Sorting algorithms

Data Structures and Algorithms

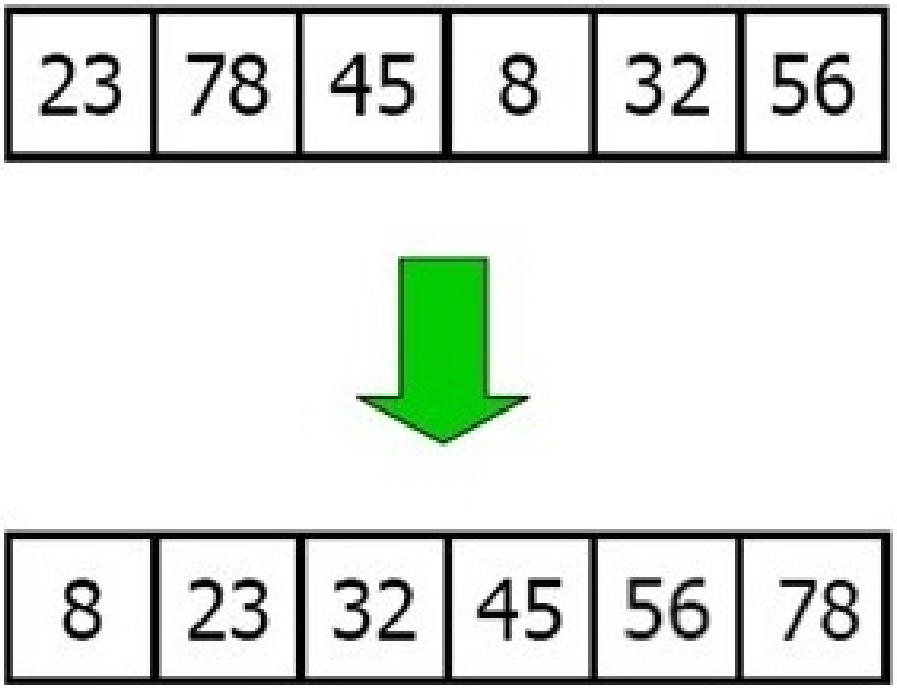
MEng. Duy Tran Ngoc Bao
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Overview

Sorting concepts

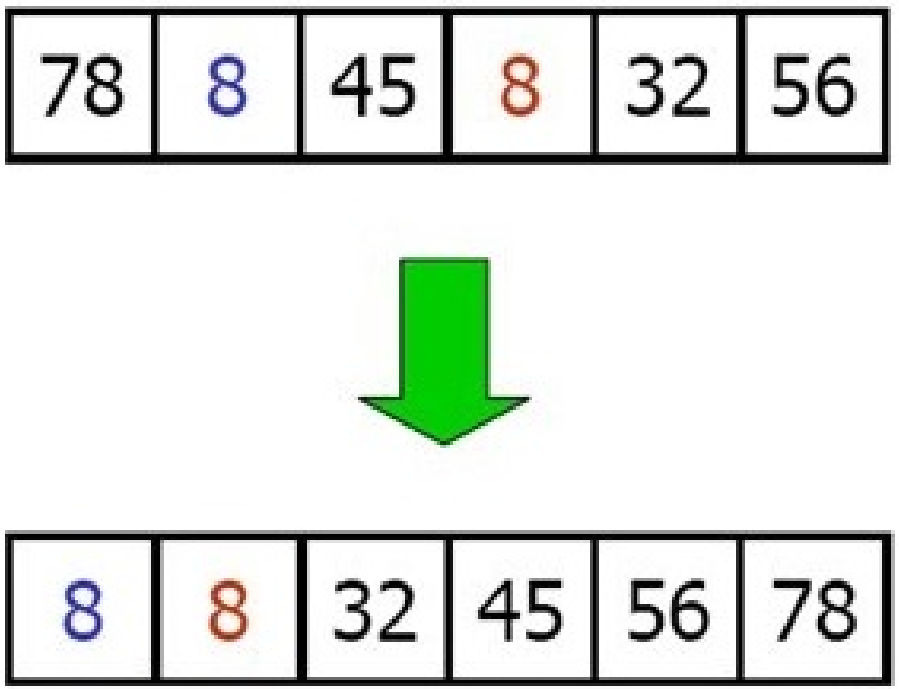
Sorting

One of the most important concepts and common applications in computing.



Sorting

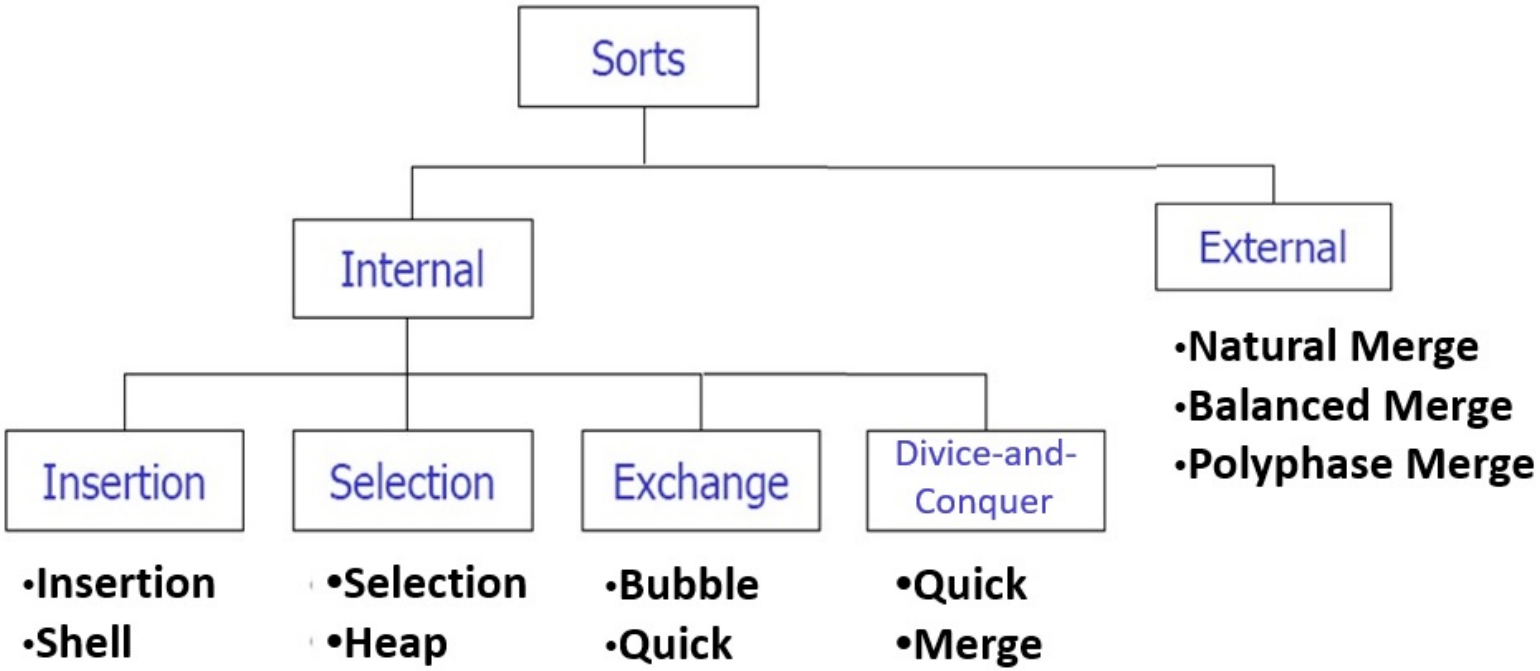
Sort stability: data with equal keys maintain their relative input order in the output.



Sorting

Sort efficiency: a measure of the relative efficiency of a sort = number of comparisons + number of moves.

Sorting



Selection Sort

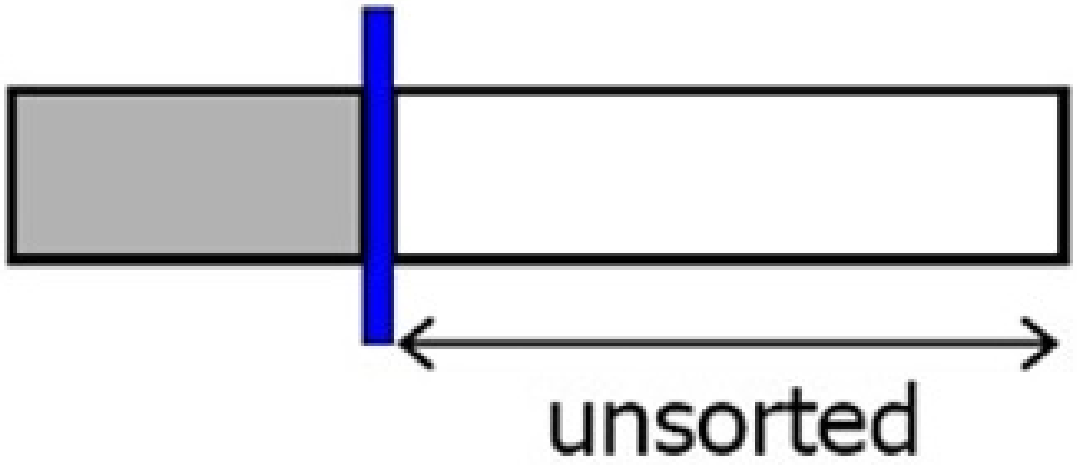
Selection Sort

Idea

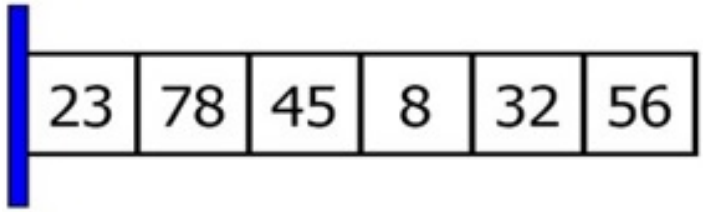
In each pass, the smallest/largest item is selected and placed in a sorted list.

