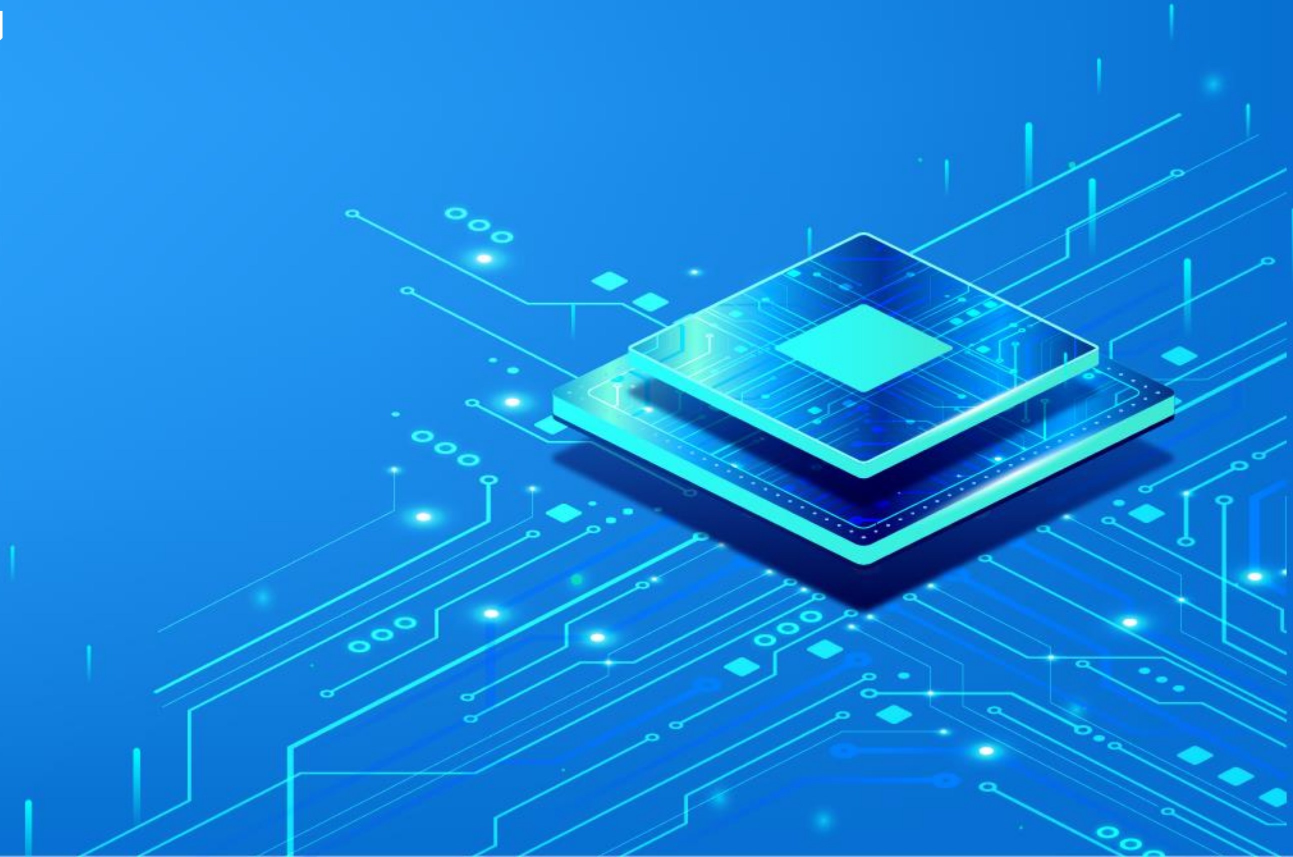
CMC UNIVERSITY

CHAPTER 3. SORTING



Outline

- ✓ Bubble sort
- ✓ Insertion sort
- ✓ Selection sort
- ✓ Quicksort
- ✓ Timsort

Sorting problem

- Sorting means reorganizing data in such a way that it is in ascending or descending order.
- Sorting is one of the most important algorithms in computer science and is widely used in database related algorithms. For several applications, if the data is sorted, it can efficiently be retrieved, for example, if it is a collection of names, telephone numbers, or items on a simple to-do list.
- In general, data can appear in many different forms. In the subject area, we convention: the set of sorted objects is a set of records, each record includes a number of data fields, corresponding to different attributes. However, not all data fields in the record are considered when sorting, but only a certain field is sorted, this field is called a *key*. Sorting will be done based on the value of this key.

Sorting problem

Example: List of telephones of state administrative agencies in Hanoi, including records for each agency. Each record consists of 3 fields: Agency name, address, and phone number. Sort key is agency name

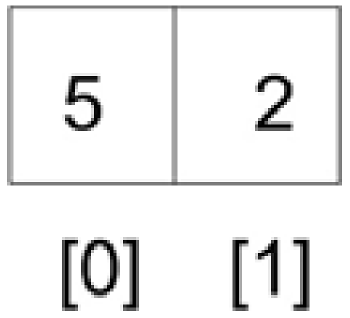
- We can compare different sorting algorithms by how much time and memory space is required to use them.
- The time taken by an algorithm changes depending on the input size. Moreover, some algorithms are relatively easy to implement, but may perform poorly with respect to time and space complexity, whereas other algorithms are slightly more complex to implement, but can perform well when sorting longer lists of data

CMC UNIVERSITY

BUBBLE SORT

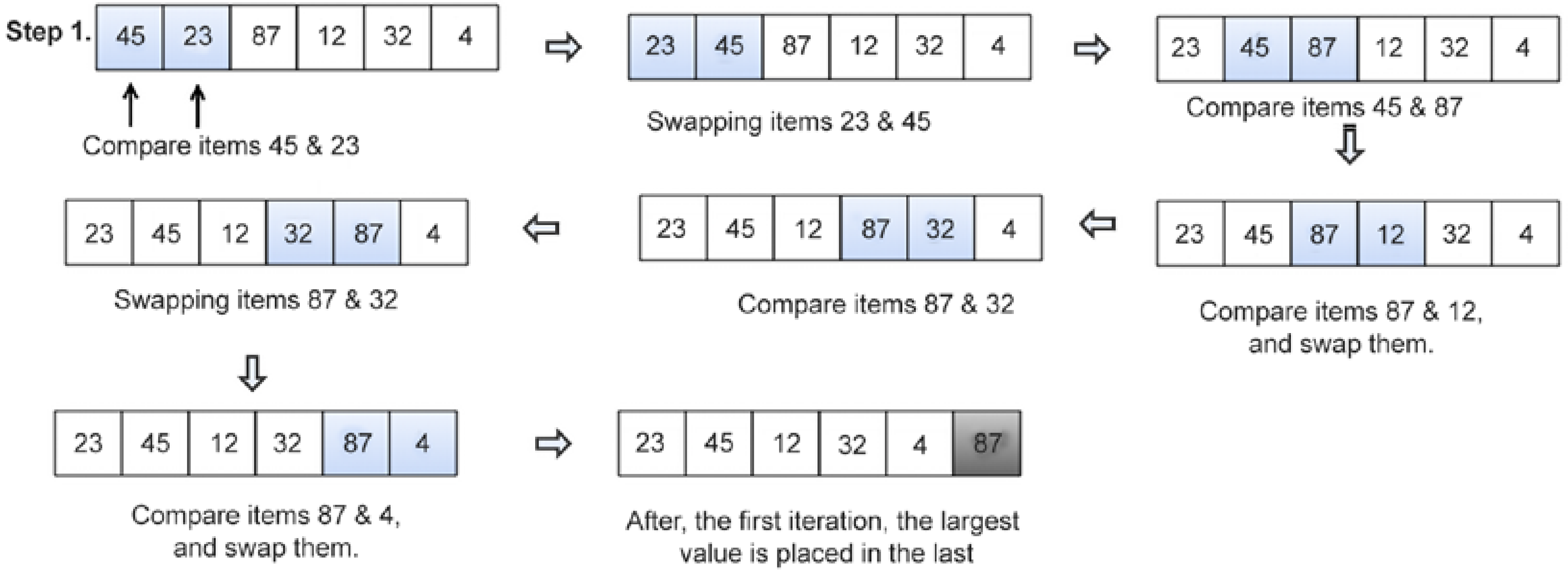
Idea of the bubble sort algorithm

- The idea behind the bubble sort algorithm is very simple. Given an unordered list, we compare adjacent elements in the list, and after each comparison, we place them in the right order according to their values. So, we swap the adjacent items if they are not in the correct order. This process is repeated *n-1* times for a list of *n* items.
- In each iteration, the largest element of the list is moved to the end of the list. After the second iteration, the second largest element will be placed at the second-to-last position in the list. The same process is repeated until the list is sorted.



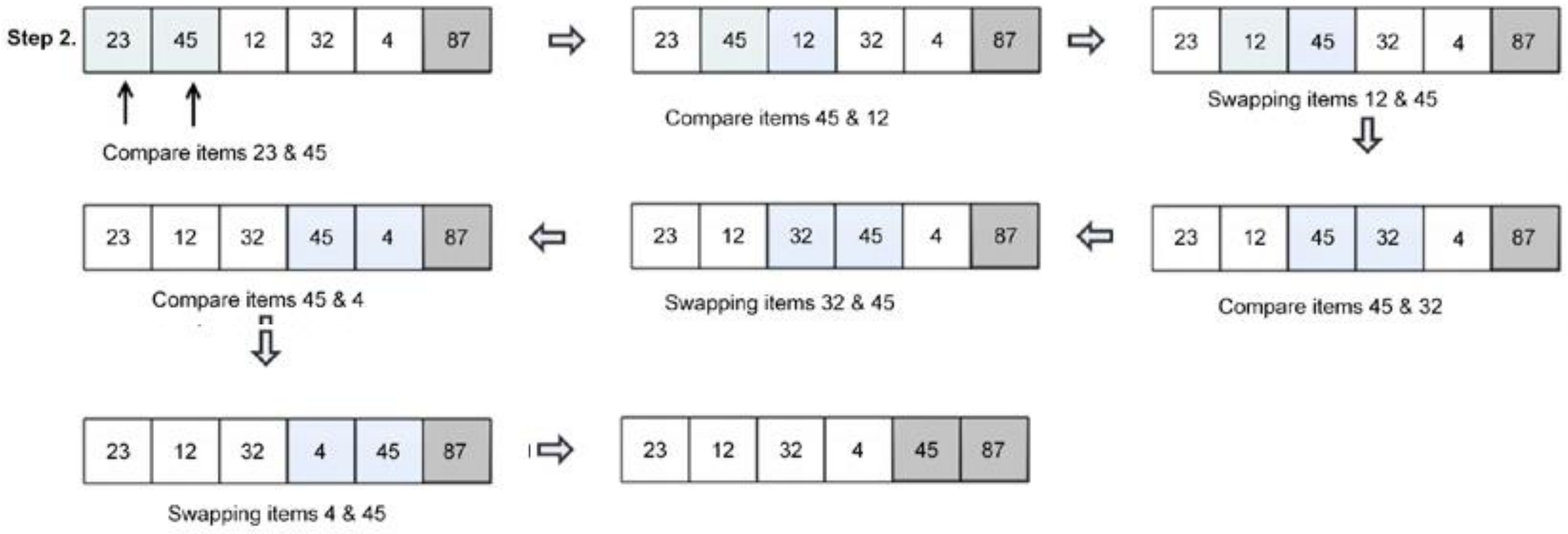
Example

- We can see that after the first iteration of the bubble sort, the largest element, 87. is placed in the last position of the list:



