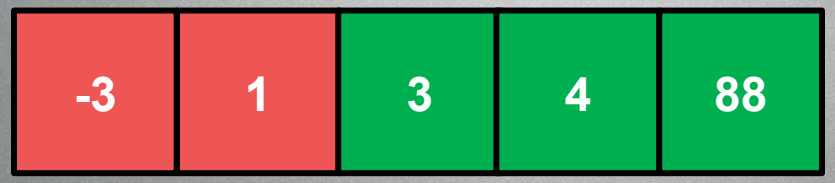
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

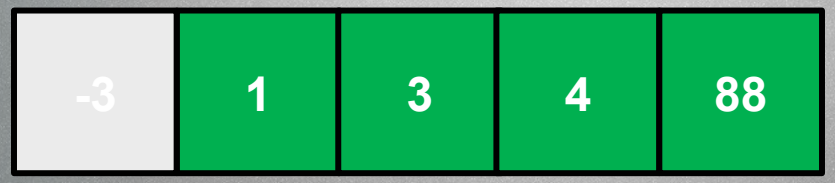
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

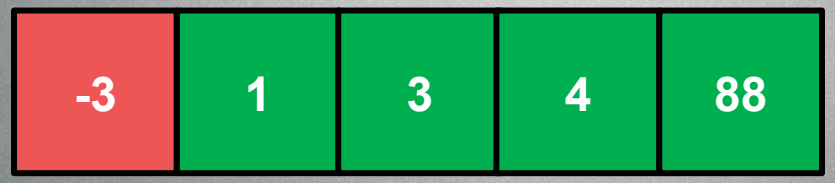
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

Bubble sort

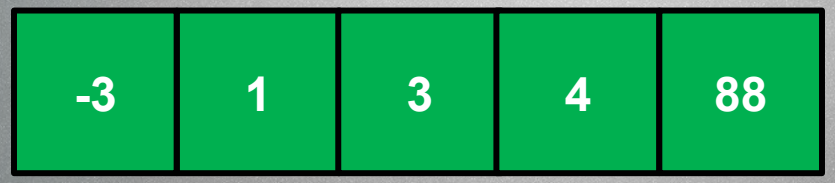
```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

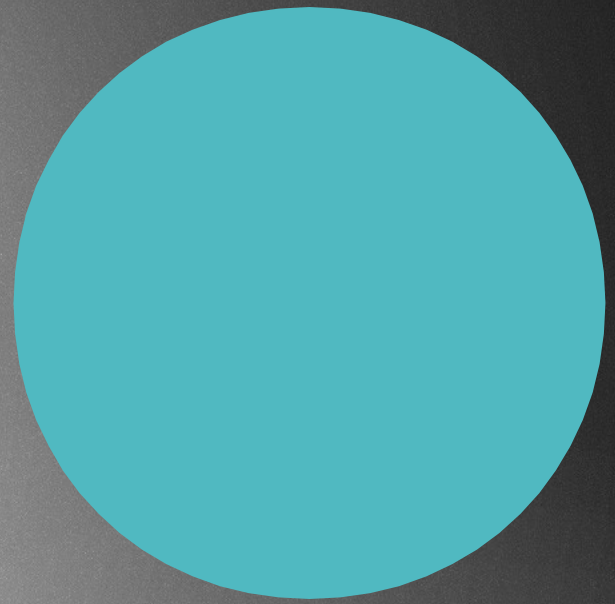
Bubble sort

```
bubbleSort(array)

    for i in range array.length-1
        for j in range array.length-1-i
            if array[j] > array[j+1]
                swap(array,j,j+1)