Straight Selection Sort

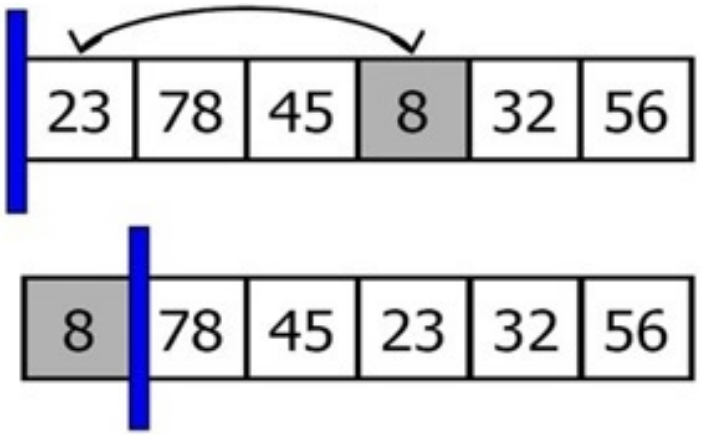
- The list is divided into two parts: sorted and unsorted.
- In each pass, in the unsorted sublist, the smallest element is selected and exchanged with the first element.



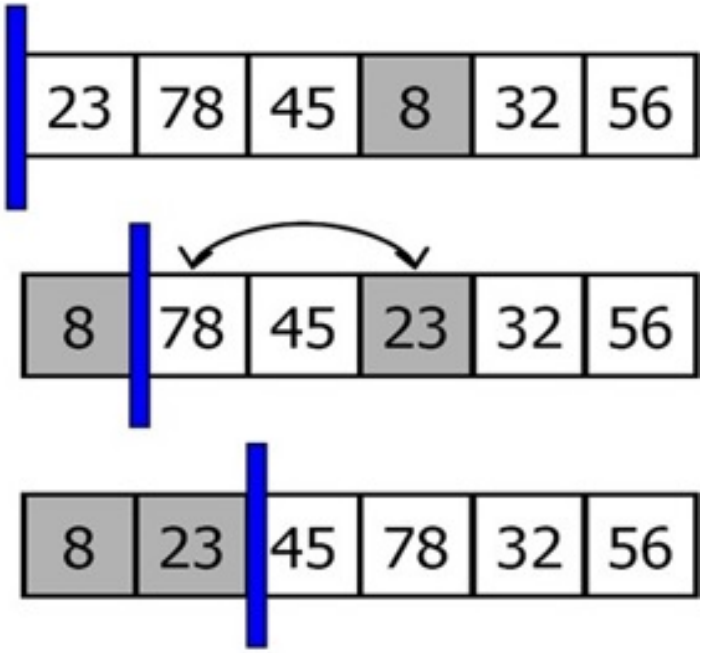
Straight Selection Sort



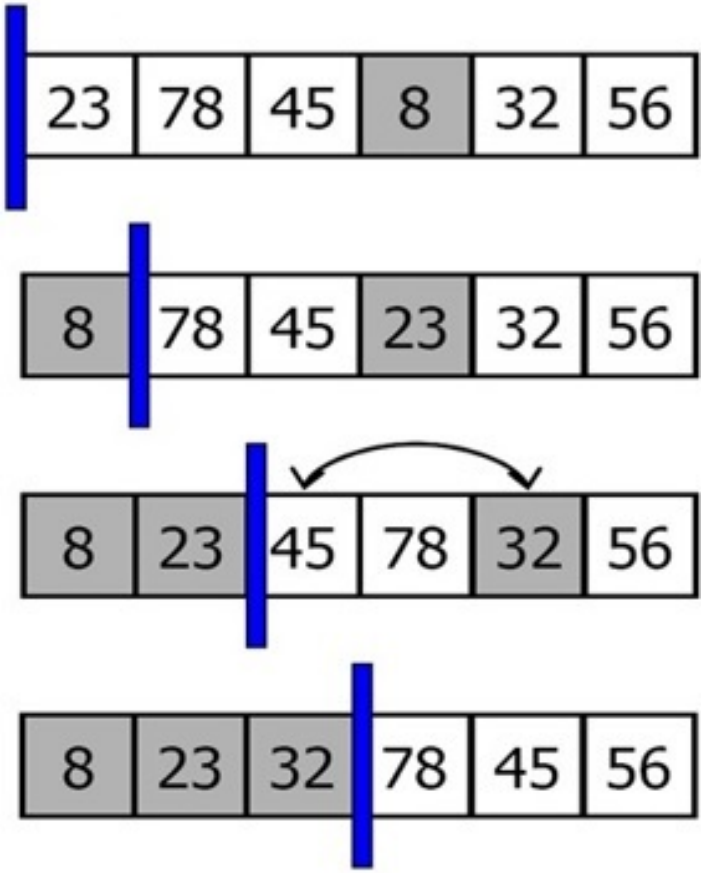
Straight Selection Sort



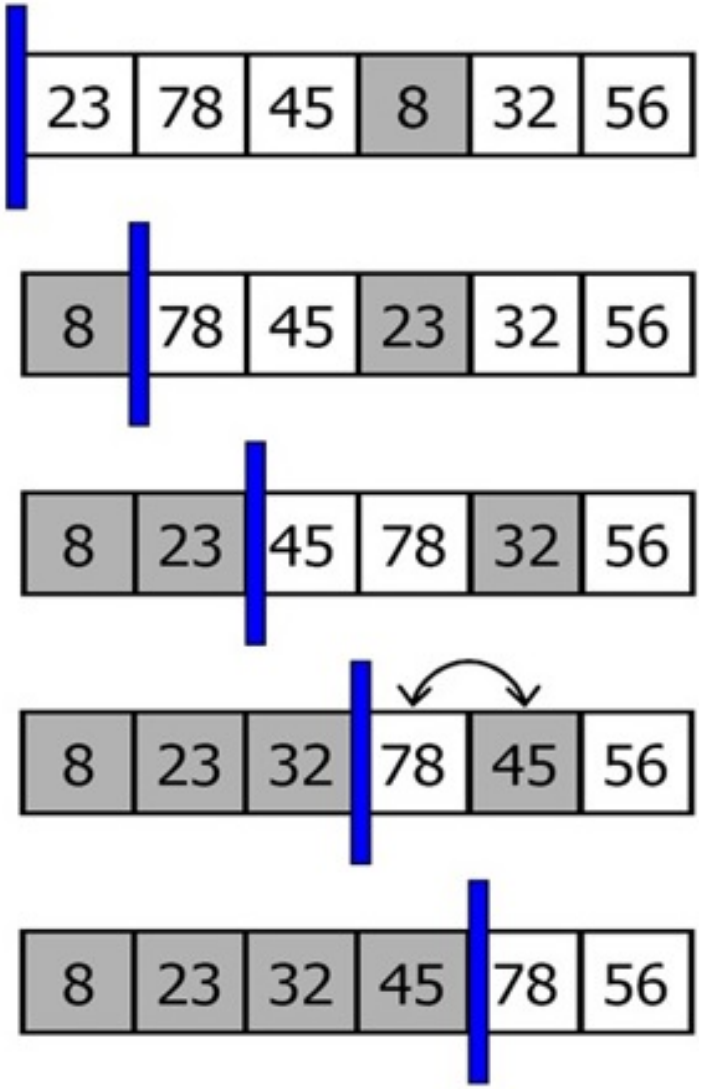
Straight Selection Sort



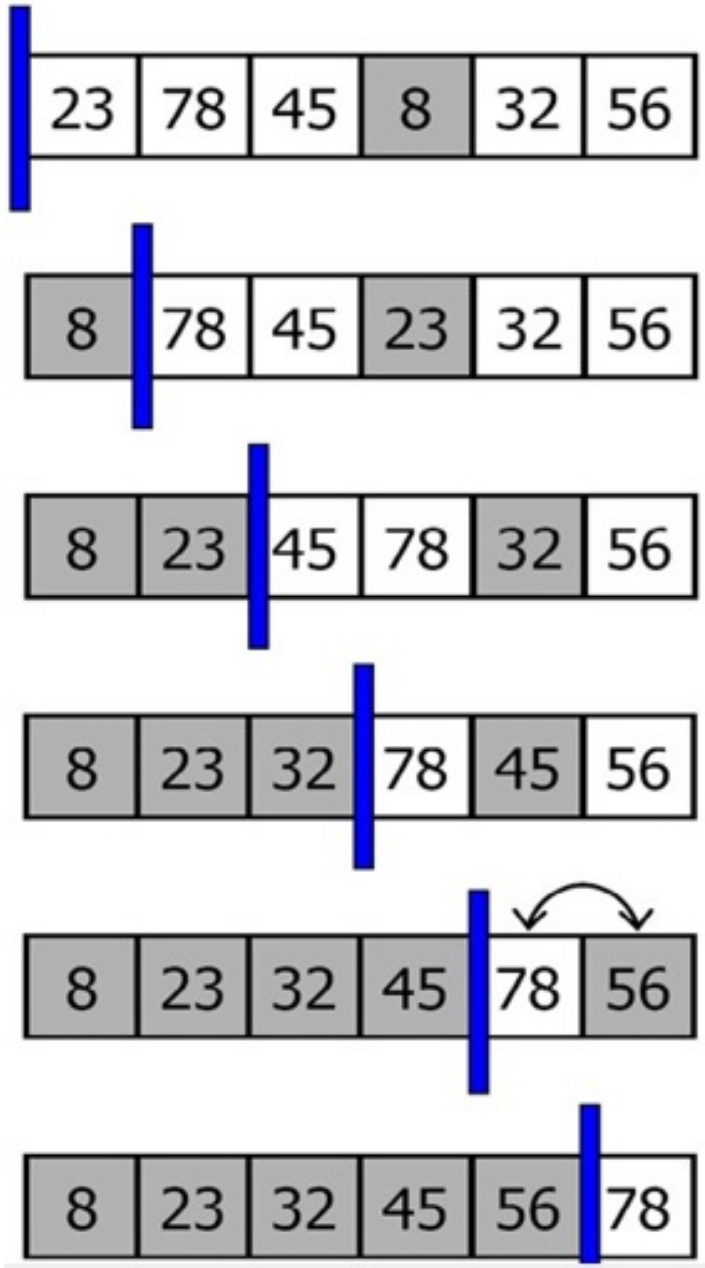
Straight Selection Sort



Straight Selection Sort



Straight Selection Sort



Straight Selection Sort: Pseudocode

```
1  current = 0
2  while current < count - 1 do