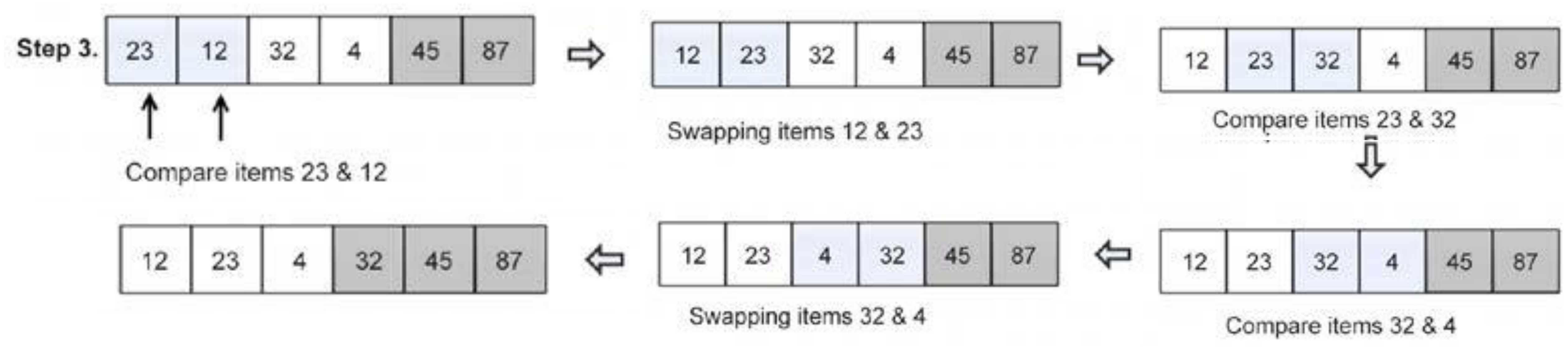
Example

- After the first iteration, we just need to arrange the remaining *(n-1)* elements; we repeat the same process by comparing the adjacent elements for the remaining five elements. After the second iteration, the second largest element, *45*, is placed at the second-to-last position in the list:



Example

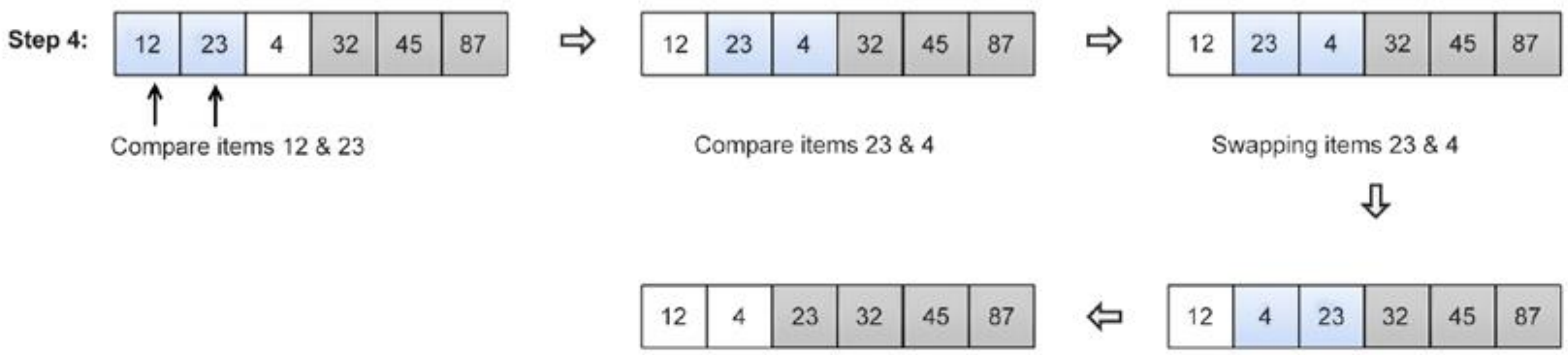
- Next, we have to compare the remaining *(n-2)* elements to arrange them:



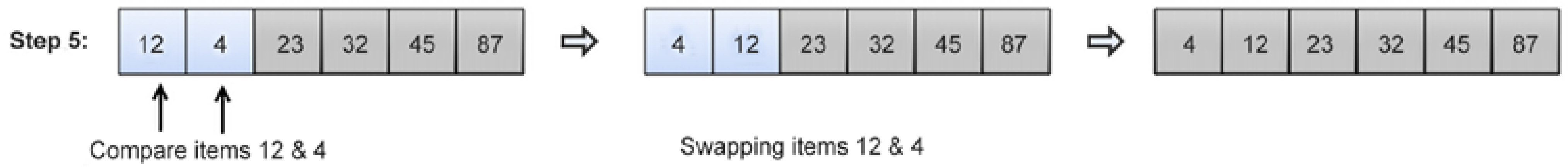
CMC UNIVERSITY

Example

Similarly, we compare the remaining elements to sort them:



Finally, for the last two remaining elements, we place them in the correct order to obtain the final sorted list:



CMC UNIVERSITY

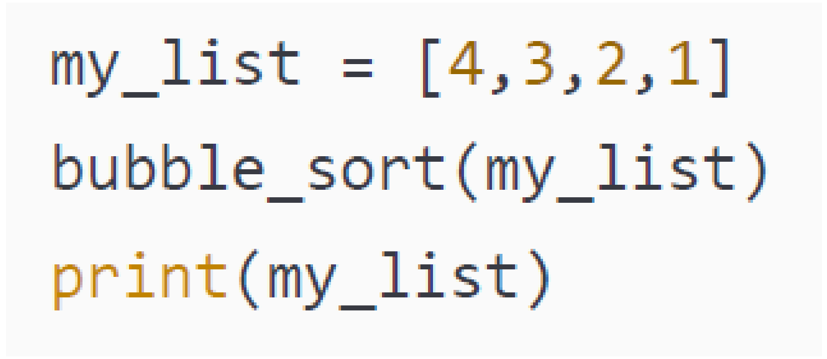
Implementation

The complete Python code of the bubble sort algorithm is shown below, and afterward, each step is explained in detail:

```
def bubble_sort(unordered_list):
    iteration_number = len(unordered_list)-1
    for i in range(iteration_number,0,-1):
        for j in range(i):
            if unordered_list[j] > unordered_list[j+1]:
                temp = unordered_list[j]
                unordered_list[j] = unordered_list[j+1]
                unordered_list[j+1] = temp
```

Implementation

The following code snippet can be used to deploy the bubble sort algorithm:



```
my_list = [4,3,2,1]
bubble_sort(my_list)
print(my_list)
```

In the worst case, the number of comparisons required in the first iteration will be (n-1), in the second, the number of comparisons will be (n-2), and in the third iteration it will be (n-3), and so on. Therefore, the total number of comparisons required in the bubble sort will be as follows: $(n-1) + (n-2) + (n-3) + \ldots + 1 = n(n-1)/2 \rightarrow O(n^2)$

CMC UNIVERSITY

Evaluation

- The bubble sort algorithm is not an efficient sorting algorithm:
  - A worst-case runtime complexity of $O(n^2)$, when we want to sort the given list in ascending order and the given list is in descending order,
  - A best-case complexity of $O(n)$, when the given list is already sorted (there will not be any need for swapping)
- Generally, the bubble sort algorithm should not be used to sort large lists. The bubble sort algorithm is suitable for applications where performance is not important or the length of the given list is short, and moreover, short and simple code is preferred. The bubble sort algorithm performs well on relatively small lists.

CMC UNIVERSITY

INSERTION SORT

The idea of insertion sort

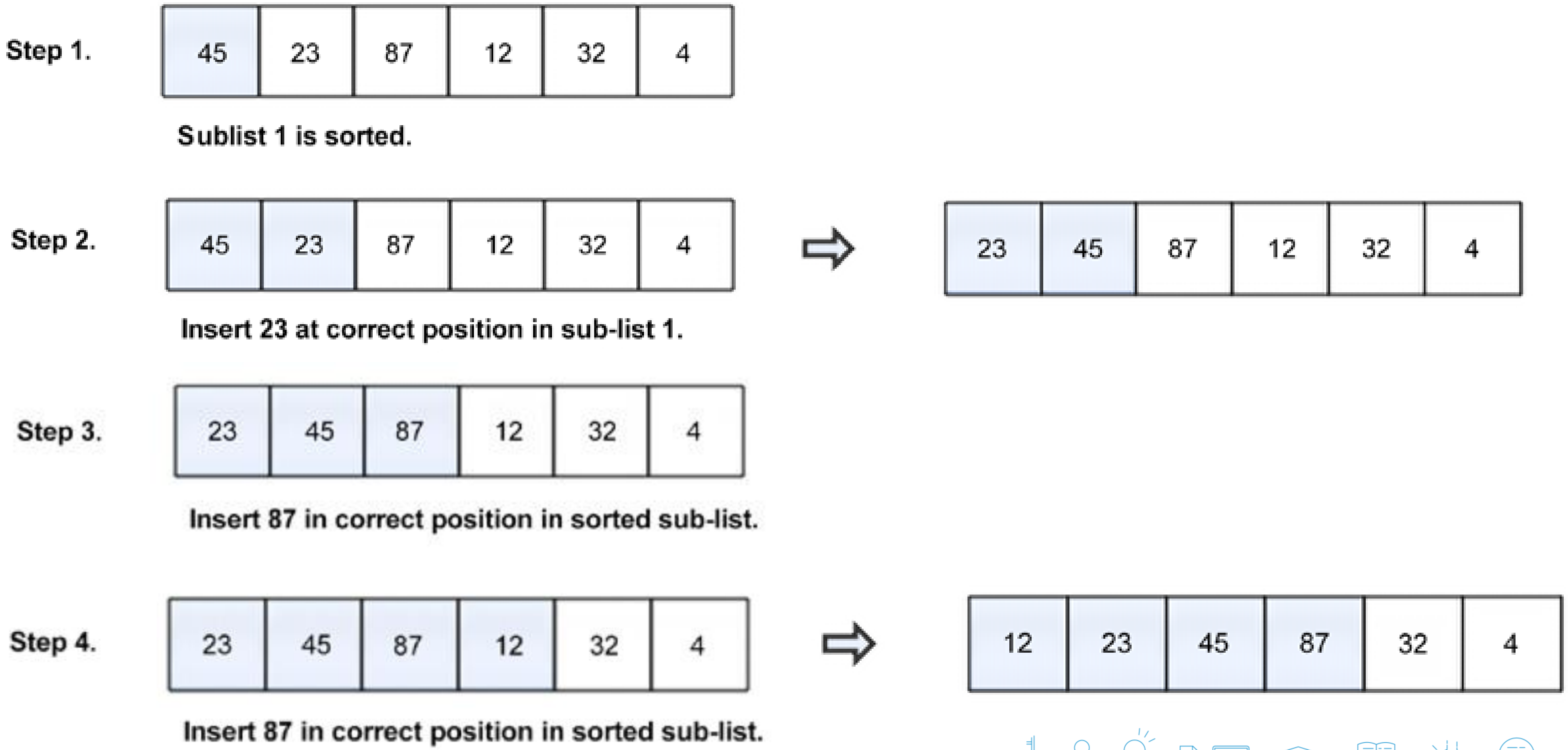
- The idea of insertion sort is that we maintain two sublists (a sublist is a part of the original larger list), one that is sorted and one that is not sorted. We take elements from the unsorted sublist and insert them in the correct position in the sorted sublist, in such a way that this sublist remains sorted.
- The insertion sort algorithm:
  - Start with one element, taking it to be sorted,
  - Take elements one by one from the unsorted sublist and place them at the correct positions (in relation to the first element) in the sorted sublist

