Data Structures and Algorithms

Lesson 10. Sorting

Alexandra Ciobotaru

28 apr 2022

Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- **Lesson 7. Hashing and Symbol Tables.**
- **Lesson 8. Graphs.**
- **Lesson 9. Searching.**
- **Lesson 10. Sorting.** Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

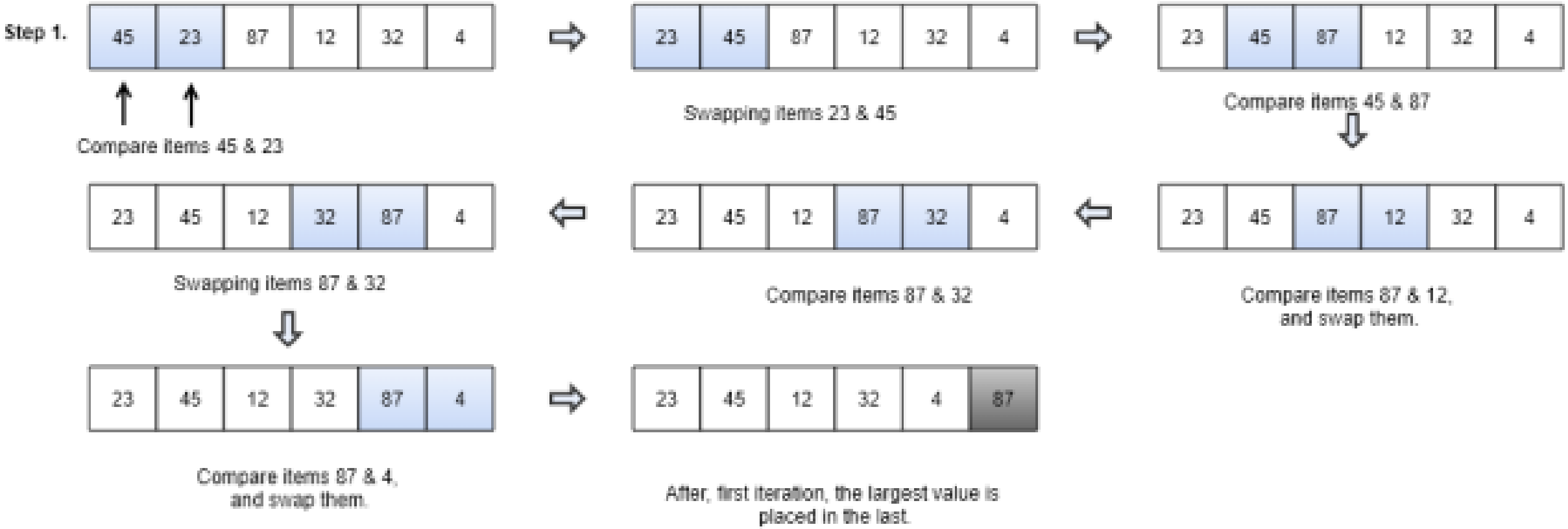
Bubble sort

- Suppose we want to sort the following list of elements: 4,2,6,5,1,3.
- Given an unordered list, we compare adjacent elements in the list, and after each comparison, we place them in the right order of magnitude - this works by **swapping** adjacent items if they are not in the correct order.
- The process is repeated n-1 times for a list of n items.
- In each such iteration, the largest element is arranged at the end of the list.
- For example, in the first iteration, the largest element would be placed in the last position of the list, and again, the same process will be followed for the remaining n-1 items.
- We start with the *first* item in the list, and if it's larger than the second item, we switch them; then we look at the *second* and if it's larger than the third, we switch them.. and so on.
- 2,4,6,5,1,3 – 2,4,6,5,1,3 – 2,4,5,6,1,3 – 2,4,5,1,6,3 – 2,4,5,1,3,**6** (6 is now sorted, we've bubbled up the largest item)
- In order to do this, we did 5 comparisons.