3  |   smallest = current
4  |   walker = current + 1
5  |   while walker < count do
6  |   |   if data[walker] < data[smallest] then
7  |   |   |   smallest = walker
8  |   |   end
9  |   |   walker = walker + 1
10 |   end
11 |   swap(current, smallest)
12 |   current = current + 1
13 end
```

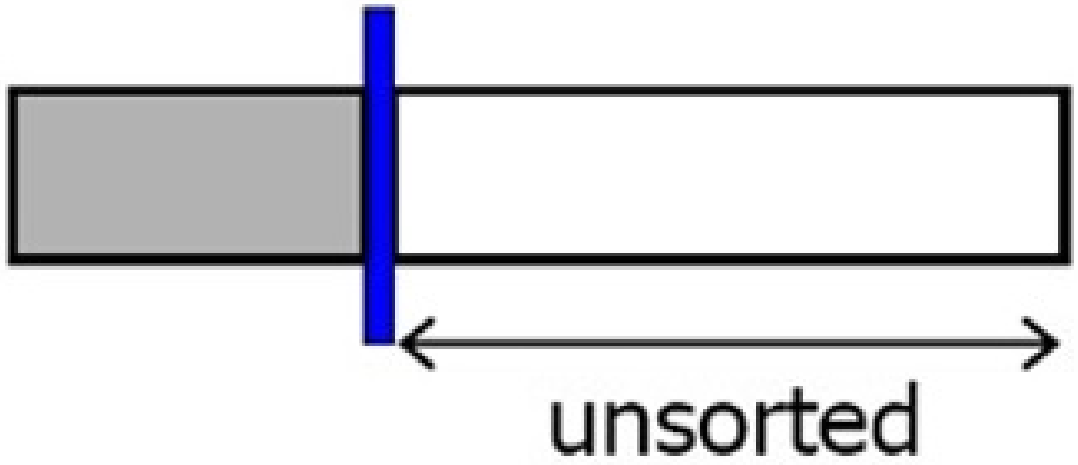

Selection Sort Efficiency

- Straight selection sort: $O(n^2)$

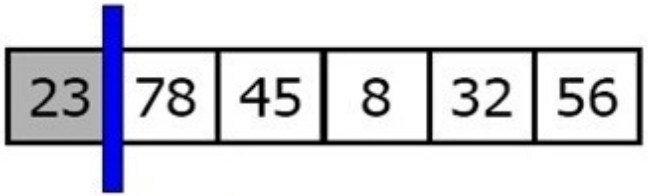
Insertion Sort

Straight Insertion Sort

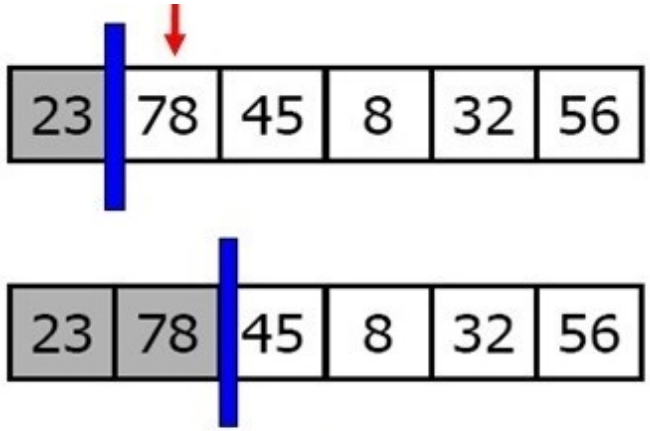
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted sublist is inserted into the sorted sublist.