Example

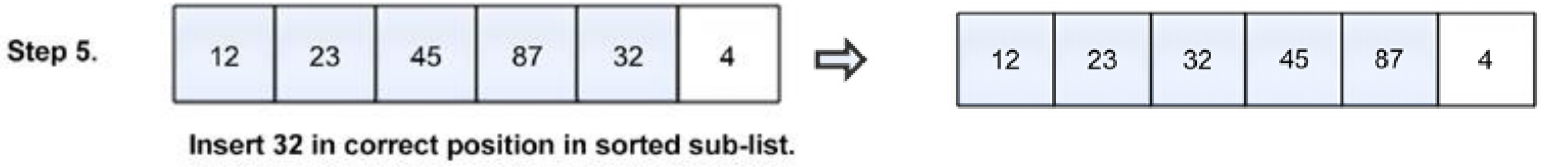
Sort a list: {45, 23, 87, 12, 32, 4}

- Start with one element 45, assuming it to be sorted,
- Then take the next element, 23, from the unsorted sublist and insert it at the correct position in the sorted sublist: 23, 45
- In the next iteration, we take the third element, 87, from the unsorted sublist, and again insert it into the sorted sublist at the correct position: 23, 45, 87
- Take the next element, 12, from the unsorted sublist and insert it at the correct position in the sorted sublist: 12, 23, 45, 87
- Take the next element, 32, from the unsorted sublist and insert it at the correct position in the sorted sublist: 12, 23, 32, 45, 87
- Take the end element, 4, from the unsorted sublist and insert it at the correct position in the sorted sublist: 4, 12, 23, 32, 45, 87

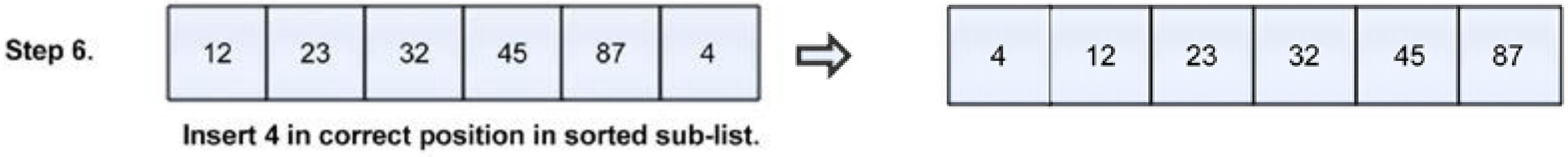
Example


Step 1.
45 23 87 12 32 4
Sublist 1 is sorted.
Step 2.
45 23 87 12 32 4
23 45 87 12 32 4
Insert 23 at correct position in sub-list 1.
Step 3.
23 45 87 12 32 4
Insert 87 in correct position in sorted sub-list.
Step 4.
23 45 87 12 32 4
12 23 45 87 32 4
Insert 87 in correct position in sorted sub-list.

Example



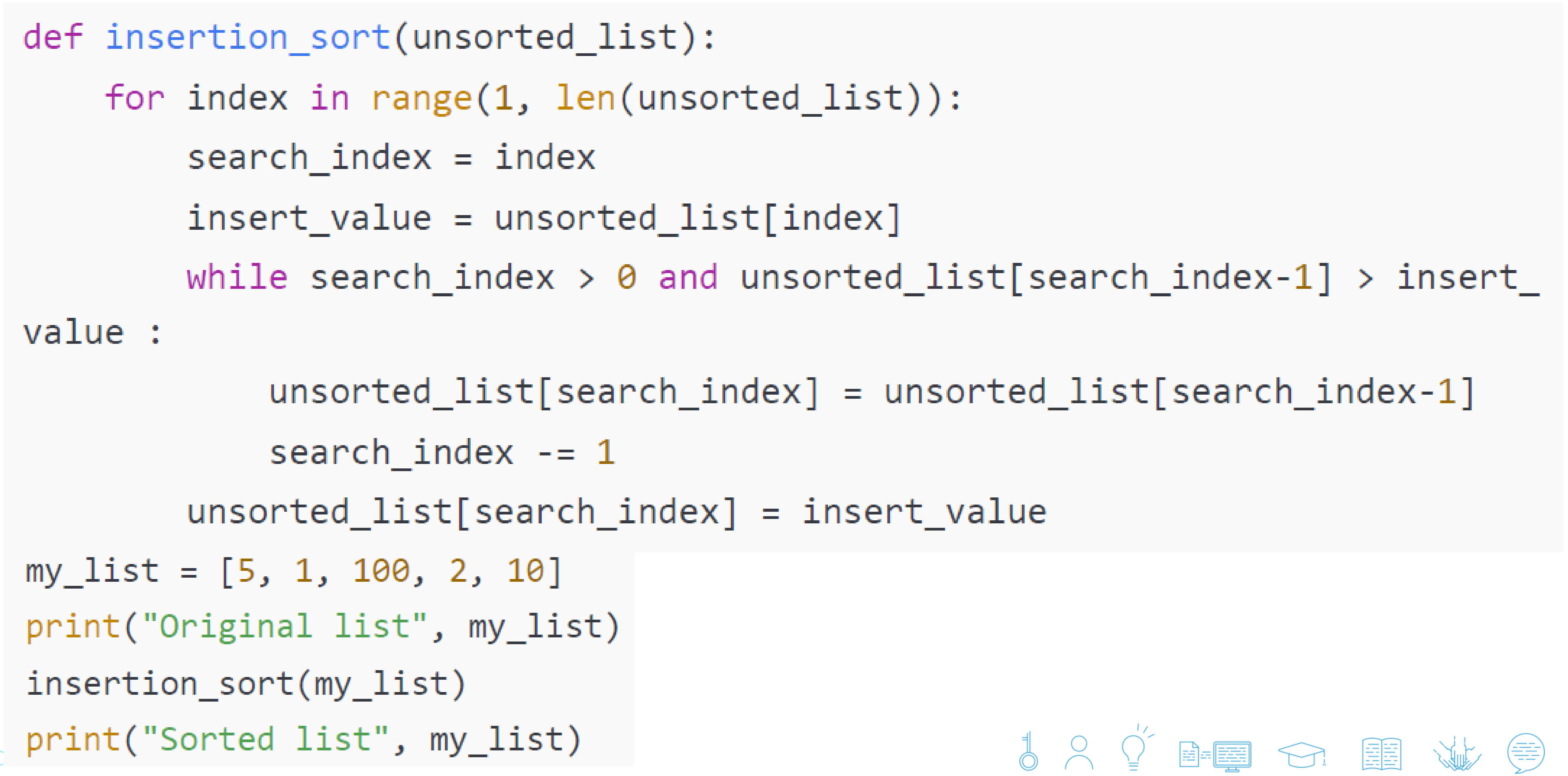
Insert 32 in correct position in sorted sub-list.



Insert 4 in correct position in sorted sub-list.

Implementation

The complete Python code for insertion sort is given below



```
def insertion_sort(unsorted_list):
    for index in range(1, len(unsorted_list)):
        search_index = index
        insert_value = unsorted_list[index]
        while search_index > 0 and unsorted_list[search_index-1] > insert_
value :
            unsorted_list[search_index] = unsorted_list[search_index-1]
            search_index -= 1
        unsorted_list[search_index] = insert_value
my_list = [5, 1, 100, 2, 10]
print("Original list", my_list)
insertion_sort(my_list)
print("Sorted list", my_list)
```

CMC UNIVERSITY

Evaluation

- So, we will need one comparison in the first iteration, two comparisons in the second iteration, and three comparisons in the third iteration, and (n-1) comparisons in the (n-1)th iteration. Thus, the total number of comparisons are:

$$1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2$$

- The worst-case time complexity of insertion sort is $O(n^2)$, when the given list of elements is sorted in reverse order. In that case, each element will have to be compared with each of the other elements.
- The runtime complexity of The best-case complexity of the insertion sort algorithm is $O(n)$, in the situation when the given input list is already sorted in which each element from the unsorted sublist is compared to only the right-most element of the sorted sublist in each iteration.

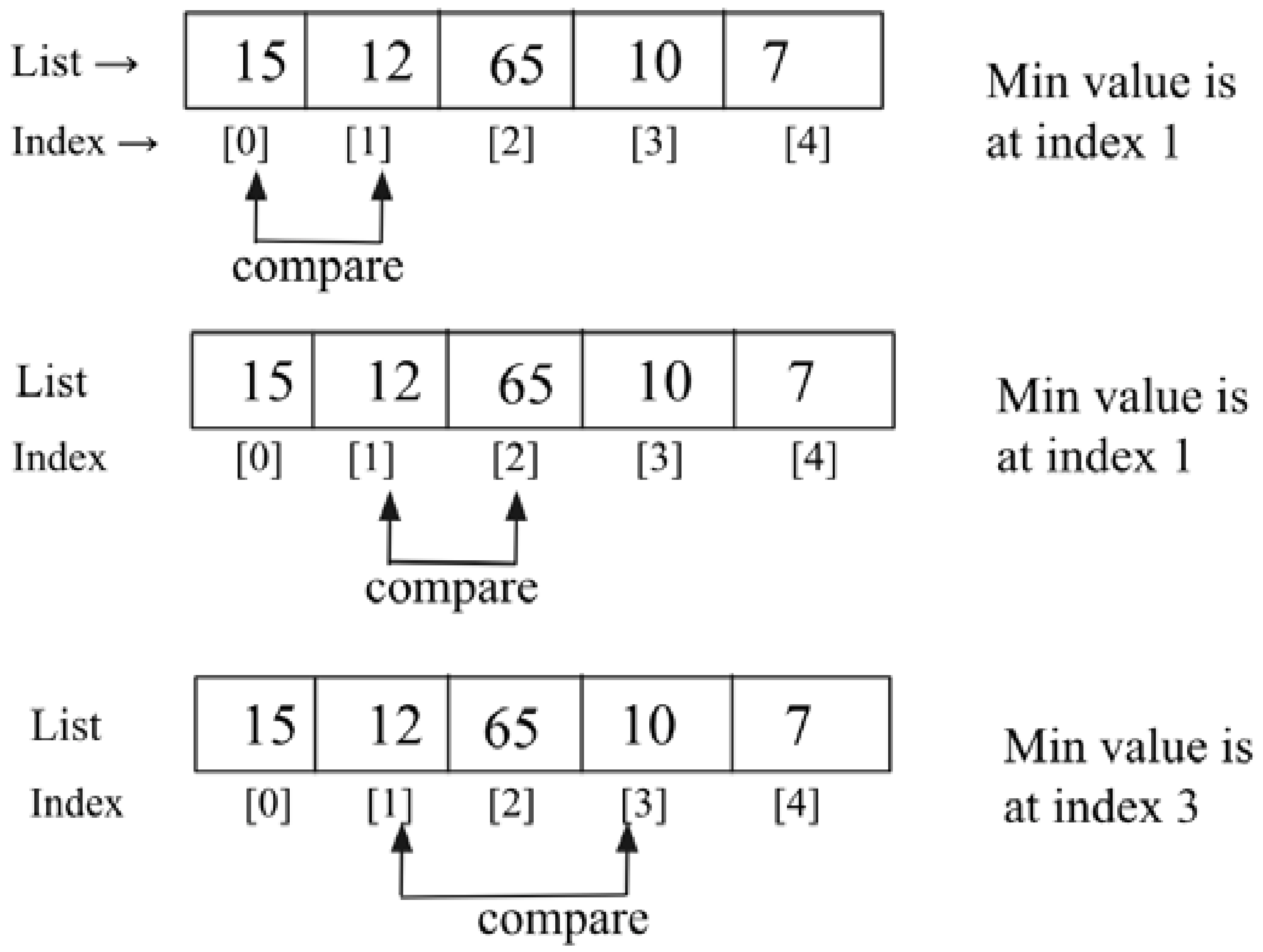
The insertion sort algorithm is good to use when the given list has a small number of elements, and it is best suited when the input data arrives one by one, and we need to keep the list sorted. Now we are going to take a look at the selection sort algorithm.