end
```



On every iteration we bubble up the largest item !!!

SORTING ALGORITHMS



SELECTION SORT

Selection sort

- Another **O(N ^{2})** running time sorting algorithm
- Selection sort is noted for its simplicity and it has performance advantages over more complicated algorithms
- Particularly useful where auxiliary memory is limited
- The algorithm divides the input list into two parts:
  - the subarray of items already sorted
  - and the subarray of items remaining to be sorted that occupy the rest of the array

Selection sort

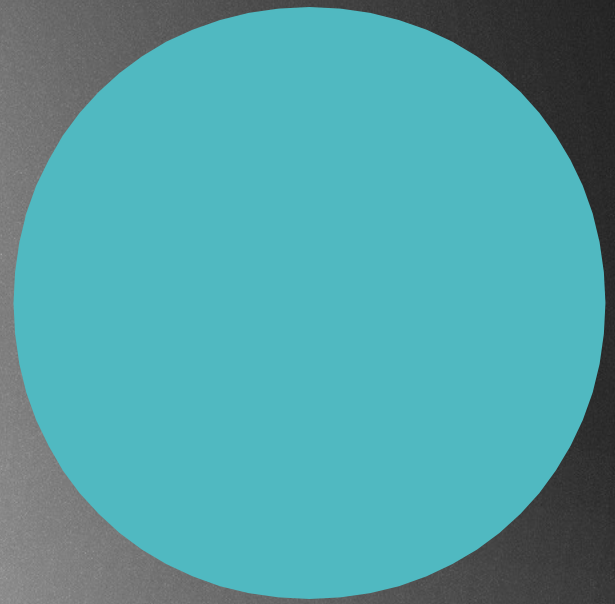
- The algorithm proceeds by finding the smallest element in the unsorted subarray
- Exchange / swap it with the leftmost unsorted element → putting it in sorted order
- Moving the subarray boundaries one element to the right
- It is an in place algorithm → no need for extra memory
- Selection sort almost always outperforms bubble sort
- Not a stable sort → does not preserve the order of keys with equal values

Selection sort

- Quite counter-intuitive: selection sort and insertion sort are both typically faster for small arrays // arrays with **10-20** items
- Usual optimization method → recursive algorithms switch to insertion sort or selection sort for small subarrays
- Makes less writes than insertion sort → this can be important if writes are significantly more expensive than reads,
- For example with **EEPROM** or flash memory where every write lessens the lifespan of the memory

Selection sort

```
selectionSort(array)

    for i in range array.length-1
        index = i

        for j from i+1 to array.length
            if array[j] < array[index]
                index = j

        if index not i
            swap(array, index, i)

end
```



Selection sort

```
selectionSort(array)

    for i in range array.length-1
        index = i

        for j from i+1 to array.length
            if array[j] < array[index]
                index = j

        if index not i
            swap(array, index, i)

end
```

We have to consider all the items

Selection sort

```
selectionSort(array)

    for i in range array.length-1
        index = i

        for j from i+1 to array.length
            if array[j] < array[index]
                index = j

    if index not i
        swap(array, index, i)

end
```

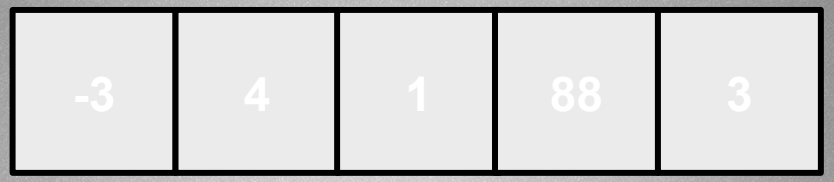
Basically we make a simple linear search for the minimum element !!!

Selection sort

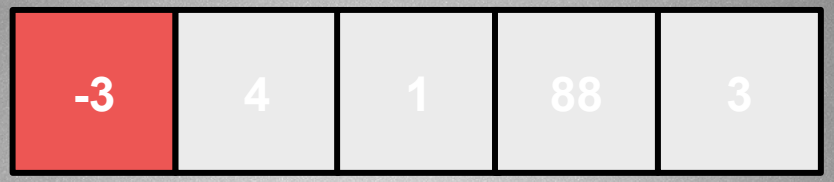
```
selectionSort(array)

    for i in range array.length-1
        index = i

        for j from i+1 to array.length
            if array[j] < array[index]
                index = j

        if index not i
          swap(array, index, i)