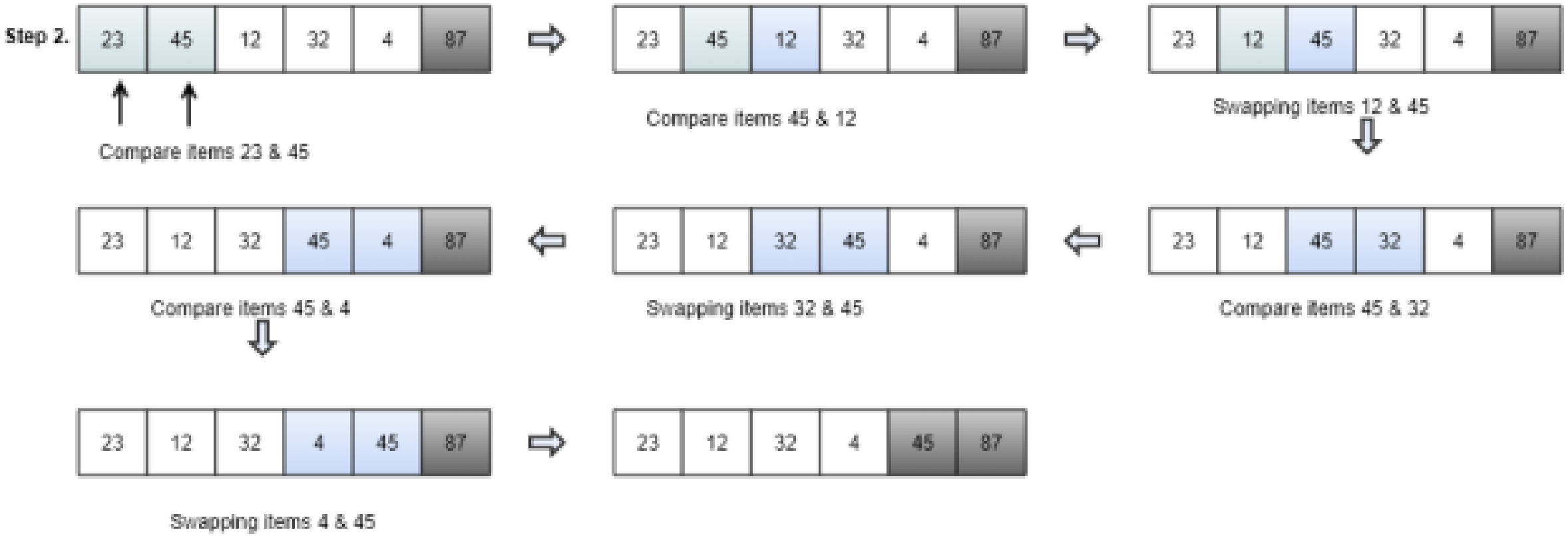
Bubble sort

- 2,4,6,5,1,3 – 2,4,6,5,1,3 – 2,4,5,6,1,3 – 2,4,5,1,6,3 – 2,4,5,1,3,**6**
- Now we only have 5 items left to sort, so in this round we will only have 4 comparisons.
- In the second iteration, the second largest element will be placed at the second-to-last position in the list, and the process will then be repeated until the list is sorted.
- 2,4,5,1,3,6 – 2,4,5,1,3,6 – 2,4,1,5,3,6 – 2,4,1,3,**5**,6 (now the second largest element has been sorted – has been bubbled up)
- 2,4,1,3,5,6 – 2,1,4,3,5,6 – 2,1,3,**4**,5,6 (now the fourth item is bubbled up)
- 1,2,3,4,5,6 - 1,2,**3**,4,5,6 (now the third item is bubbled up)
- 1,**2**,3,4,5,6 (and finally we have a completely sorted list)