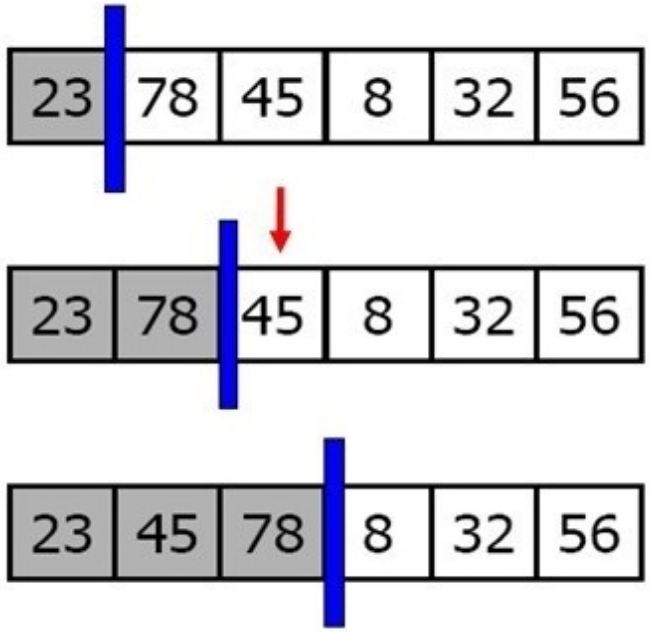
Straight Insertion Sort



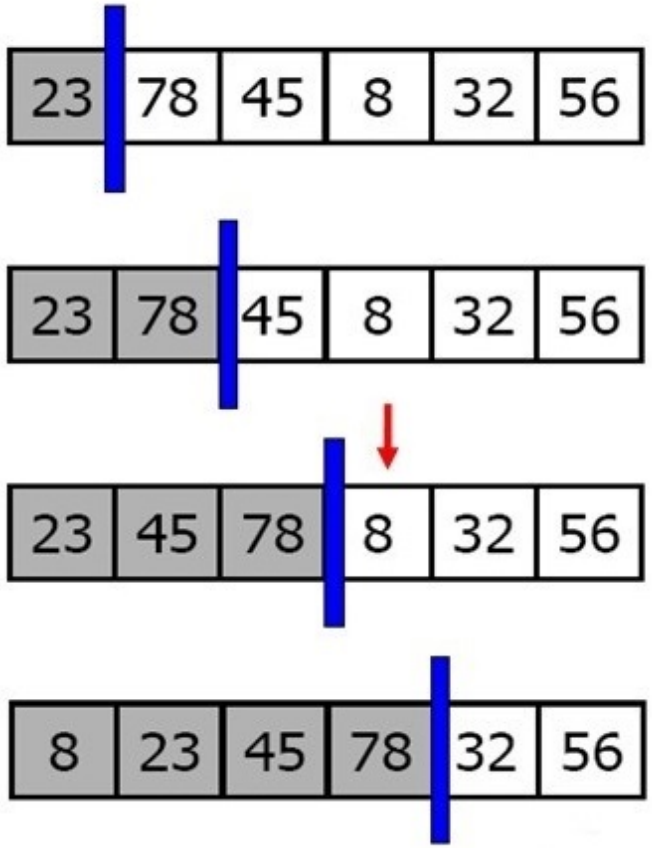
Straight Insertion Sort



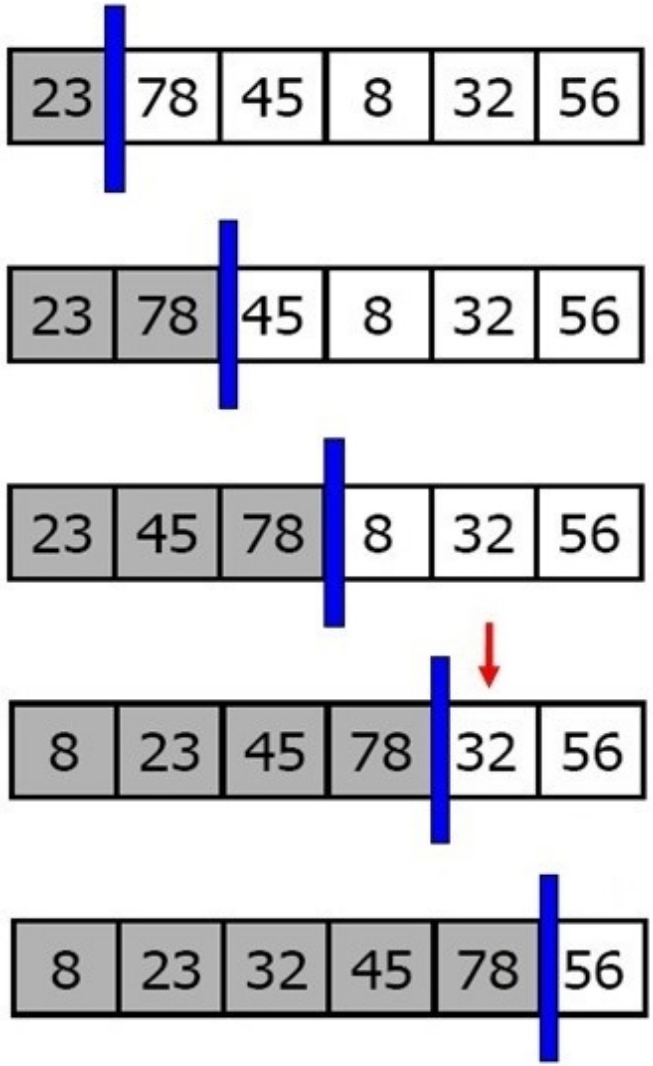
Straight Insertion Sort



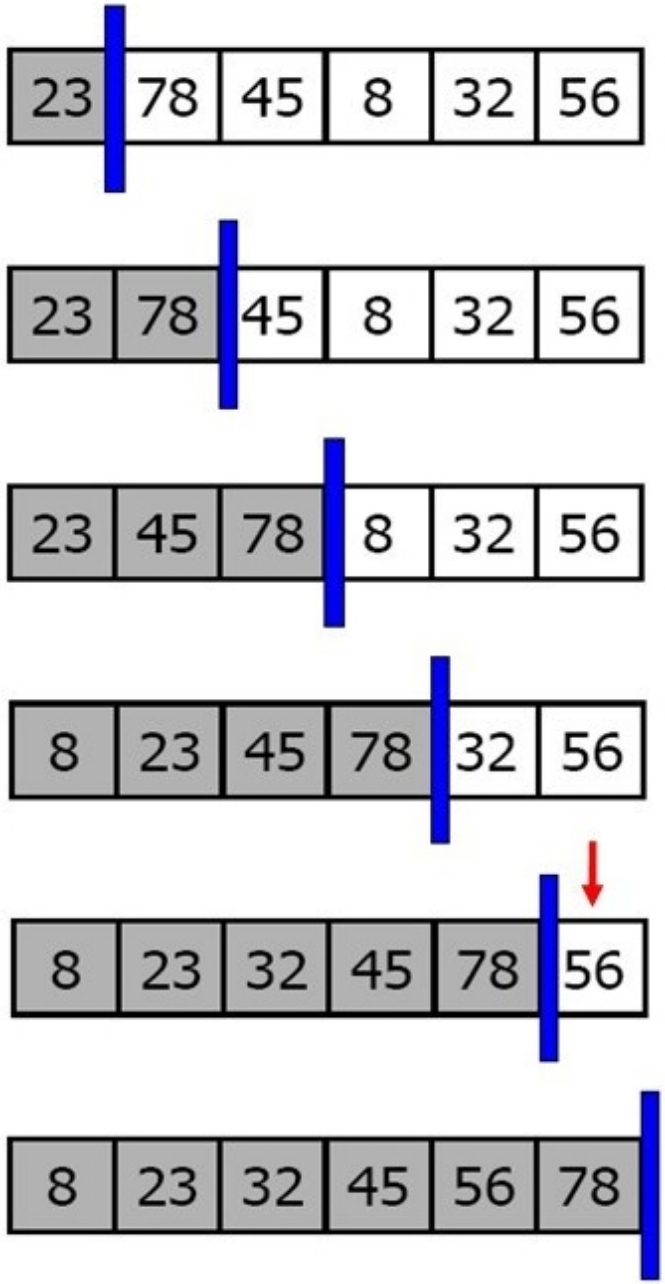
Straight Insertion Sort



Straight Insertion Sort



Straight Insertion Sort



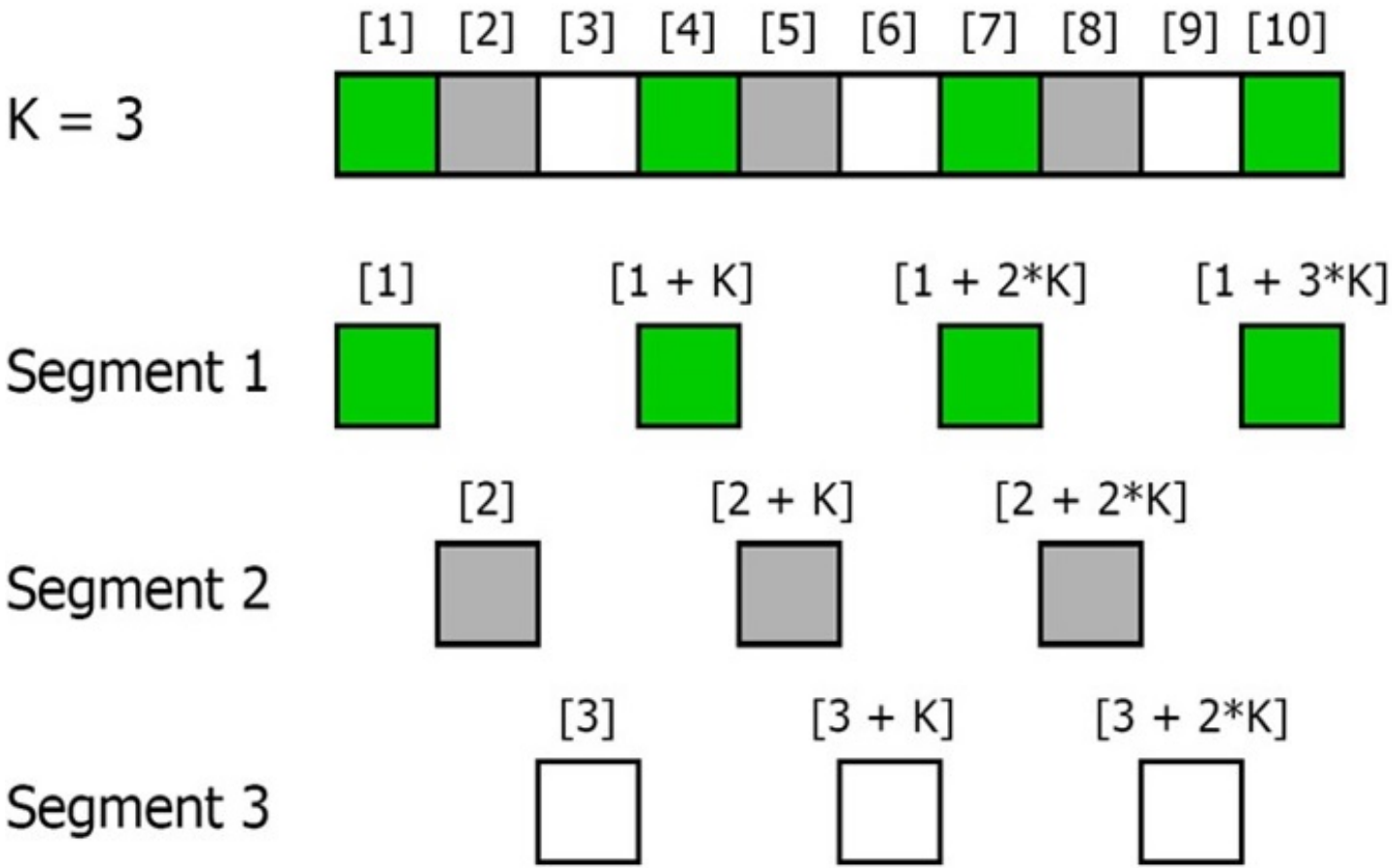
Straight Insertion Sort: Pseudocode

```
1  if count > 1 then
2      curr = 1
3      while curr < count do
4          tmp = data[curr]
5          step = curr - 1
6          while step ≥ 0 AND tmp < data[step] do
7              data[step + 1] = data[step]
8              step = step - 1
9          end
10         data[step + 1] = tmp
11         curr = curr + 1
12     end
13 end
```

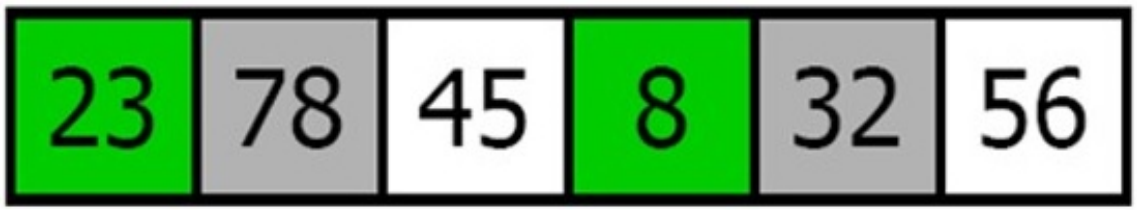

Shell Sort

- Named after its creator Donald L. Shell (1959).
- Given a list of N elements, the list is divided into K segments (K is called the increment).
- Each segment contains $\frac{N}{K}$ or more elements.
- Segments are dispersed throughout the list.
- Also is called diminishing-increment sort.