end
```

If ***index = i*** it means the smallest item is index i so no need to swap the number with itself

Selection sort

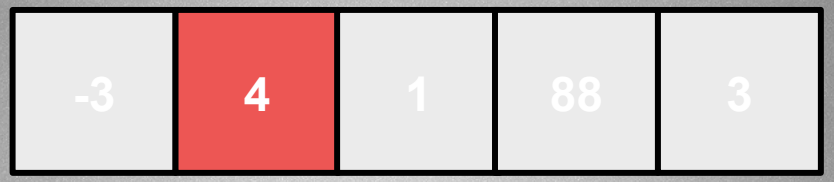


Selection sort



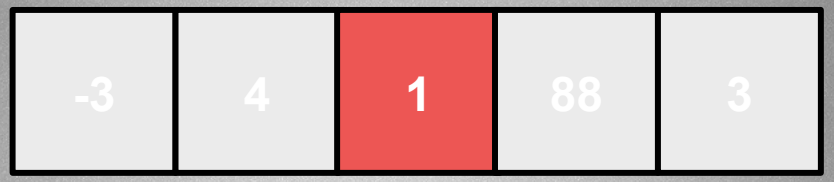
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



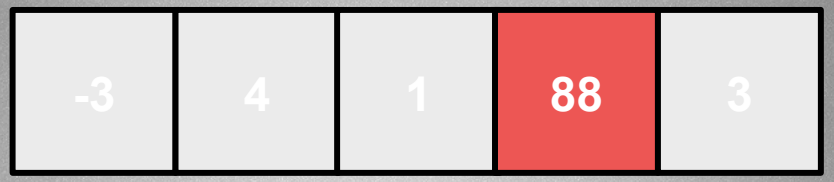
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



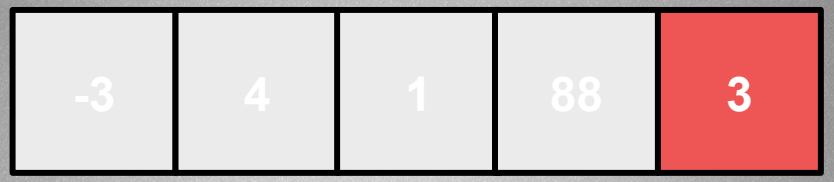
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



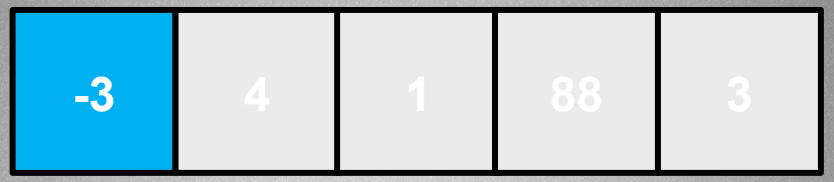
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

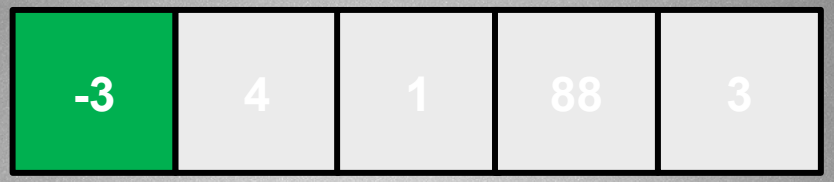
Selection sort



We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Minimum item: -3 → swap it with the leftmost item

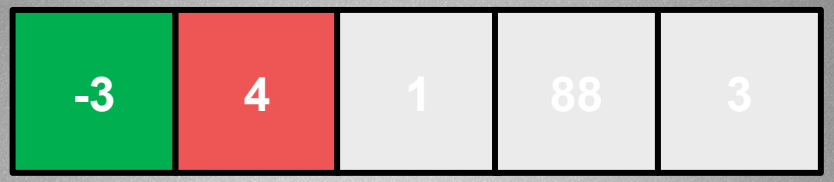
Selection sort



We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

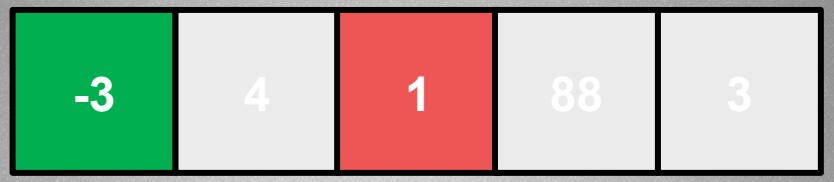
Minimum item: -3 → swap it with the leftmost item

Selection sort



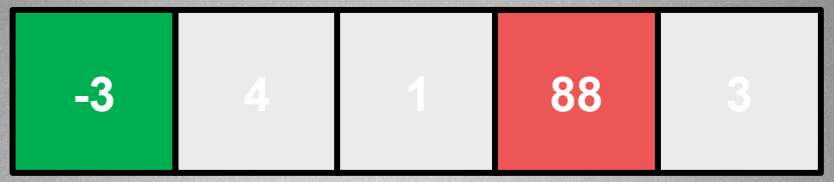
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