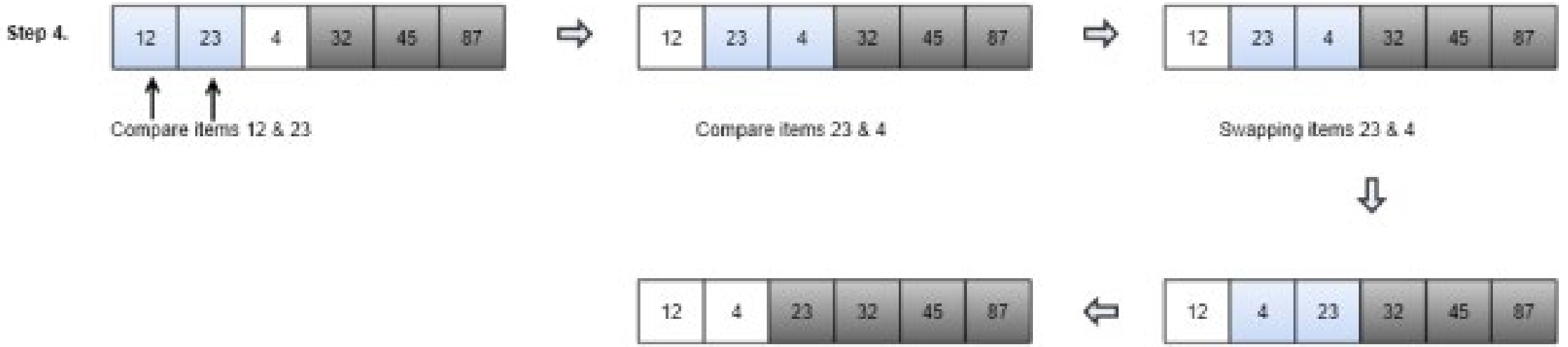
Bubble sort – another example



Bubble sort – another example



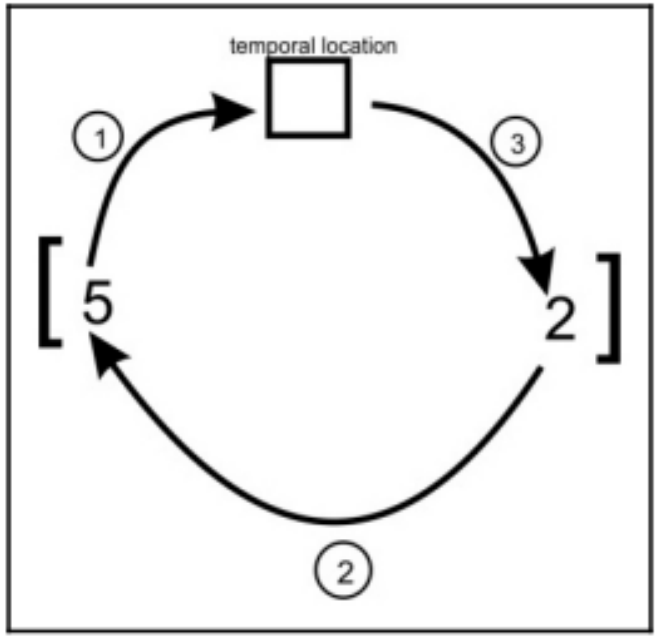

Step 3.
23 12 32 4 45 87
Compare items 23 & 12
12 23 32 4 45 87
Swapping items 12 & 23
12 23 32 4 45 87
Compare items 23 & 32
12 23 4 32 45 87
Swapping items 32 & 4
12 23 4 32 45 87
Compare items 32 & 4


Step 4.
12 23 4 32 45 87
Compare items 12 & 23
12 23 4 32 45 87
Compare items 23 & 4
12 23 4 32 45 87
Swapping items 23 & 4
12 4 23 32 45 87
12 4 23 32 45 87


Step 5.
12 4 23 32 45 87
Compare items 12 & 4
4 12 23 32 45 87
Swapping items 12 & 4
4 12 23 32 45 87

Bubble sort

- Switching two elements will be done by using a temporal value



Bubble Sort - Code

```
def bubble_sort(my_list):
    # we create a for loop in which we decrement at each iteration
    for i in range(len(my_list) - 1, 0 ,-1):
        # we create a second loop that goes forward through the list
        # and here we do the comparisons and switches, using temp
        for j in range(i):
            if my_list[j] > my_list[j+1]:
                temp = my_list[j]
                my_list[j] = my_list[j+1]
                my_list[j+1] = temp
    return my_list

print(bubble_sort([4,2,6,5,1,3]))
```