SELECTION SORT

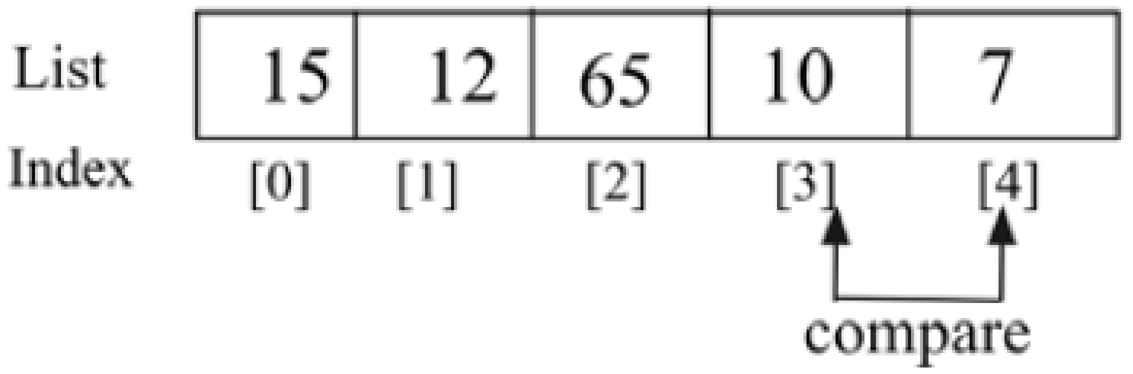
Idea of the selection sort algorithm

- The selection sort algorithm begins by finding the smallest element in the list and interchanges it with the data stored at the first position in the list. Thus, it sorts the sublist sorted up to the first element. This process is repeated for (*n*-*1*) times to sort n items.
- 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 the smallest element remaining in the list is swapped with the element in the third index on the list. This means that the first three elements are now sorted.

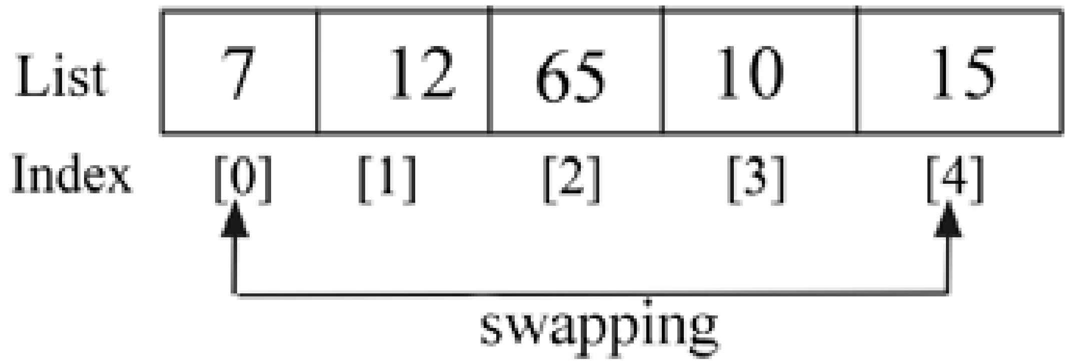
Example


List →
Index →
15
12
65
10
7
[0]
[1]
[2]
[3]
[4]
compare
Min value is at index 1
List
Index
15
12
65
10
7
[0]
[1]
[2]
[3]
[4]
compare
Min value is at index 1
List
Index
15
12
65
10
7
[0]
[1]
[2]
[3]
[4]
compare
Min value is at index 3

Example

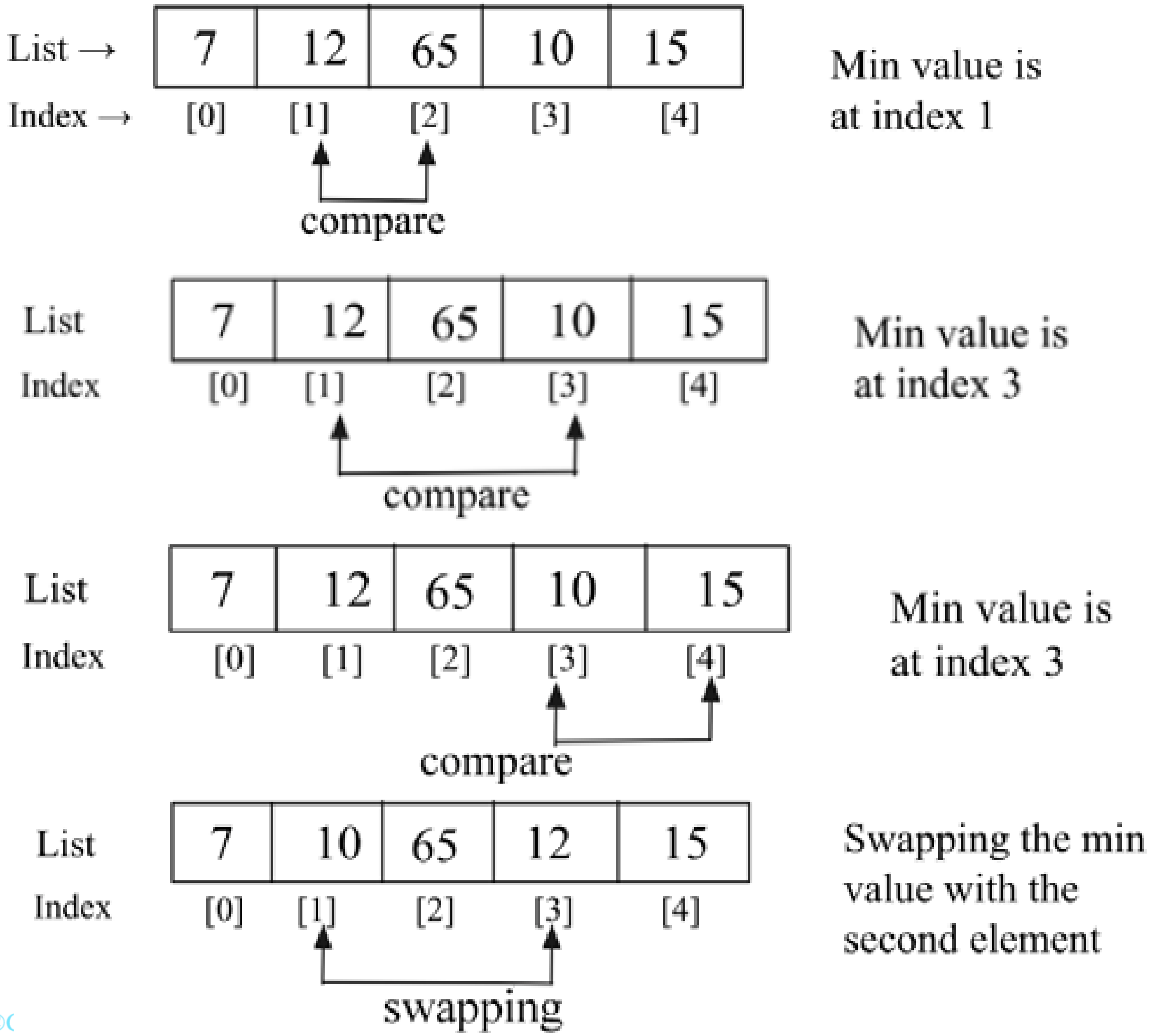


Min value is at index 4



Swapping the min value with the first element

Example


List →
Index →
7
12
65
10
15
[0]
[1]
[2]
[3]
[4]
compare
Min value is at index 1
List
Index
7
12
65
10
15
[0]
[1]
[2]
[3]
[4]
compare
Min value is at index 3
List
Index
7
12
65
10
15
[0]
[1]
[2]
[3]
[4]
compare
Min value is at index 3
List
Index
7
10
65
12
15
[0]
[1]
[2]
[3]
[4]
swapping
Swapping the min value with the second element

CMC UNIVERSITY

Implementation of the algorithm

```
def selection_sort(unsorted_list):
    size_of_list = len(unsorted_list)
    for i in range(size_of_list):
        small = i
        for j in range(i+1, size_of_list):
            if unsorted_list[j] < unsorted_list[small]:
                small = j
        temp = unsorted_list[i]
        unsorted_list[i] = unsorted_list[small]
        unsorted_list[small] = temp
```

```
a_list = [3, 2, 35, 4, 32, 94, 5, 7]
print("List before sorting", a_list)
selection_sort(a_list)
print("List after sorting", a_list)
```

CMC UNIVERSITY

Evaluation

- In the selection sort, $(n-1)$ comparisons are required in the first iteration, and $(n-2)$ comparisons are required in the second iteration, and $(n-3)$ comparisons are required in the third iteration, and so on. So, the total number of comparisons required is: $(n-1) + (n-2) + (n-3) + \ldots\ldots + 1 = n(n-1)/2$.
- Thus, the worst-case time complexity of the selection sort is $O(n^2)$. The worst-case situation is when the given list of elements is reverse ordered.
- The selection sorting algorithm gives the best-case runtime complexity of $O(n^2)$.
- The selection sorting algorithm can be used when we have a small list of elements.

CMC UNIVERSITY

2.4 QUICK SORT

Divide and Conquer

- Divide: - If the sequence is too small (1 or two elements) then sorting is easy

  - If the sequence is big, divide it to two parts and solve each part separately
- Conquer: Recursively solve the sub problems associated with the subsets
- Combine: Take the solutions to the sub problems and merge them into a solution to the original problem

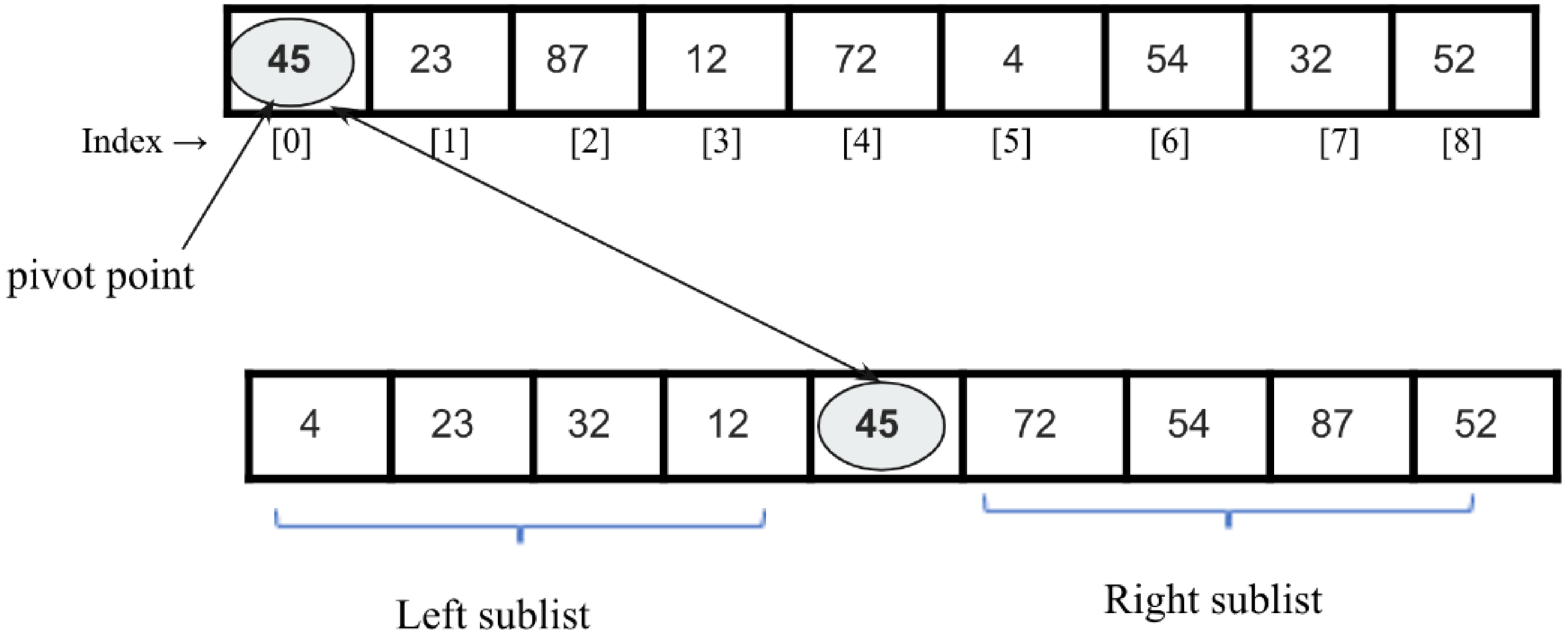
The idea of the algorithm

- Quicksort is an efficient sorting algorithm. The quicksort algorithm is based on the divide-and - conquer class of algorithms, similar to the merge sort algorithm, where we break (divide) a problem into smaller chunks that are much simpler to solve, and further, the final results are obtained by combining the outputs of smaller problems (conquer).
- The concept behind quick sorting is partitioning a given list or array.
  - We first select a data element from the given list, which is called a pivot element. For the sake of simplicity, we'll take the first element in the array
  - Next, all the elements in the list are compared with this pivot element. The elements which are less than the pivot element are arranged to the left of the pivot, the elements that are greater than the pivot element are arranged to the right of the pivot. After the first iteration, the chosen pivot point is placed in the list at its correct position, and, we obtain two unordered sublists
  - Follow the same process again on these two sublists