Shell Sort



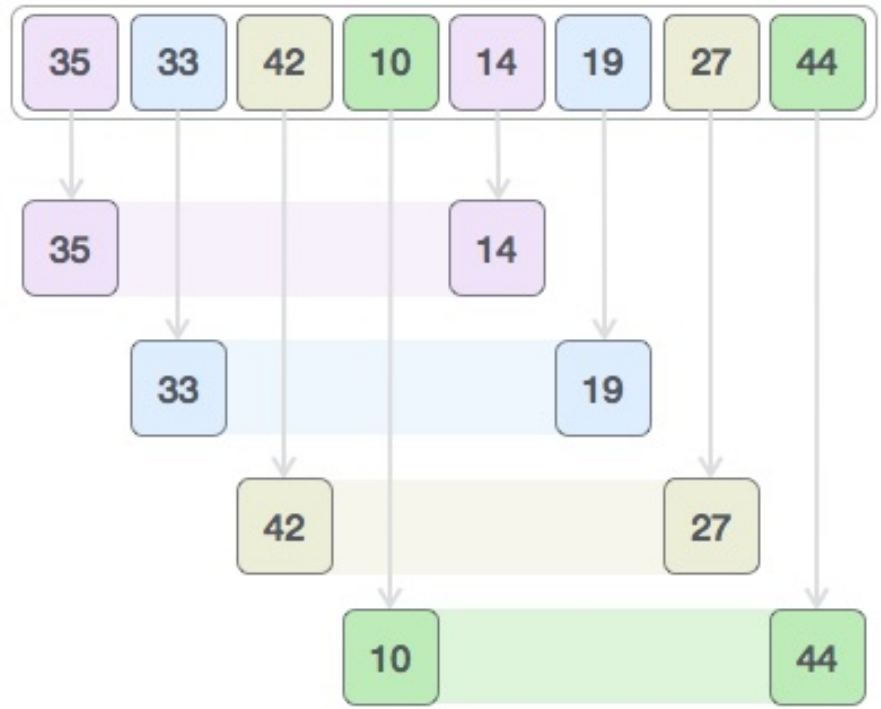
Shell Sort



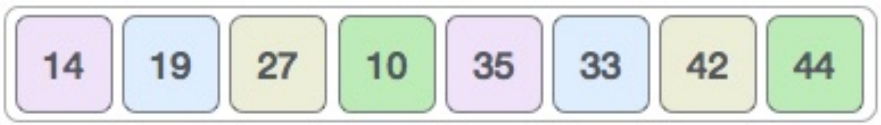
- For the value of K in each iteration, sort the K segments.
- After each iteration, K is reduced until it is 1 in the final iteration.

Example of Shell Sort

$K = 4$:

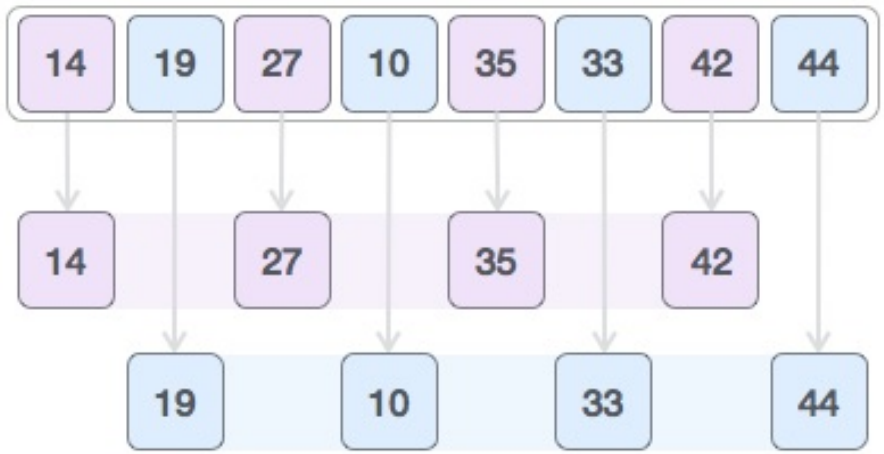


Result:

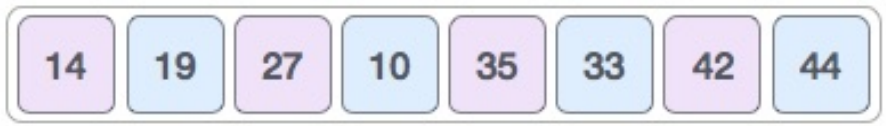


Example of Shell Sort

$K = 2$:

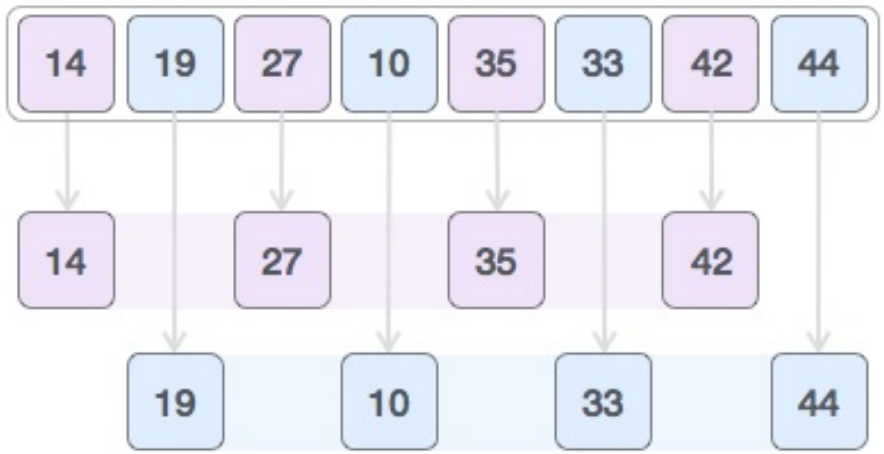


Result:

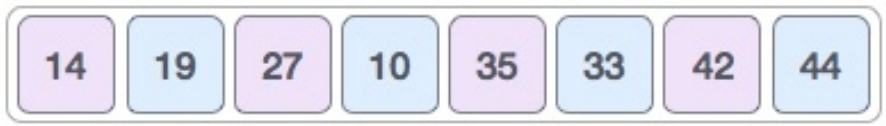


Example of Shell Sort

$K = 2$:

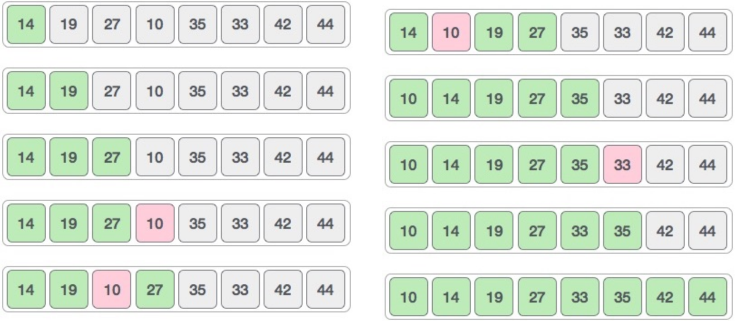


Result:

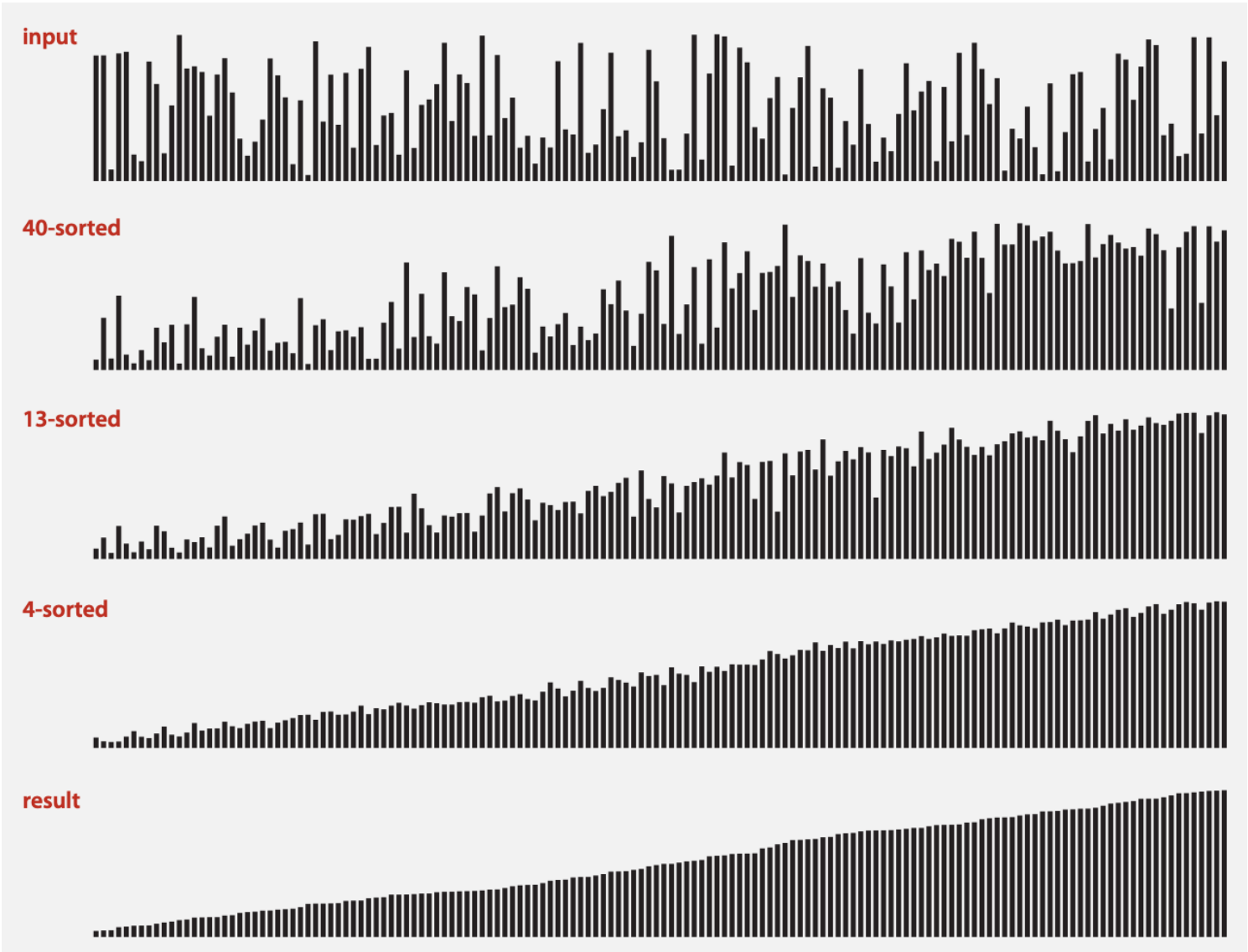


Example of Shell Sort

$K = 1$:



Shell sort: Visual trace



Choosing incremental values

- From more of the comparisons, it is better when we can receive more new information.
- Incremental values should not be multiples of each other, other wise, the same keys compared on one pass would be compared again at the next.
- The final incremental value must be 1.

Choosing incremental values

- Incremental values may be:
  $1, 4, 13, 40, 121, ...$
  $k_t = 1$
  $k_{i-1} = 3 * k_i + 1$
  $t = \lfloor \log_3 n \rfloor - 1$

- or:
  $1, 3, 7, 15, 31, ...$
  $k_t = 1$
  $k_{i-1} = 2 * k_i + 1$
  $t = \lfloor \log_2 n \rfloor - 1$

Shell Sort: Pseudocode

```
1 k = next_increment()
2 while k ≥ 1 do
3 |   segment = 1
4 |   while segment ≤ k do
5 |   |   sort_segment(segment, k)
6 |   |   segment = segment + 1
7 |   end
8 |   k = next_increment()
9 end
```


Shell Sort - Sort Segment: Pseudocode

```
sort_segment(val segment <int>, val k <int>)

1  curr = segment + k
2  while curr < count do
3  |   temp = data[curr]
4  |   step = curr - k
5  |   while step >=0 AND tmp < data[step] do
6  |   |   data[step + k] = data[step]
7  |   |   step = step - k
8  |   end
9  |   data[step + k] = tmp
10 |   curr = curr + k
11 end
```


Insertion Sort Efficiency

- Straight insertion sort:
$$f(n) = \frac{n(n+1)}{2} = O(n^2)$$
- Shell sort:
$O(n^{1.25})$ (Empirical study)

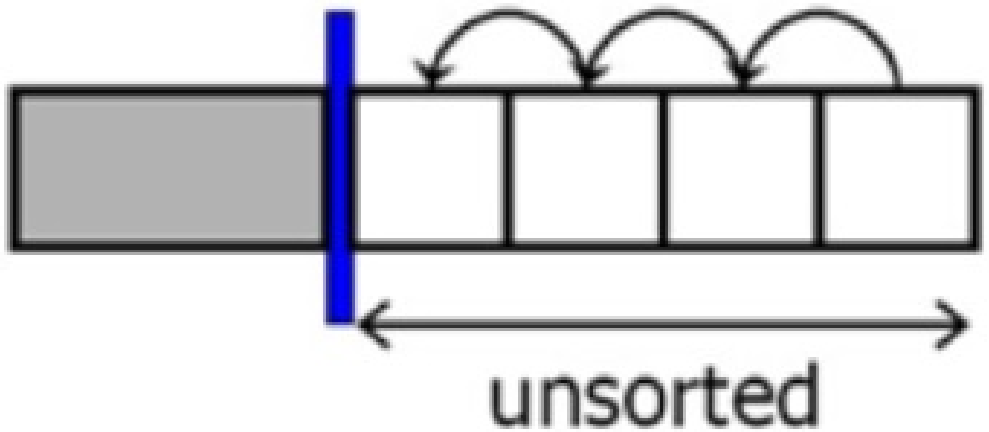
Exchange Sort

Exchange Sort

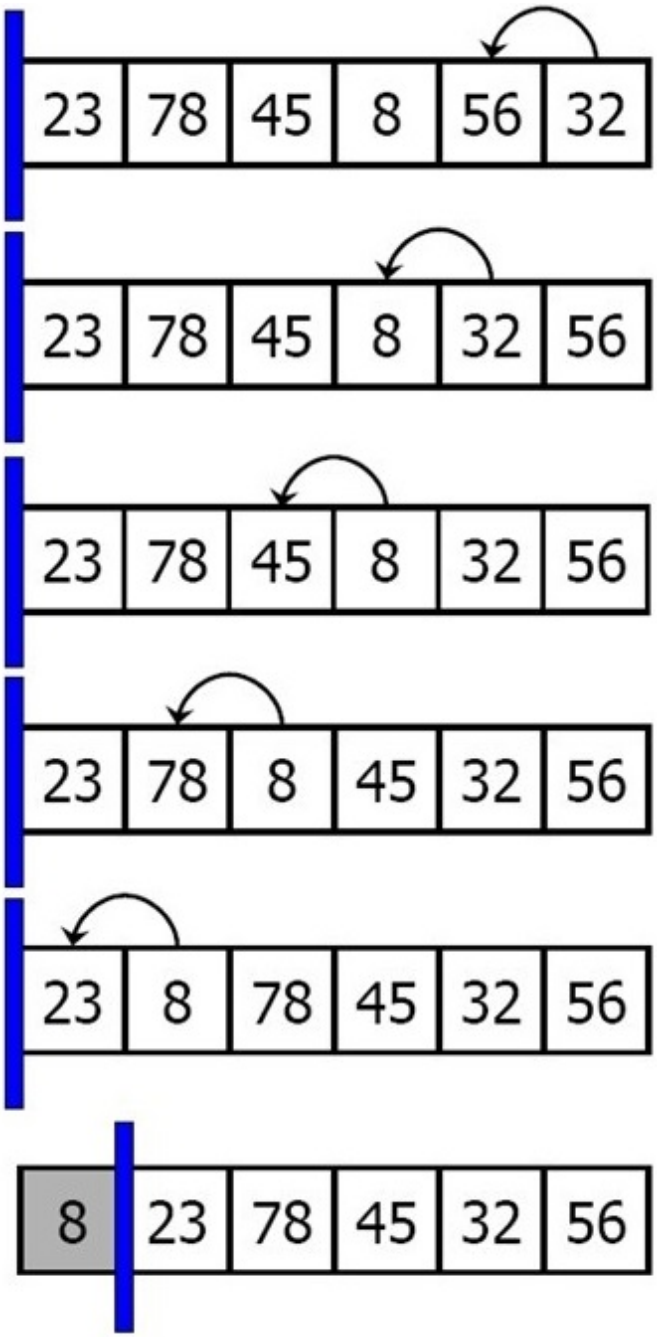
- In each pass, elements that are out of order are exchanged, until the entire list is sorted.
- Exchange is extensively used.