Insertion Sort

- The insertion sorting algorithm maintains a sub-list that is always sorted, while the other portion of the list remains unsorted.
- We take elements one by one from the **unsorted sub-list and insert them in the correct location in the sorted sub-list,** in such a way that this sub-list remains sorted.
- In insertion sorting, we start with one element, assuming it to be sorted, and then take another element from the unsorted sub-list and place it at the correct position (in relation to the first element) in the sorted sub-list.
- We repeatedly follow this process to insert all the elements one by one from the unsorted sub-list into the sorted sub-list.

Step 1.

Sublist 1 is sorted.

Step 2.

Insert 23 at correct position in sub-list 1.

Step 3.

Insert 87 in correct position in sorted sub-list.

Step 4.

Insert 12 in correct position in sorted sub-list.

Step 5.

Insert 32 in correct position in sorted sub-list.

Step 6.

Insert 4 in correct position in sorted sub-list.

Insertion Sort - Code

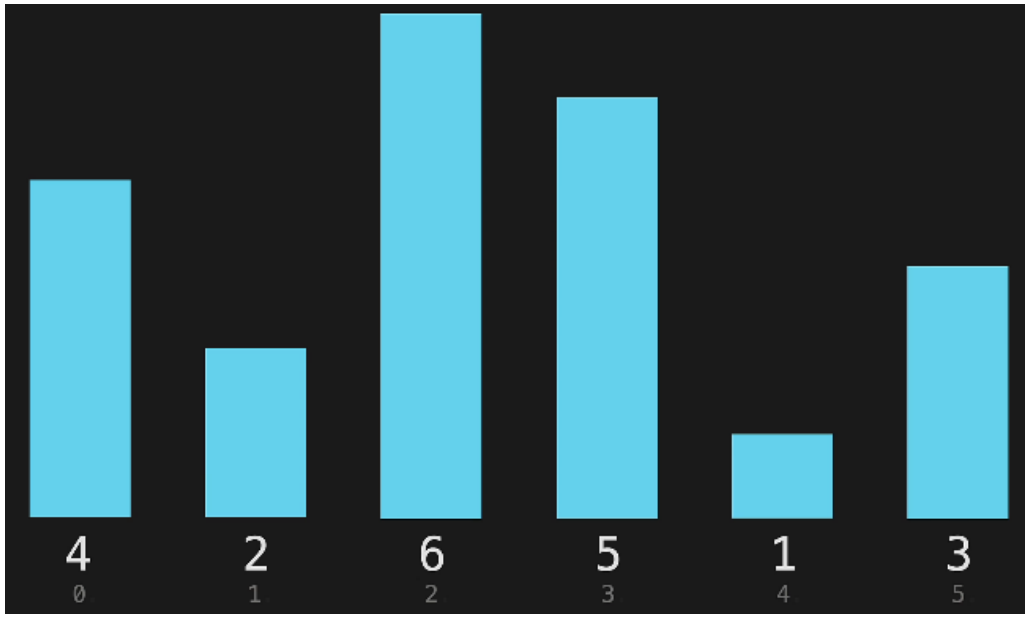
```
def insertion_sort(my_list):
    # we start at index of 1, because we consider the first element sorted
    for i in range(1, len(my_list)):
       # we hold the current value in a temporary variable and we compare
it to the item before it (at index j)
        temp = my_list[i]
        j = i-1 # j is the previous index
        while temp < my_list[j] and j > -1:
            my_list[j+1] = my_list[j] # we swap elements at indexes j and i
            my_list[j] = temp
            j -= 1 # we decrement j so we move to the left with the comparison
    return my_list
print(insertion_sort([4,2,6,5,1,3]))
```