The idea of the algorithm

Example: Sort a list of numbers {45, 23, 87, 12, 72, 4, 54, 32, 52}

The following figure illustrates the initial list partitioned into two sublists:



The quicksort algorithm

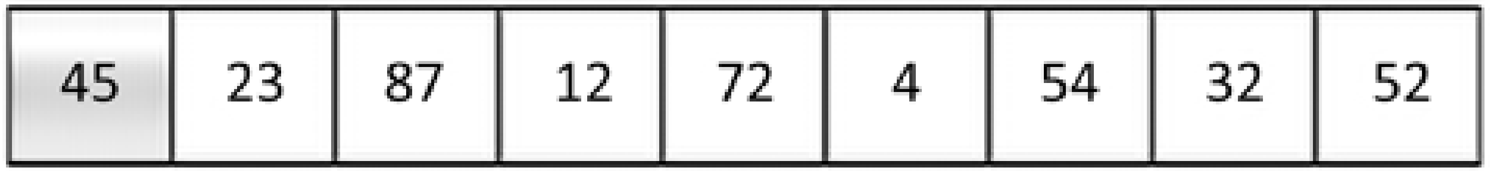
1. We start by choosing a pivot element with which all the data elements are to be compared, and at the end of the first iteration, this pivot element will be placed in its correct position in the list. In order to place the pivot element in its correct position, we use two pointers, a left pointer, and a right pointer.
    - Start with the *left pointer*, moving in a left-to-right direction until we reach a position where the data item in the list has a greater value than the pivot element
    - Move the *right pointer* toward the left until we find a data item less than the pivot element
    - Swap these two values indicated by the left and right pointers
    - Repeat the same process until both pointers cross each other, in other words, until the right pointer index indicates a value less than that of the left pointer index

The quicksort algorithm

2. After each iteration described in step 1, the pivot element will be placed at its correct position in the list, and the original list will be divided into two unordered sublists, left and right. We follow the same process (as described in step 1) for both these left and right sublists until each of the sublists contains a single element.

3. Finally, all the elements will be placed at their correct positions, which will give the sorted list as an output.

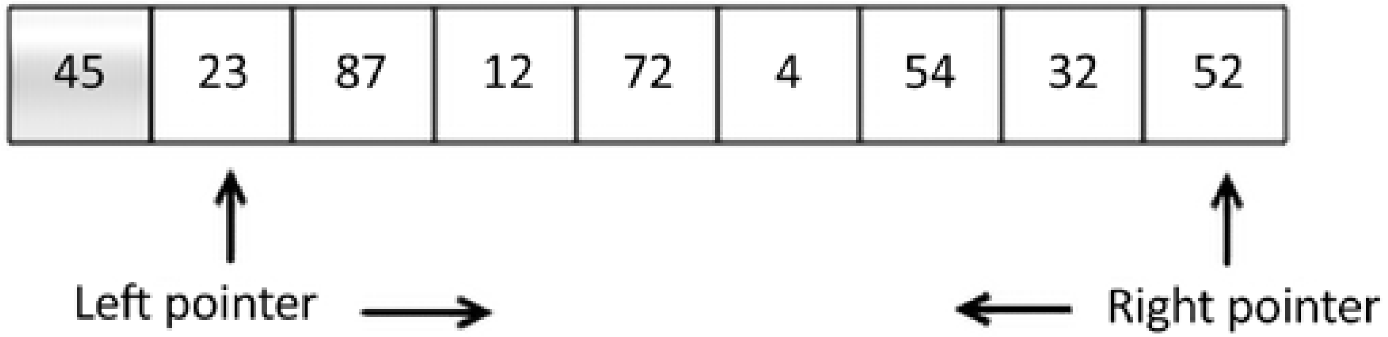
Example: Sort a list of numbers {45, 23, 87, 12, 72, 4, 54, 32, 52}



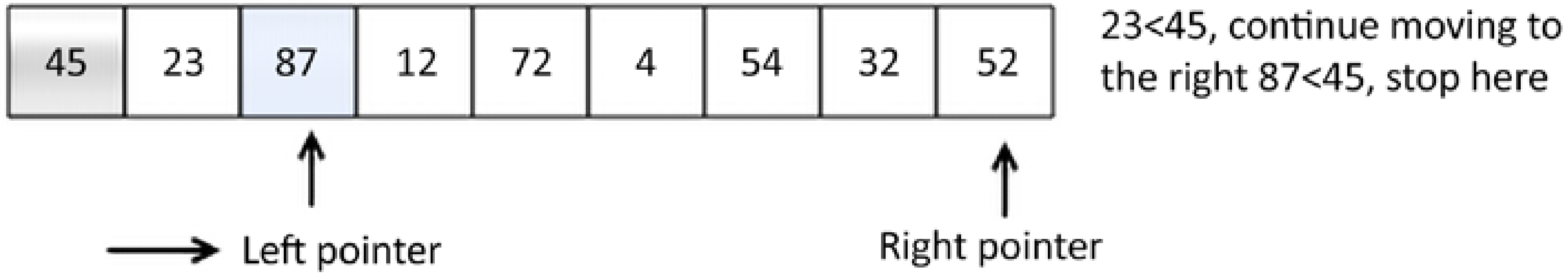
45	23	87	12	72	4	54	32	52

Example

- Assume that the pivot element (also called the pivot point) in our list is the first element, 45

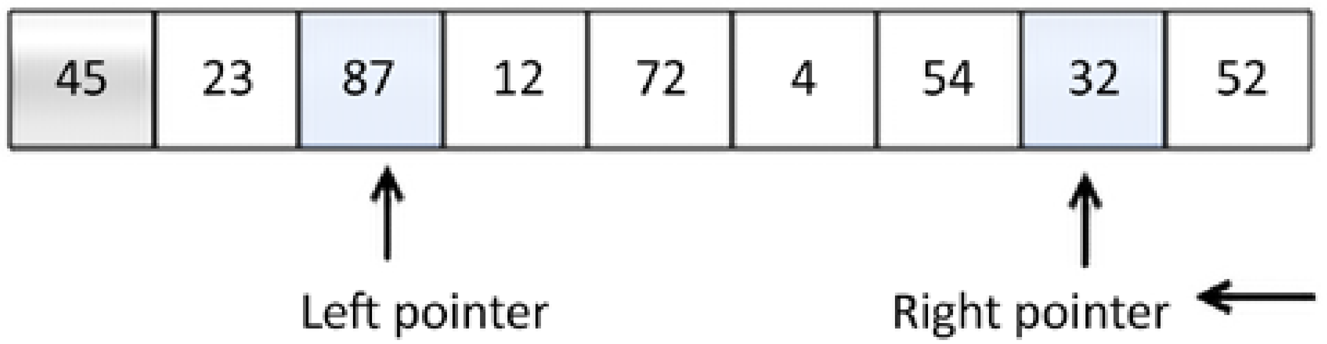


- Move the left pointer from index 1 in a rightward direction, and stop when we reach the value 87, because (87 > 45)



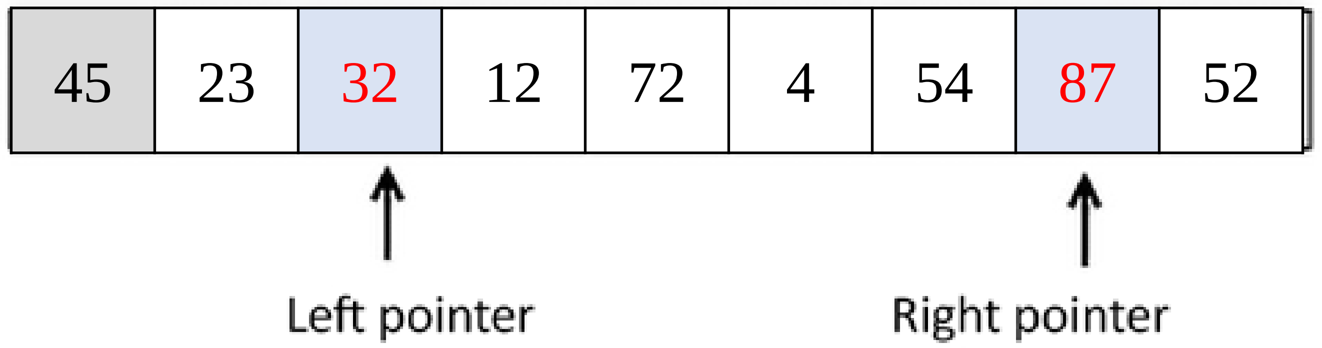
- Move the right pointer toward the left and stop when we find the value 32, because (32 < 45). Now, we swap these two values

Example



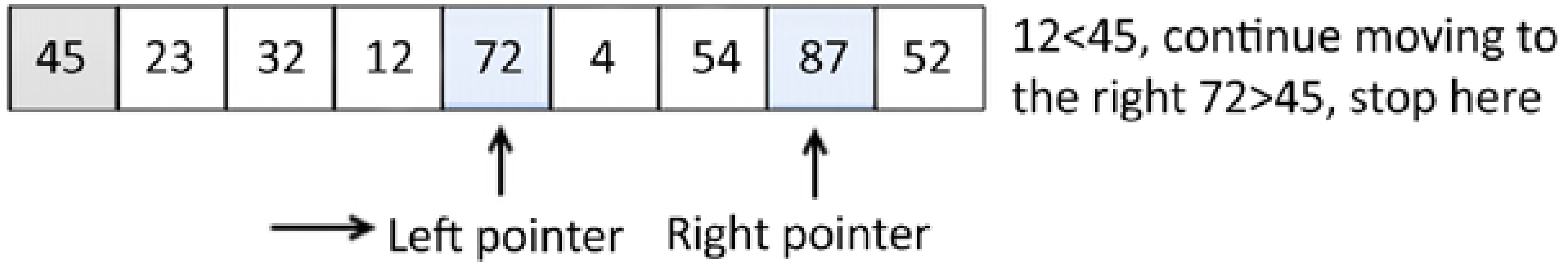
52>45, continue moving to the left
32<45, stop here

Swap 87 and 32

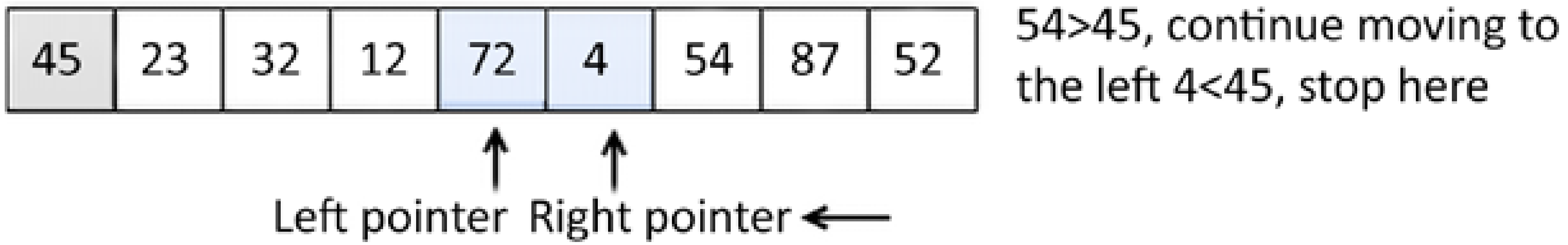


Example

- Repeat the same process and move the left pointer toward the right, and stop when we find the value 72, because (72 > 45)