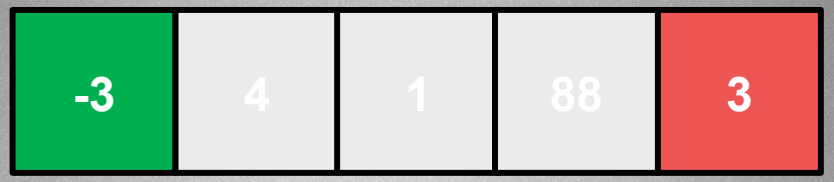
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



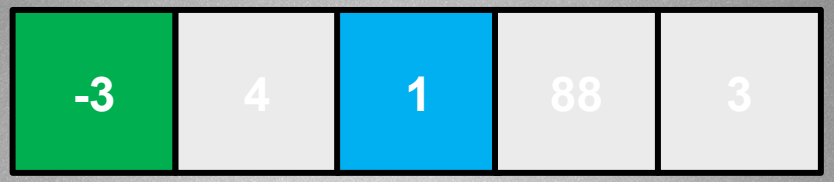
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

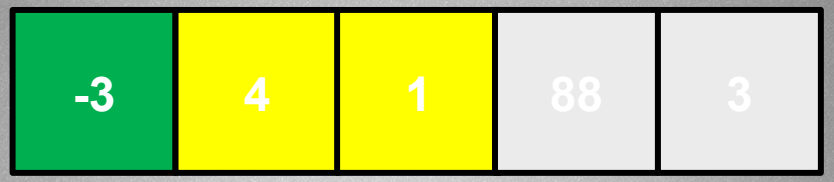
Selection sort



We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

Smallest value: 1 → we have to swap it with the leftmost item that has not been considered sorted

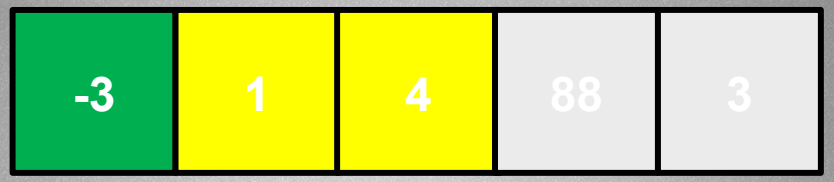
Selection sort



We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

Smallest value: 1 → we have to swap it with the leftmost item that has not been considered sorted

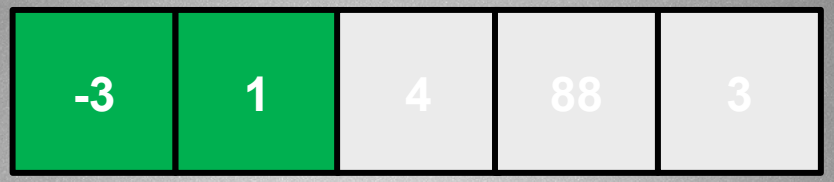
Selection sort



We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

Smallest value: 1 → we have to swap it with the leftmost item that has not been considered sorted

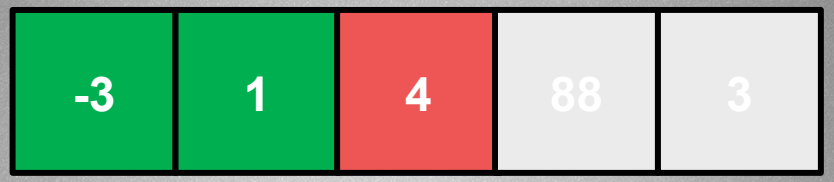
Selection sort



We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

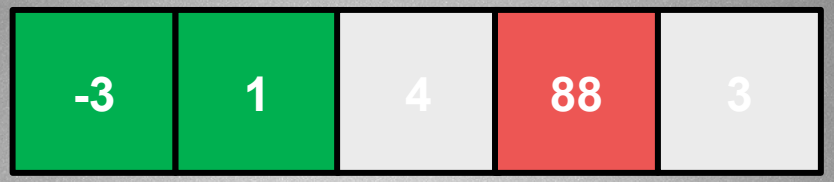
Smallest value: 1 → we have to swap it with the leftmost