Bubble Sort

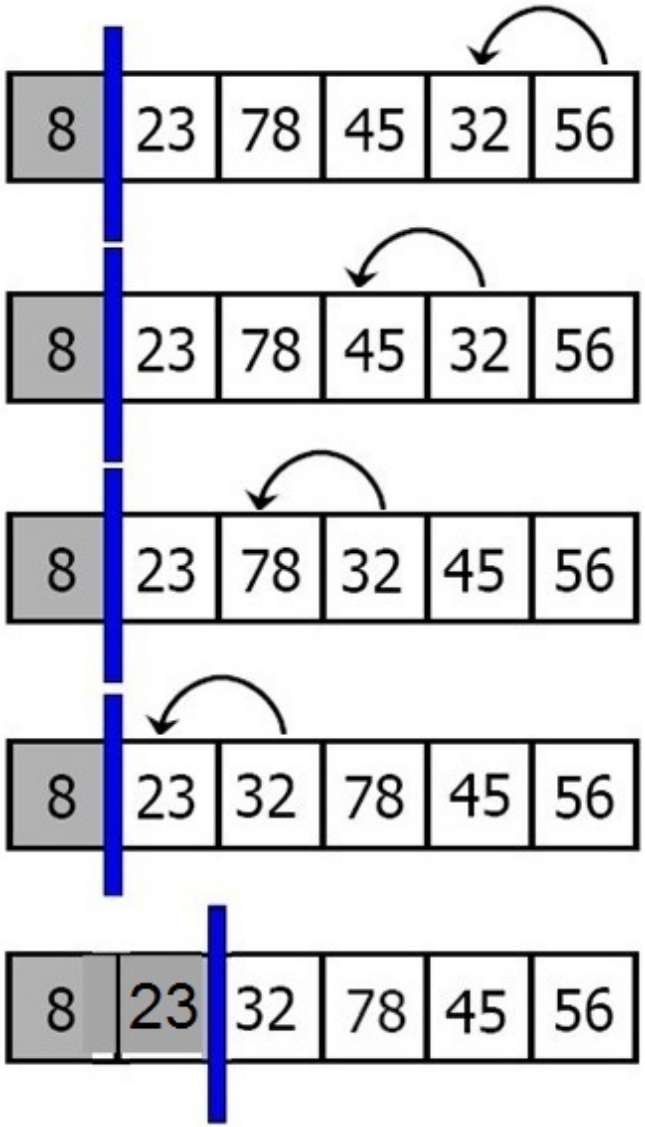
- The list is divided into two parts: sorted and unsorted.
- In each pass, the smallest element is bubbled from the unsorted sublist and moved to the sorted sublist.



Bubble Sort



Bubble Sort



Bubble Sort

```
1  curr = 0
2  flag = False
3  while curr < count AND flag = False do
4  |   step = count - 1
5  |   flag = True
6  |   while step > curr do
7  |   |   if data[step] < data[step - 1] then
8  |   |   |   flag = False
9  |   |   |   swap(data[step], data[step - 1])
10 |   |   end
11 |   |   step = step - 1
12 |   end
13 |   curr = curr + 1
14 end
```


Exchange Sort Efficiency

- Bubble sort:

$$f(n) = \frac{n(n+1)}{2} = O(n^2)$$

Divide-and-Conquer

Divide-and-Conquer Sort

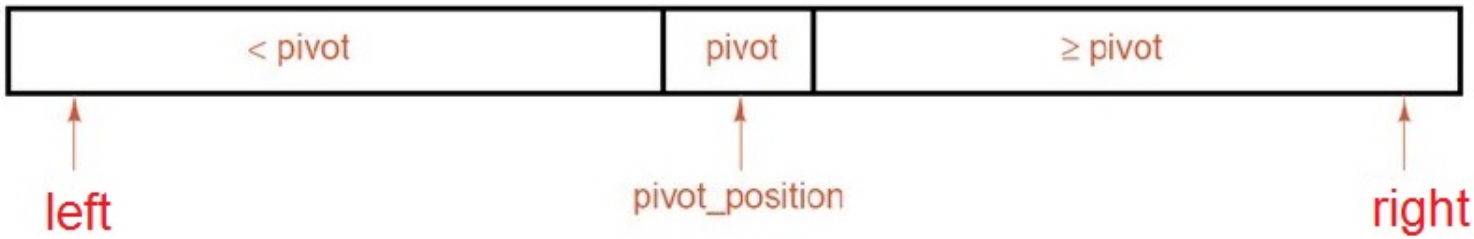
```
1 Algorithm DivideAndConquer()
2 if the list has length > 1 then
3 |   partition the list into lowlist and
  |    highlist
4 |   lowlist.DivideAndConquer()
5 |   highlist.DivideAndConquer()
6 |   combine(lowlist, highlist)
7 end
8 End DivideAndConquer
```


Divide-and-Conquer Sort

	Partition	Combine
Merge Sort	easy	hard
Quick Sort	hard	easy

Quick Sort

Given a pivot value, the partition rearranges the entries in the list as the following figure:



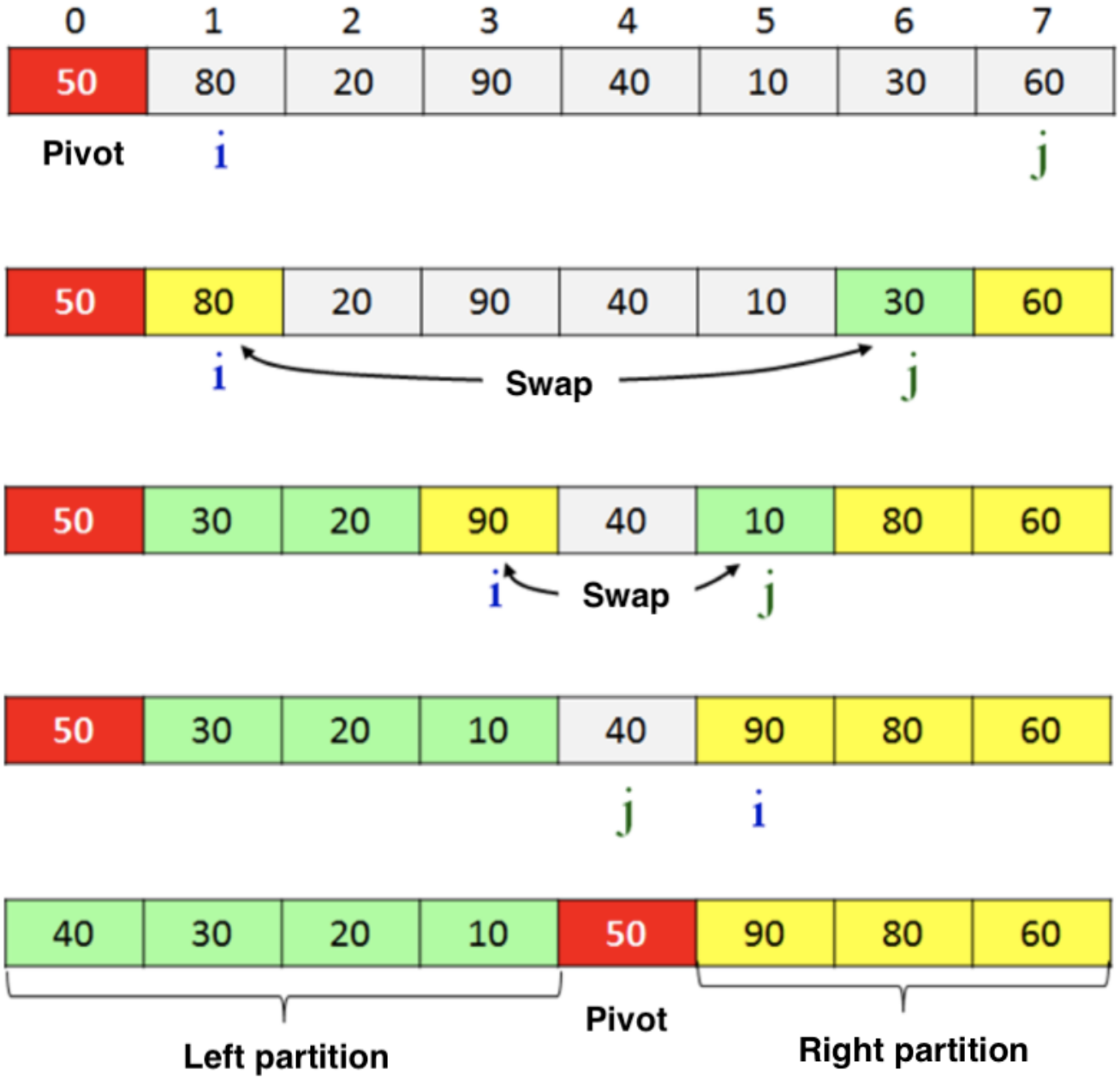
Quick Sort

```
1 Algorithm QuickSort()
2 Sorts the contiguous list using quick sort.

3 recursiveQuickSort(0, count - 1)
4 End QuickSort
```