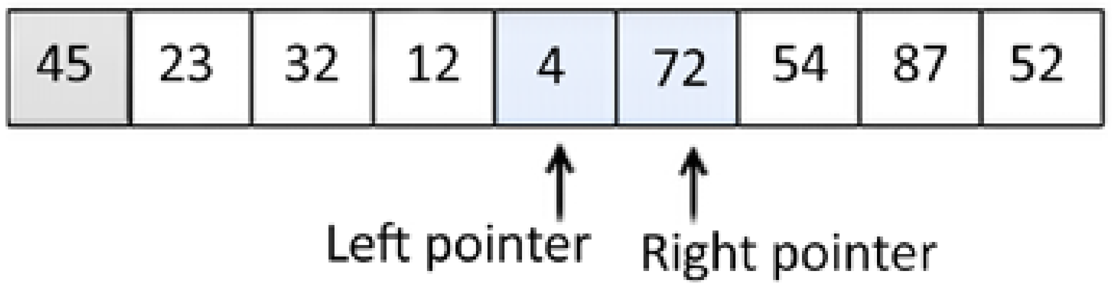
- Move the right pointer toward the left and stop when we reach the value 4, because (4 < 45)

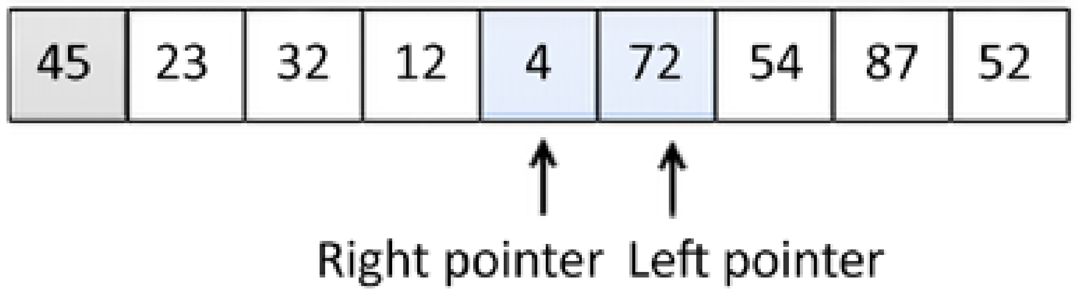


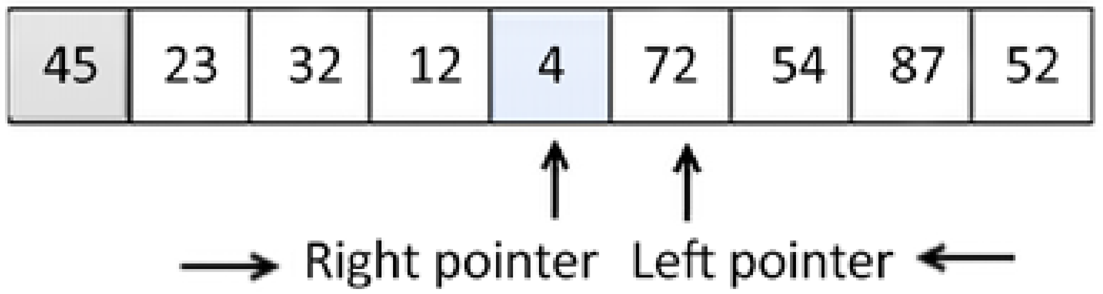
- Swap these two values, because they are on the wrong sides of the pivot value

Example

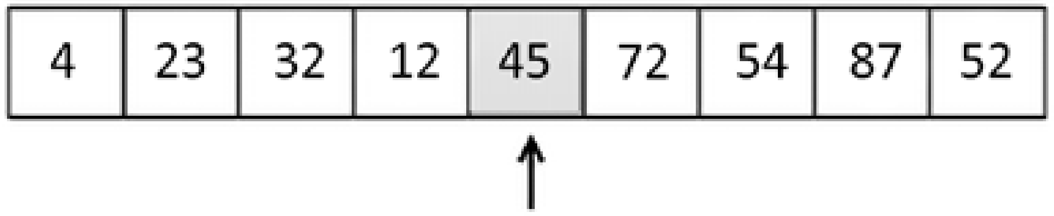
Swap 72 and 4







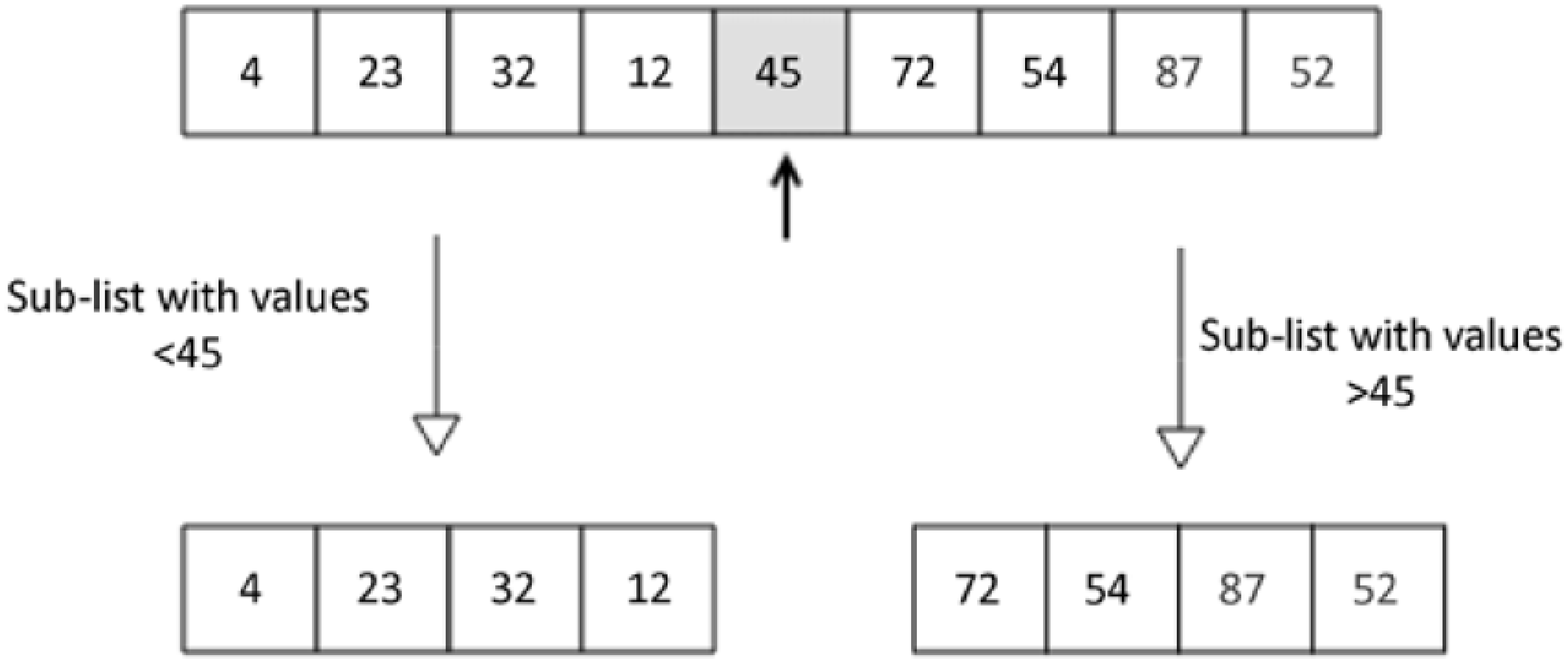
72>45, stop here
4<45, stop here
Swap 45 and 4.



CMC UNIVERSITY

Example

Now we have two sublists:



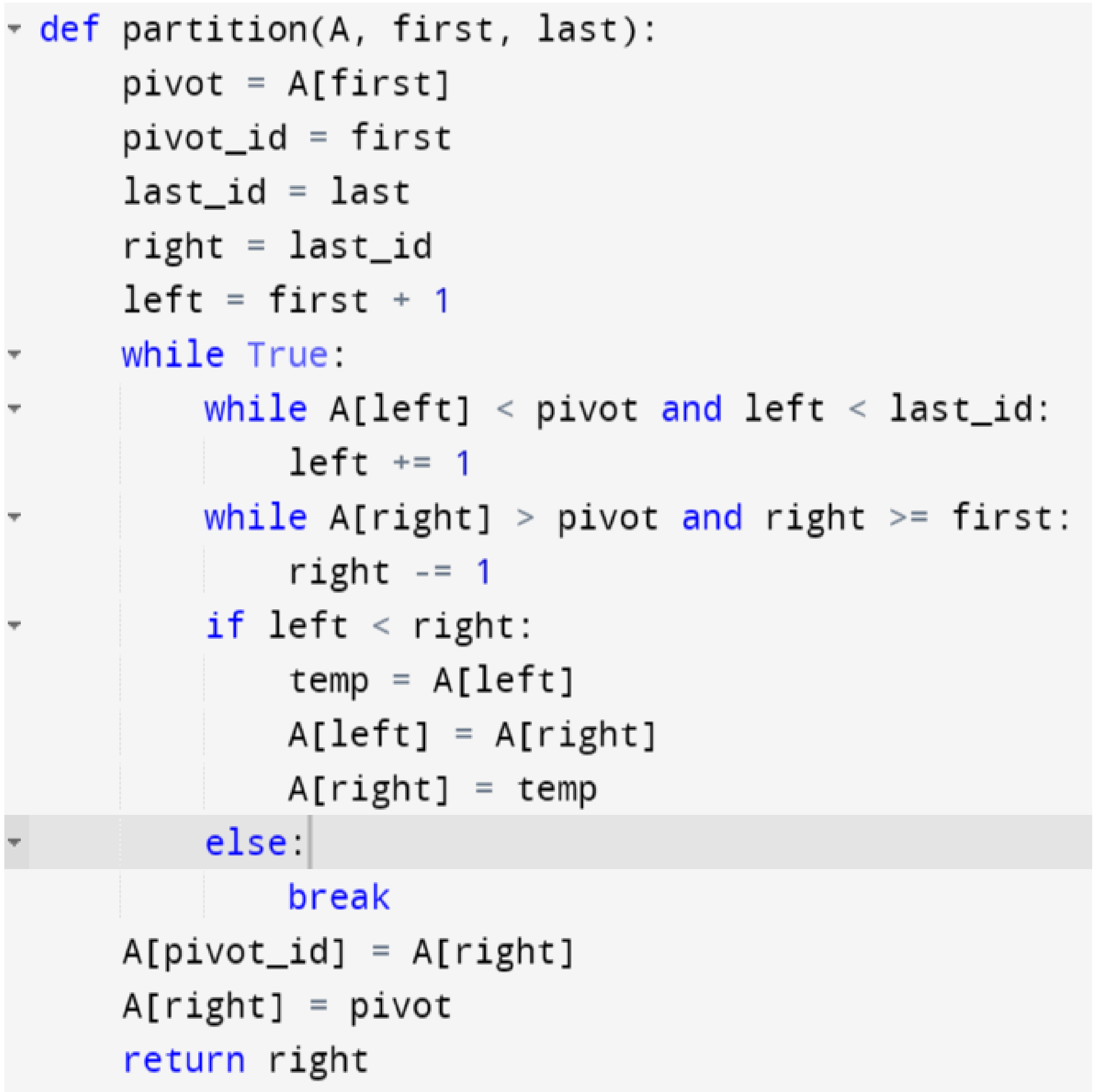
We will apply the quicksort algorithm recursively on these two sublists, and repeat it until the whole list is sorted