item that has not been considered sorted

Selection sort



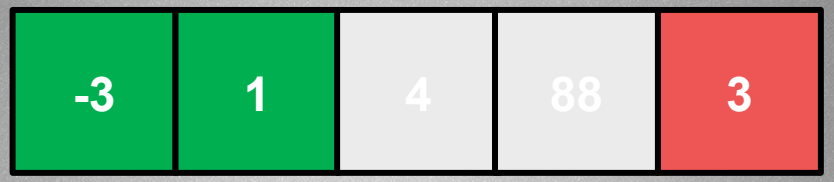
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



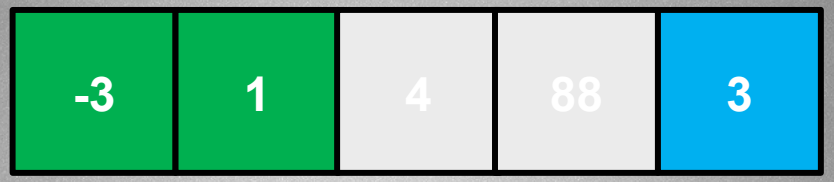
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

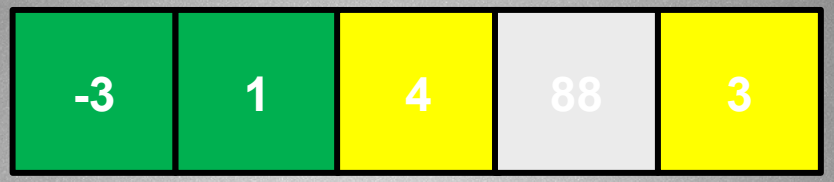
Selection sort



We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

Minimum item: 3 → we have to swap it with the leftmost item that has not been sorted yet

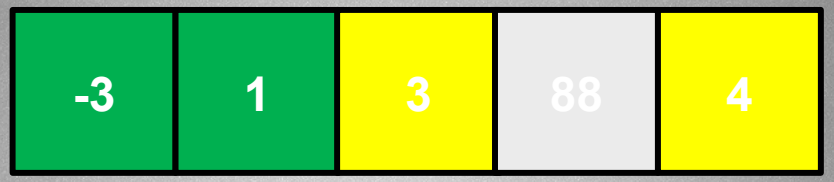
Selection sort



We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

Minimum item: 3 → we have to swap it with the leftmost item that has not been sorted yet

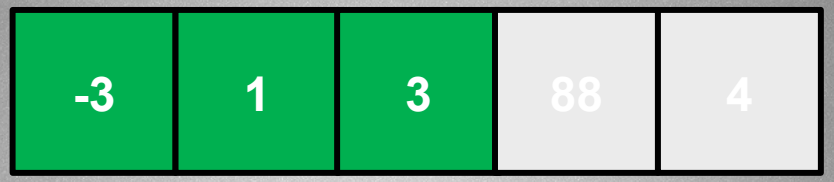
Selection sort



We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

Minimum item: 3 → we have to swap it with the leftmost item that has not been sorted yet

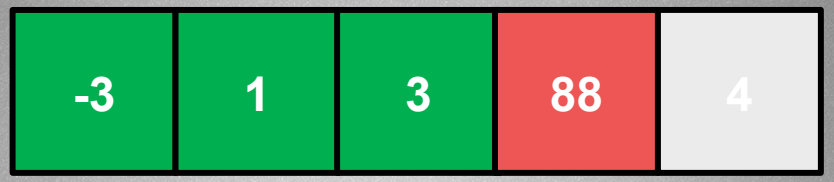
Selection sort



We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

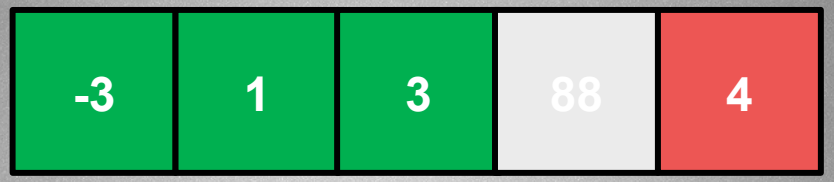