Quick Sort

```
1 Algorithm recursiveQuickSort(val left
    <int>, val right <int>)
2 Sorts the contiguous list using quick sort.
3 Pre: left and right are valid positions
    in the list
4 Post: list sorted

5 if left < right then
6 |   pivot_position = Partition(left, right)
7 |   recursiveQuickSort(left,
  |     pivot_position - 1)
8 |   recursiveQuickSort(pivot_position +
  |     1, right)
9 end
```

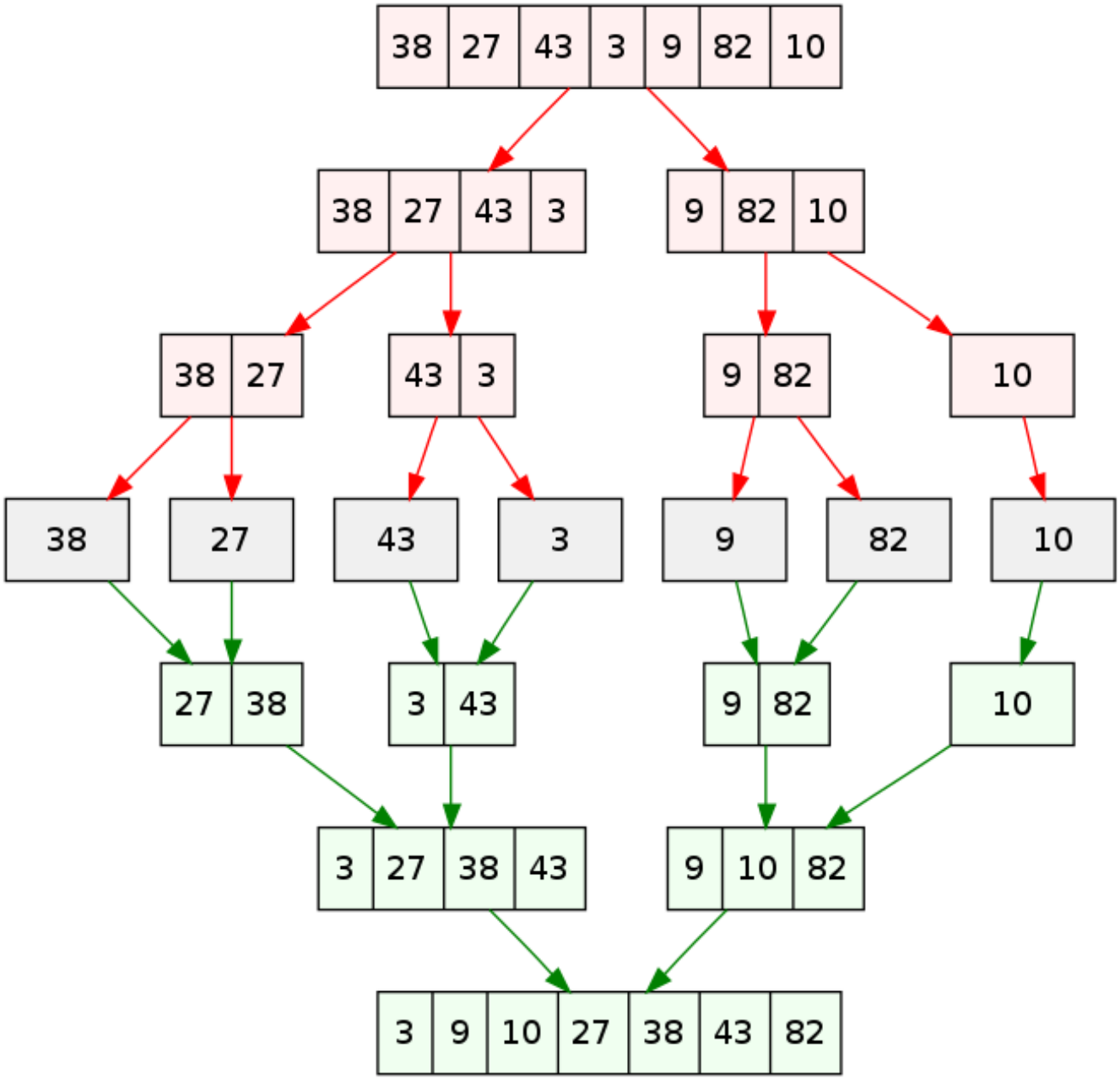

Partitioning



Quick Sort Efficiency

- Quick sort: $O(n \log_2 n)$

Merge Sort



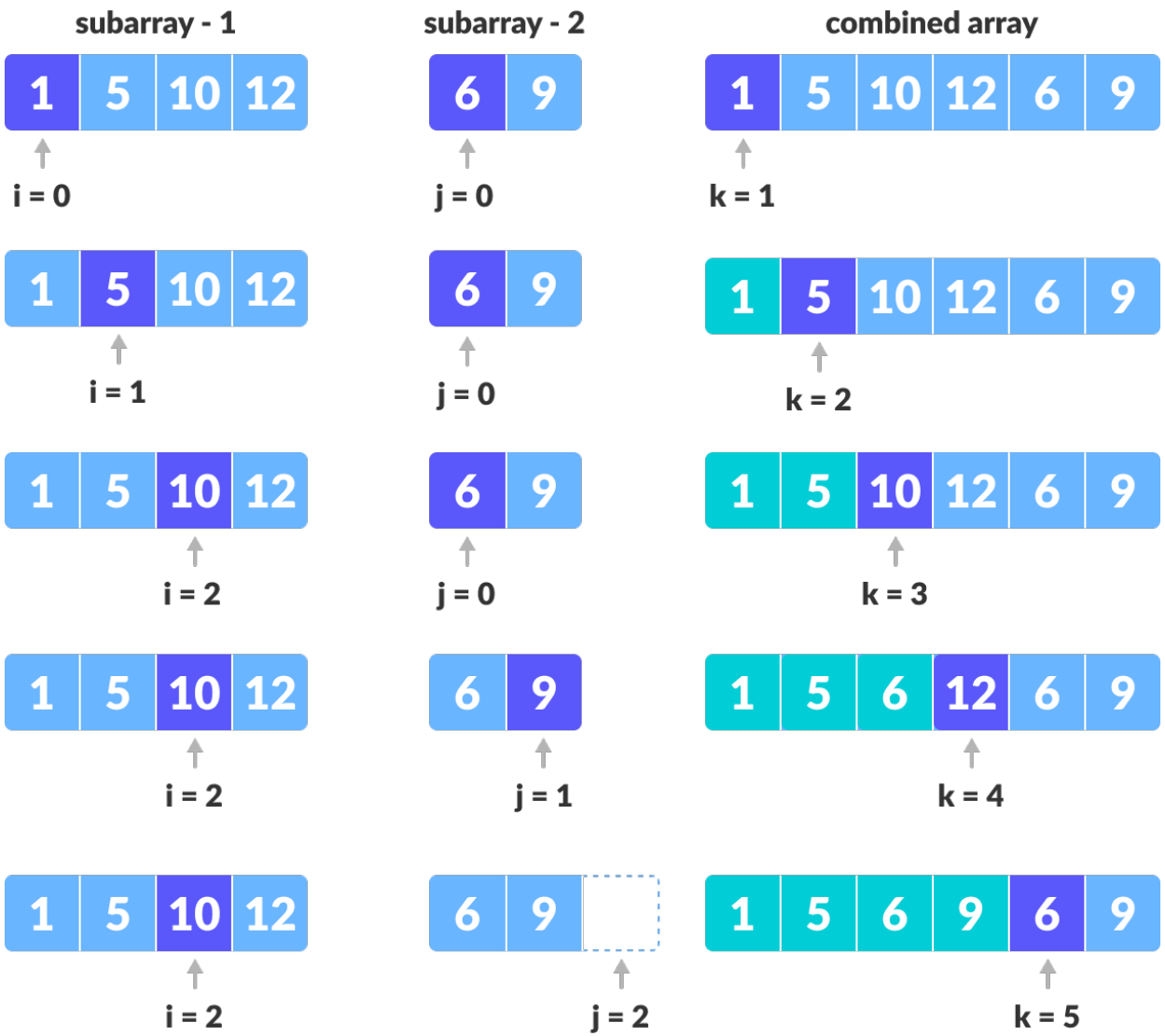
Merge Sort

```
1 Algorithm MergeSort()
2 Sorts the contiguous list using merge sort.

3 recursiveMergeSort(arr, 0, arr.length() - 1)
4 End MergeSort
```


Merge Sort

```
1 Algorithm recursiveMergeSort(ref arr
    <array>, val hi <int> , val lo <int>)

2 if hi > lo then
3 |   mid = lo + (hi - lo) / 2
4 |   recursiveMergeSort(arr, lo, mid)
5 |   recursiveMergeSort(arr, mid + 1, hi)
6 |   merge(arr, lo, mid, hi)
7 end
8 End recursiveMergeSort
```


Merge - Combine operation



Merge

```
1  Algorithm merge(ref arr <array>, val hi
     <int>, val mid <int>, val lo <int>)

2  Allocate new array aux with the length of
     arr.
3  k = lo
4  while k ≥ hi do
5  |   aux[k] = arr[k] i++
6  end
7  k = lo
8  while k ≥ hi do
9  |   if i > mid then
10 |   |   a[k] = aux[j++]
11 |   end
```


THANK YOU.