Implementation of quicksort

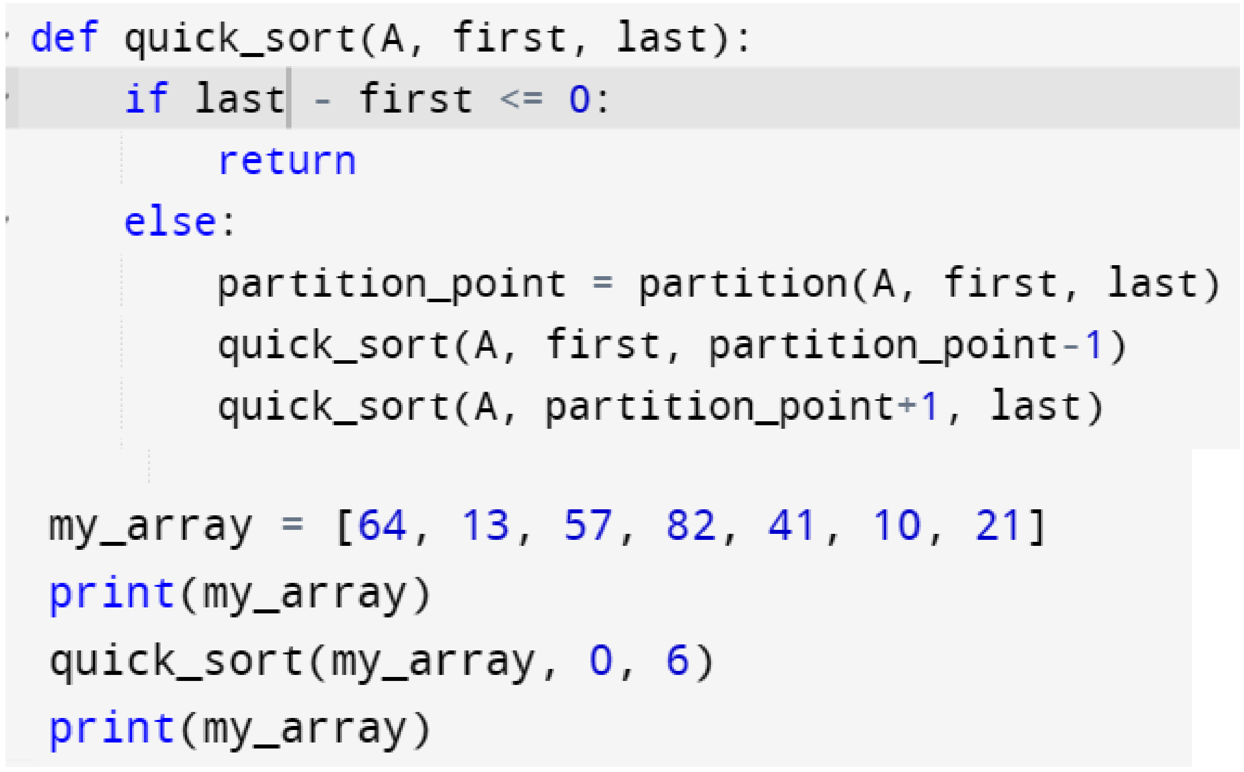
We use the following notations:

- *A* is the vector representing the given sequence of elements
- *first* is the index of the first element of the sequence, *last* is the index of the last element of the sequence
- *right* is the index corresponding to the pivot after separating the array of elements into 2 partitions

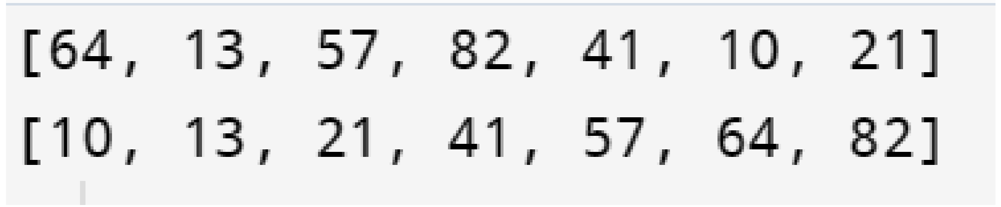
Implementation of quicksort



```python
def partition(A, first, last):
    pivot = A[first]
    pivot_id = first
    last_id = last
    right = last_id
    left = first + 1
    while True:
        while A[left] < pivot and left < last_id:
            left += 1
        while A[right] > pivot and right >= first:
            right -= 1
        if left < right:
            temp = A[left]
            A[left] = A[right]
            A[right] = temp
        else:
            break
    A[pivot_id] = A[right]
    A[right] = pivot
    return right
```



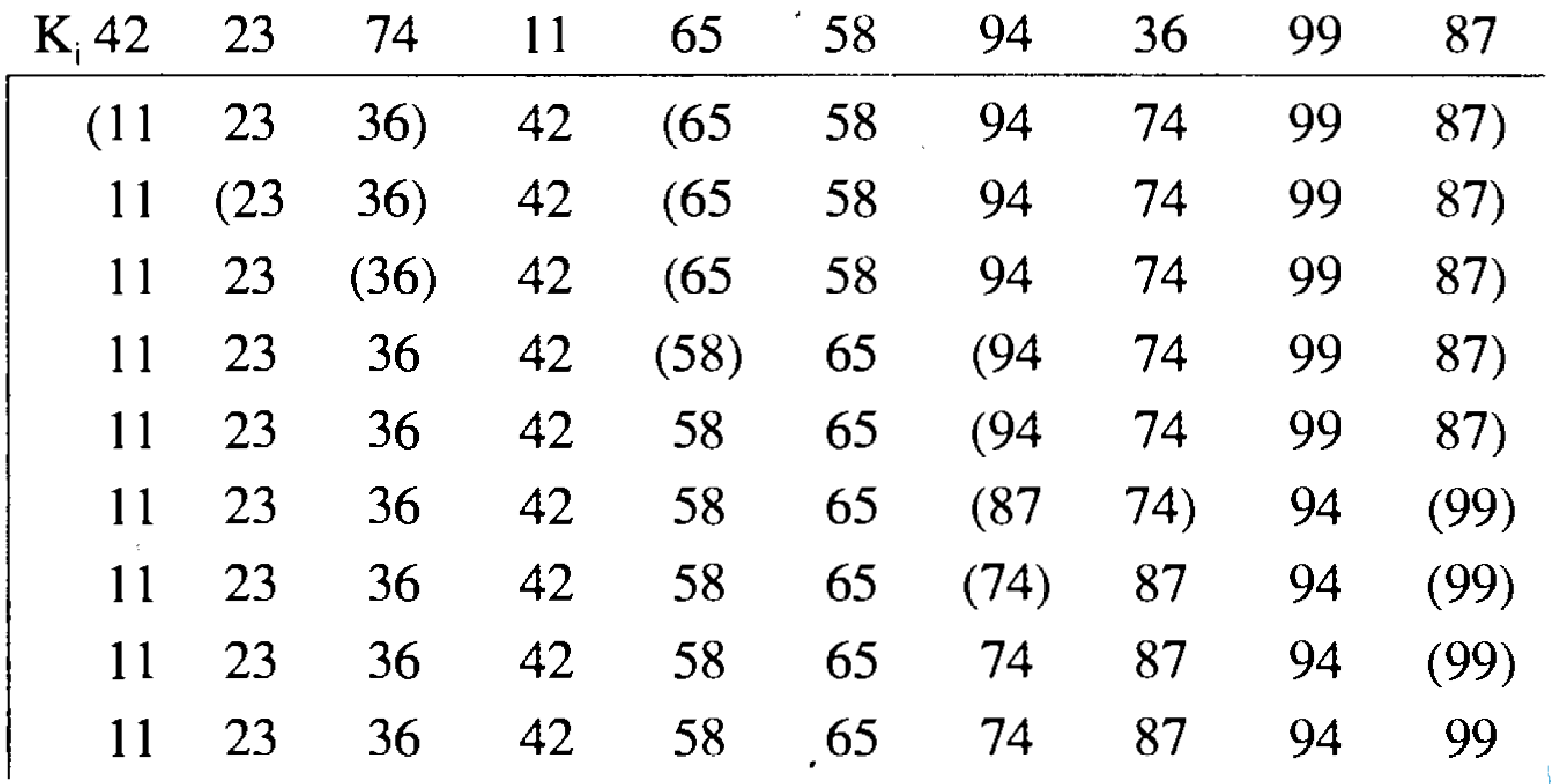
```python
def quick_sort(A, first, last):
    if last - first <= 0:
        return
    else:
        partition_point = partition(A, first, last)
        quick_sort(A, first, partition_point-1)
        quick_sort(A, partition_point+1, last)

my_array = [64, 13, 57, 82, 41, 10, 21]
print(my_array)
quick_sort(my_array, 0, 6)
print(my_array)
```



```
[64, 13, 57, 82, 41, 10, 21]
[10, 13, 21, 41, 57, 64, 82]
```

Implementation of quicksort

- Example: Illustrate the quick sort algorithm on the following sequence of numbers: {42, 23, 74, 11, 65, 58, 94, 36, 99, 87}



K_i 42	23	74	11	65	58	94	36	99	87
(11	23	36)	42	(65	58	94	74	99	87)
11	(23	36)	42	(65	58	94	74	99	87)
11	23	(36)	42	(65	58	94	74	99	87)
11	23	36	42	(58)	65	(94	74	99	87)
11	23	36	42	58	65	(94	74	99	87)
11	23	36	42	58	65	(87	74)	94	(99)
11	23	36	42	58	65	(74)	87	94	(99)
11	23	36	42	58	65	74	87	94	(99)
11	23	36	42	58	65	74	87	94	99

Review and evaluation

Selecting the pivot

- The quicksort algorithm gives a worst-case: when it selects the worst pivot point every time, and one of the partitions always has a single element.

For example, if the list is already sorted, the partition picks the smallest (or largest) element as a pivot point.

- If the *median* of an element sequence is called the element located in the middle of the sequence after the sequence has been sorted. The best-case is to choose the right median as the pivot.
- How can the right median be chosen?
  - If the occurrence of elements in the sequence is equally likely, then the median can be any one element in the sequence. In the above algorithm, we choose the first element as the pivot based on this basis.
  - If the element sequence tends to be in sorted order, the element should be chosen in the middle of the sequence

Review and evaluation

Algorithm complexity

- The partition algorithm takes time: $P(n) = c.n$
- The quicksort algorithm takes time: $T(n) = P(n) + T(right - first) + T(last - right)$
- A worst-case runtime complexity: when the list is already sorted, after partition, one of the two sublists is empty (*right* = *first* or *right* = *last*)

Assuming *right* = *first*: $T_w(n) = P(n) + T_w(0) + T_w(n - 1)$

$$= c.n + T_w(n - 1)$$

$$= c.n + c.(n - 1) + T_w(n - 2)$$

$$\ldots$$

$$= \sum_{k=1}^{n} c.k + T_w(0) = c.\frac{n(n+1)}{2} = O(n^2)$$

Review and evaluation

Algorithm complexity

- The best-case occurs when the sequence is always bisected, i.e. right $= \frac{first+last}{2}$

$$T_b(n) = P(n) + 2T_b(n/2)$$
$$= c.n + 2T_b(n/2)$$
$$= c.n + 2c.(n/2) + 4T_b(n/4)$$
$$\dots$$
$$= (log_2 n)c.n + 2^{log_2 n}T_b(1)$$
$$= O(nlog_2 n)$$

Review and evaluation

Algorithm complexity

- Average case: The calculation of the mean is no longer as simple as the two cases above, so we will not consider it in detail. Proven results are:

$$T_{av}(n) = O(nlog_2n)$$

- The quicksort algorithm is efficient when the given list of elements is very long; it works better compared to the other aforementioned algorithms for sorting in such situations.