Minimum item: 3 → we have to swap it with the leftmost item that has not been sorted yet

Selection sort



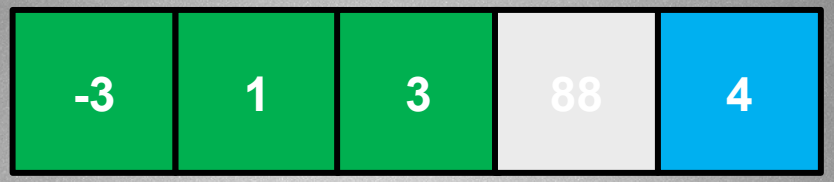
We find the minimum: for this we have to iterate through the whole array with **O(N)** time complexity ~ linear search

Selection sort



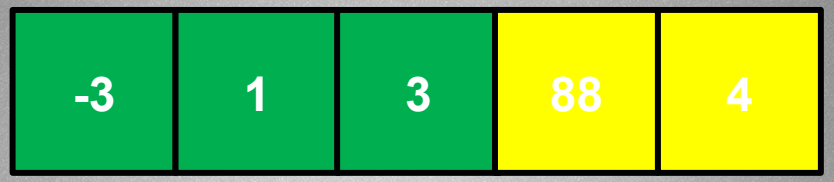
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



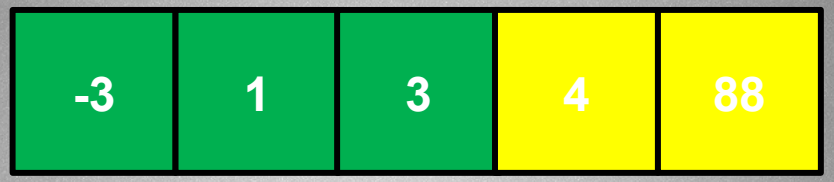
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



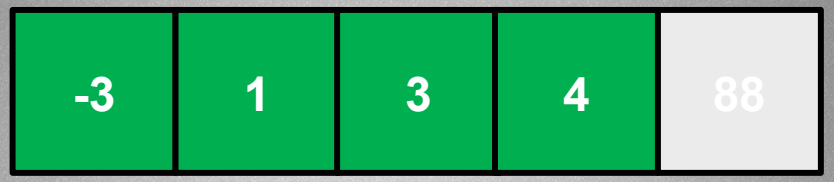
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



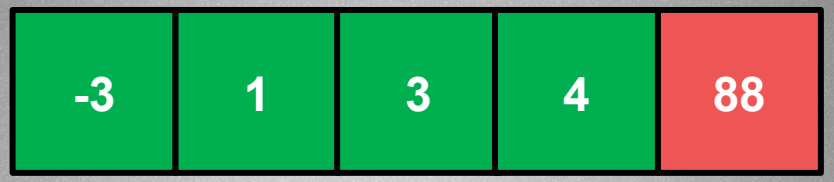
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



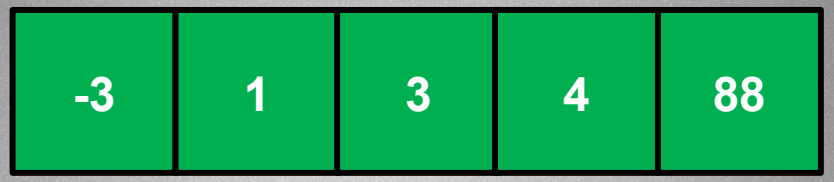
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



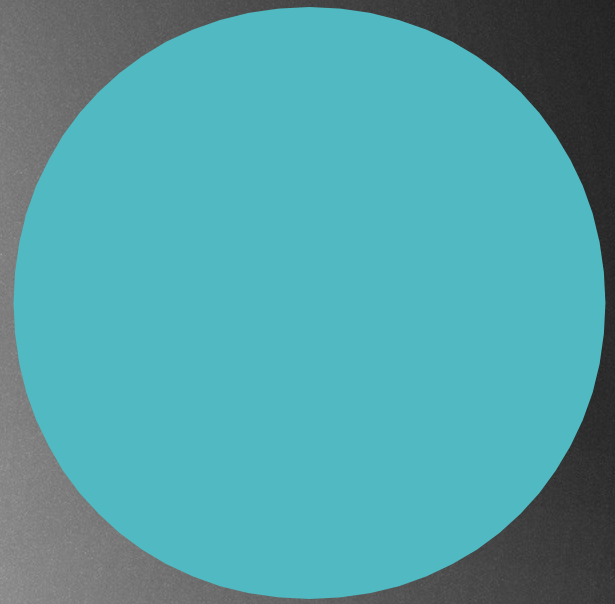
We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

Selection sort



We find the minimum: for this we have to iterate through the whole array
with **O(N)** time complexity ~ linear search

SORTING ALGORITHMS



QuickSORT