Selection Sort

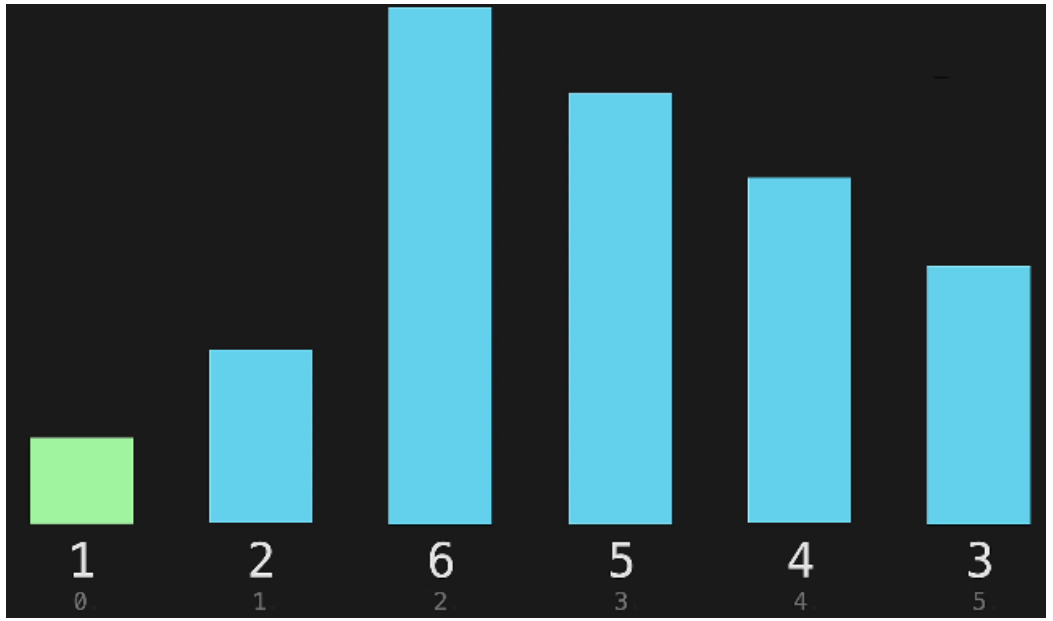
- The selection sorting algorithm begins by **finding the smallest element in the list, which it interchanges with the data stored at the first position in the list**.
- Thus, it makes a sub-list sorted up to the first element.
- Next, **the second smallest element**, which is the smallest element in the remaining list, is identified and **interchanged with the second position** in the list. This makes the initial two elements sorted.
- The process is repeated, and next, the smallest element remaining in the list should be swapped with the element in the third index on the list. This means that the first three elements are now sorted.
- This process is repeated for (n-1) times to sort n items.

Selection Sort

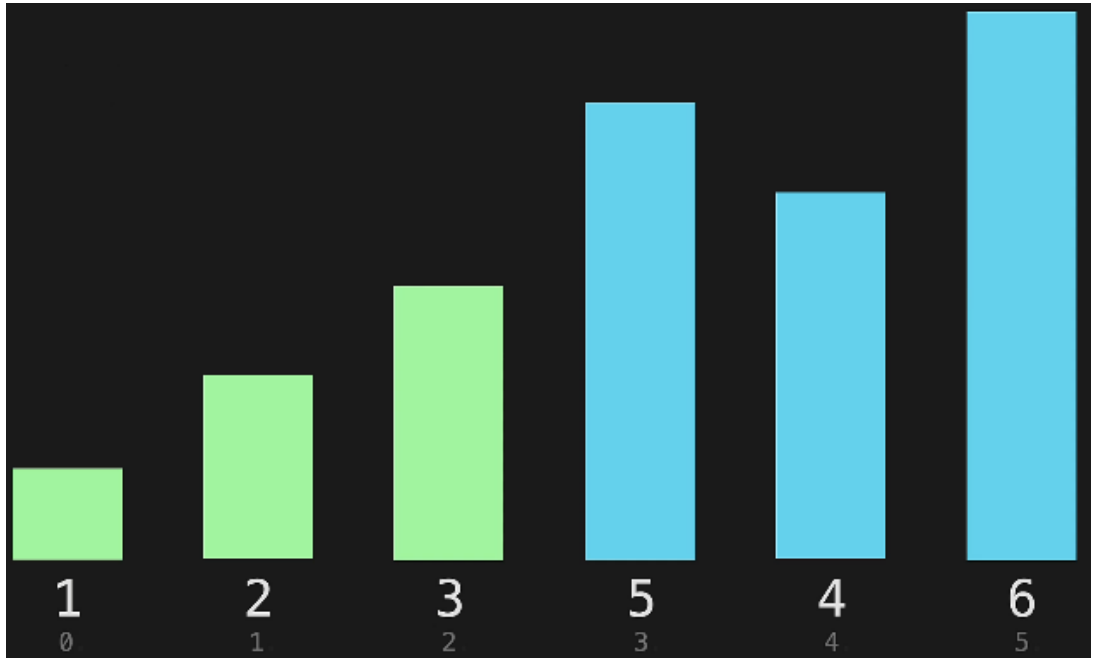
- For selection sort we need elements' indexes