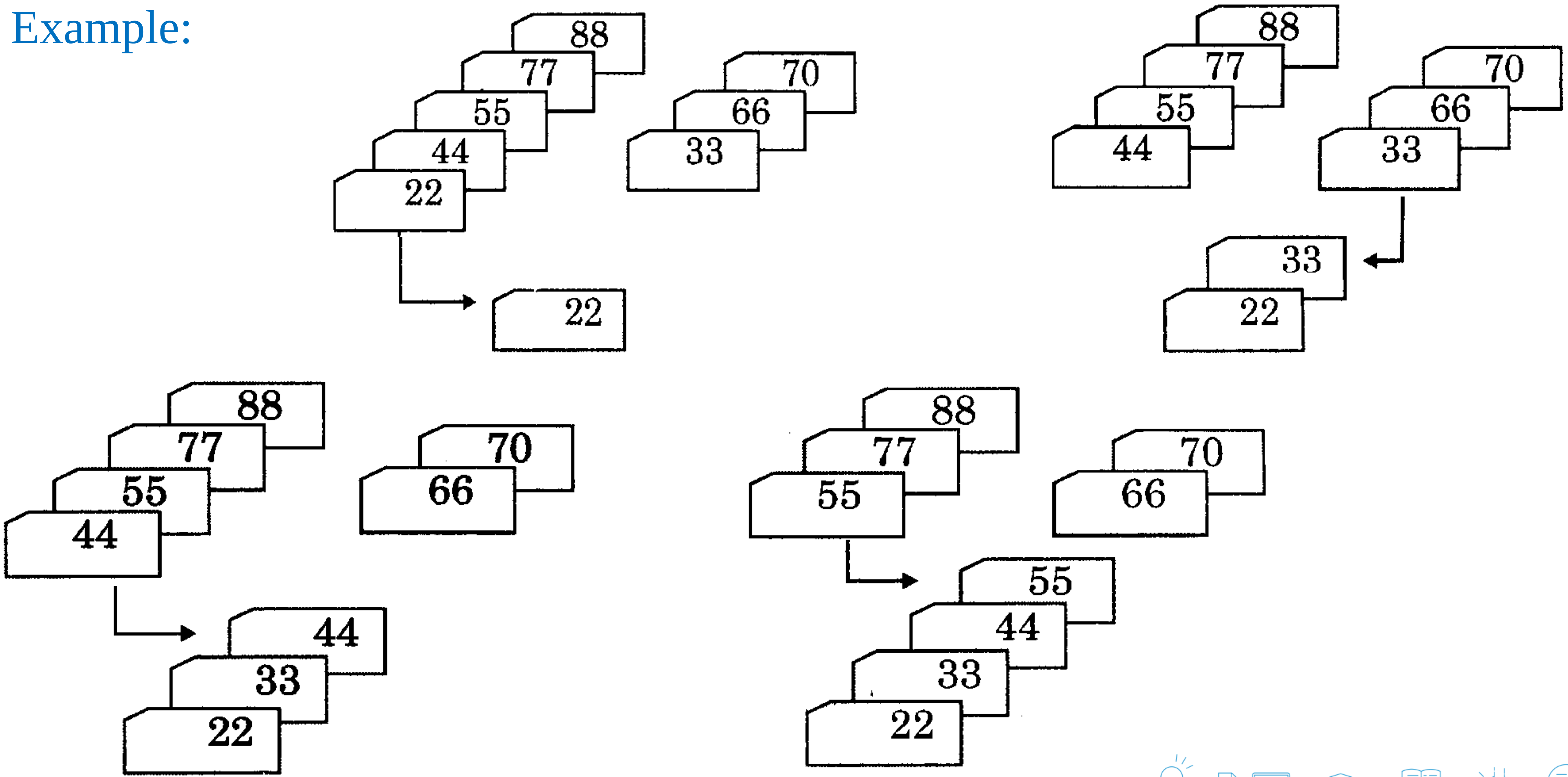
MERGE SORT

Two – ways merge

- Suppose A is a sequence consisting of r elements already sorted (called a *run*) and another sequence B consisting of s elements also sorted.
- The merging of elements of A and B into a new run consisting of $n = r + s$ elements is called two-ways merge.
- Idea of two-ways merge algorithm:
  - Compare the two smallest elements of A and B, select the smaller element to put in the resulting sequence (a storage vector of size n) and place it in the appropriate position. The selected element is removed from the run containing it.
  - The above process continues until one of the two runs has run out of elements. Transfer the entire tail of the remaining run (if any) into the resulting run.

Two – ways merge

- Example:



CMC UNIVERSITY

Timsort algorithm

- Timsort is used as the default standard sorting algorithm in all Python versions >= 2.3.
- The Timsort algorithm is an optimal algorithm for real-world long lists that is based on a combination of the *merge sort* and *insertion sort* algorithms
- The main concept of the Timsort algorithm is that it uses the insertion sort algorithm to sort small blocks (also known as chunks) of data elements, and then it uses the merge sort algorithm to merge all the sorted chunks.
- The main characteristic of the Timsort algorithm is that it takes advantage of already-sorted data elements known as *natural runs* which occur very frequently in real-world data

Timsort algorithm

The Timsort algorithm works as follows:

1. Firstly, we divide the given array of data elements into a number of *blocks* (*runs*).

2. We generally use *32* or *64* as the size of the run as it is suitable for Timsort; however, we can use any other size that can be computed from the length of the given array (*N*).

The *minrun* is the minimum length of each run. The size of the minrun can be computed by following the given principles:

a) The minrun size should not be too long as we use the insertion sort algorithm to sort these small blocks, which performs well for short lists of elements.

b) The length of the run should not be very short; in that case, it will result in a greater number of runs, which will make the merging algorithm slow.

c) Since merge sort works best when we have the number of runs as a power of *2*, it would be good if the number of runs that compute as *N/minrun* are a power of *2*.

CMC UNIVERSITY

Timsort algorithm

3. For example, if we take a run size of *32*, then the number of runs will be (*size_of_array/32*); if this is a power of *2*, then the merge process will be very efficient

4. Sort each of the runs one by one using the insertion sort algorithm.

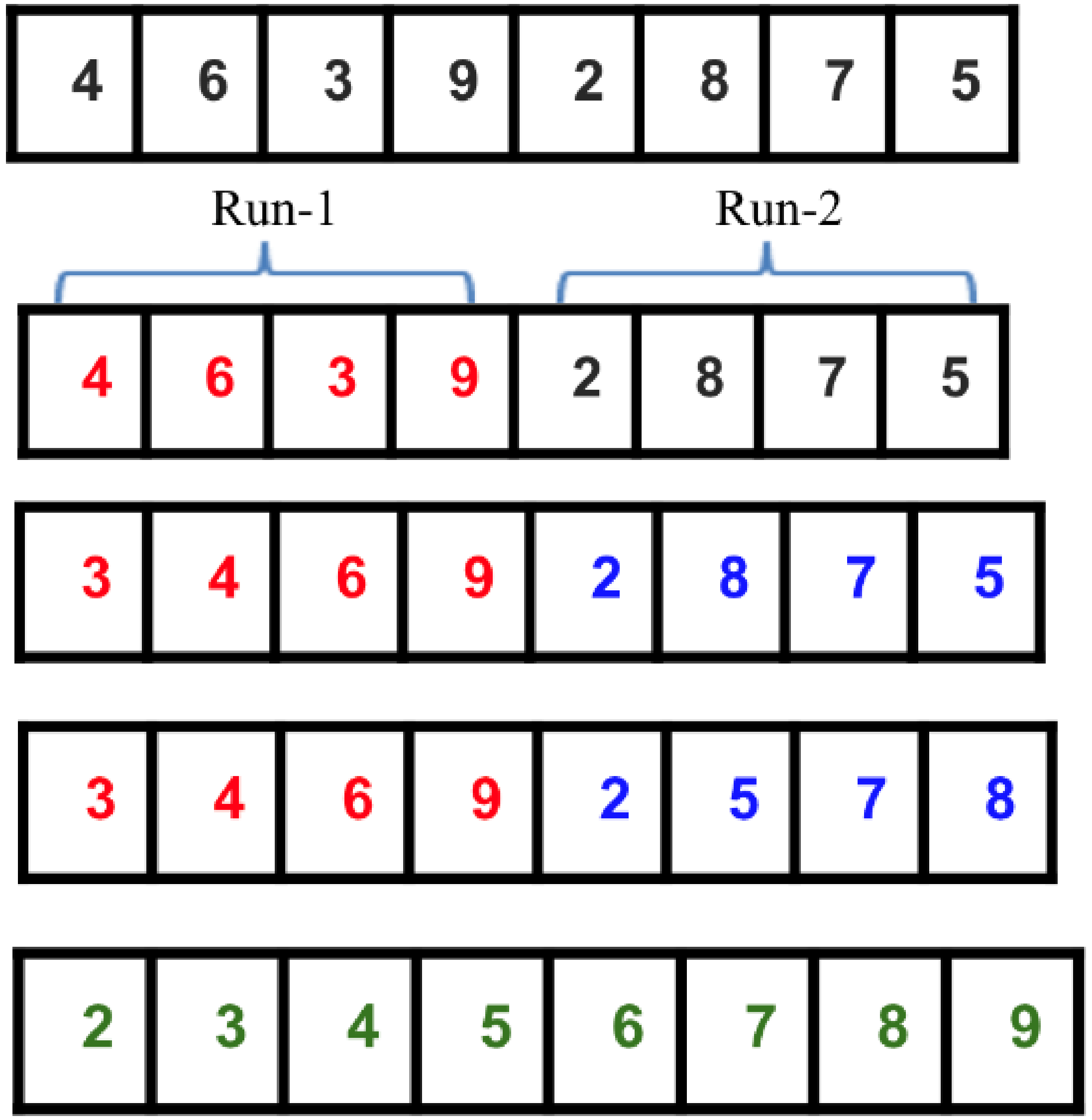
5. Merge all the sorted runs one by one using the merge method of the merge sort algorithm.

6. After each iteration, we double the size of the merged subarray.

CMC UNIVERSITY

Timsort algorithm

Example: sort {4, 6, 3, 9, 2, 8, 7, 5}



Apply insertion sort on Run-1

Run-1 is sorted, apply insertion sort on Run-2

Run-1, Run-2 are sorted, apply merge method to sort the complete list

After merging Run-1 and Run-2, get the sorted array

Implementation of the Timsort algorithm

```
def Insertion_Sort(unsorted_list):
    for index in range(1, len(unsorted_list)):
        search_index = index
        insert_value = unsorted_list[index]
        while search_index > 0 and unsorted_list[search_index-1] > insert_
value :
            unsorted_list[search_index] = unsorted_list[search_index-1]
            search_index -= 1
        unsorted_list[search_index] = insert_value
    return unsorted_list
```

Implementation of the Timsort algorithm

```
def Merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
    while j < len(second_sublist):
        merged_list.append(second_sublist[j])
        j += 1
    return merged_list
```

Implementation of the Timsort algorithm

```
def Tim_Sort(arr, run):
    for x in range(0, len(arr), run):
        arr[x : x + run] = Insertion_Sort(arr[x : x + run])

    runSize = run
    while runSize < len(arr):
        for x in range(0, len(arr), 2 * runSize):
            arr[x : x + 2 * runSize] = Merge(arr[x : x + runSize], arr[x +
runSize: x + 2 * runSize])

        runSize = runSize * 2
```

```
arr = [4, 6, 3, 9, 2, 8, 7, 5]
run = 2
Tim_Sort(arr, run)
print(arr)
```

Evaluation

- Timsort is very efficient for real-world applications since it has a worst-case complexity of O(nlogn)
- Timsort is the best choice for sorting, even if the length of the given list is short. In that case, it uses the insertion sort algorithm, which is very fast for smaller lists, and the Timsort algorithm works fast for long lists due to the merge method
- Hence, the Timsort algorithm is a good choice for sorting due to its adaptability for sorting arrays of any length in real-world usage.

CMC UNIVERSITY

Evaluation

- A comparison of the complexities of different sorting algorithms is given in the following table:

Algorithm	**Worst-case**	**Average-case**	**Best-case**
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n^2)$	O(nlogn)	O(nlogn)
Timsort	O(nlogn)	O(nlogn)	$O(n)$

CMC UNIVERSITY

THANK YOU