- We look at the first item and **we consider it to be the smallest element in the list**. Then we keep track of the index where the minimum value is (the minimum value so far is at the index of 0: *min_index = 0*)
- We move to the next element in the list and ask ourselves if that element is lower than the minimum value at the min_index, or not (on our example it is, so we change min_index to the index of our new min value, *min_index = 1*)
- The next minimum value in the list is 1, so *min_index = 4*, and because we found the smallest element in the list **we switch 1 with 4**: 1,2,6,5,4,3
- When we do the switch, we consider the value at position 0 to be sorted.



- Now that element at position 0 is sorted, we move on to the next element, and consider that one to be the minimum (*min_index = 1*), and we compare it to all the elements in the list. In this case, no element in the list is less than 2, so it remains in the same position (no swapping).
- We move to the next item, 6, and we set *min_index = 2*. But 5 is even less, so *min_index = 3*, and so on we move to the right until we reach 3, and we set *min_index = 5*. Now that we reached the end of the list, we swap the value at index 5 with the value at index 2: **1,2,3**,5,4,6



- We repeat the process again. We start with *min_index = 3*, we compare 5 with 4 and the element at index 4 is less, so *min_index = 4.*
- 6 is not less, so min_index remains 4. We switch the two: **1,2,3,4**,5,6
- Now that 4 is sorted, we move to the next one: *min_index = 4*. 6 is not less than 5, so min_index remains 4 and we don’t need to swap here, because these are sorted. Now all elements are sorted.

Selection Sort - code

```
def selection_sort(my_list):
    for i in range(len(my_list)-1):
        # we position min_index at the first index
        min_index = i
        # we compare the item at min_index with all the other items after that, in the list
        for j in range(i+1, len(my_list)):
        # if the item at index j is less than the item at min_index, we set a new min_index
            if my_list[j] < my_list[min_index]:
                min_index = j
        # and now we swap the item at the last min_index in the loop with the item at index i
        # but we swap only if we find an item less than the initial min_index (i)
        if i != min_index:
            temp = my_list[i]
            my_list[i] = my_list[min_index]
            my_list[min_index] = temp
    return my_list
print(selection_sort([4,2,6,5,1,3]))
```

Big O – which of these is the most efficient?

- Bubble Sort
  - The bubble sort is an inefficient sorting algorithm that provides worst-case and average-case runtime complexity of $O(N^2)$, and a best-case complexity of O(N).
  - Generally, the bubble sort algorithm should not be used to sort large lists. However, on relatively small lists, it performs fairly well.
- Insertion Sort
  - Worst possible scenario – $O(N^2)$
  - Best possible scenario (when we have sorted, or almost sorted data) – O(N)
  - 1,2,4,3,5,6 (only one swap)
- Selection Sort
  - Both worst-case and best-case scenarios give $O(N^2)$

Thank you!