[SWE2015-41] Introduction to Data Structures (자료구조개론)

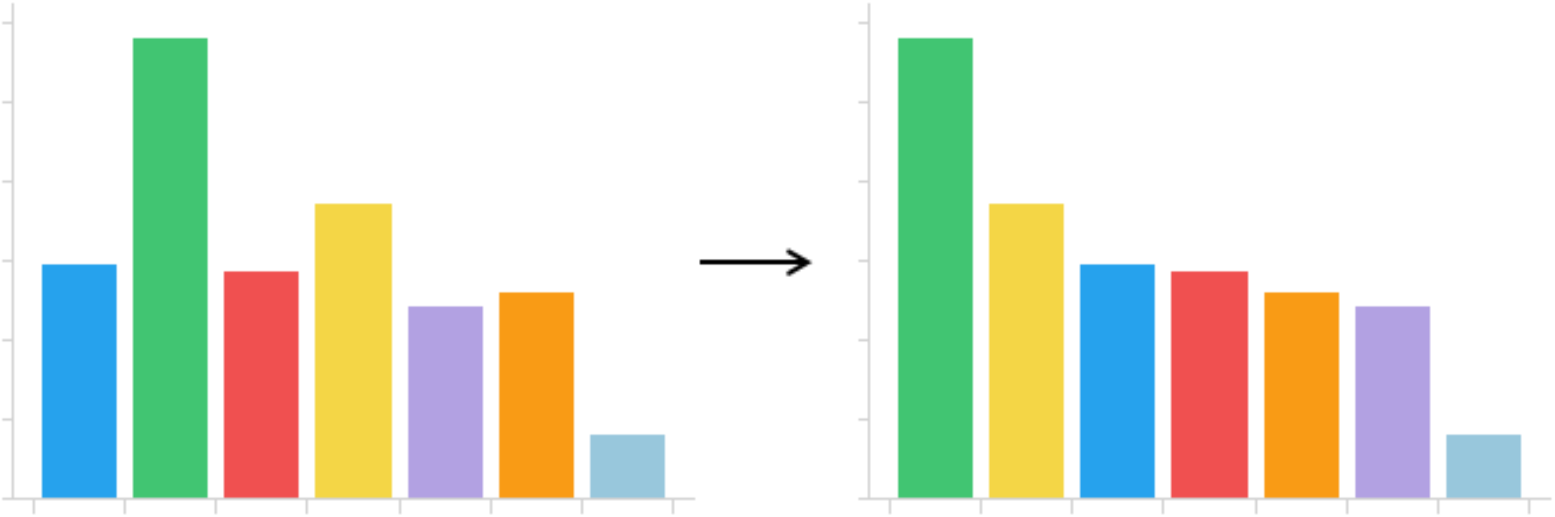
Sorting Algorithms

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

What is the Sorting Algorithm?

- A **sorting algorithm** aims to **arrange a set of items in a specific order**
  - The most frequently used orders are numerical and lexicographical orders
    - Numerical order: 1, 2, 3, 4, … / lexicographical order: a, aa, ab, b, c, …
    - Ascending: 1, 2, 3, 4, …, 9, 10 / descending: 10, 9, 8, 7, …, 2, 1
  - The output is a permutation of the input items



Why is Sorting Important?

- **Efficient sorting** is important for optimizing efficiency of other algorithms
  - E.g., the Kruskal's algorithm requires the sorted list of edges
    - Its complexity is $O(|E|\log|E|)$, which is the sorting complexity of the edge set E
  - Other examples: search, merge, matching, …
  - Sorting is useful for canonicalizing data and for producing human-readable outputs

- No best algorithm for all situations
  - Efficiency depends on the initial ordering and/or the number of items
  - E.g., insertion sort is useful if the items are already mostly sorted

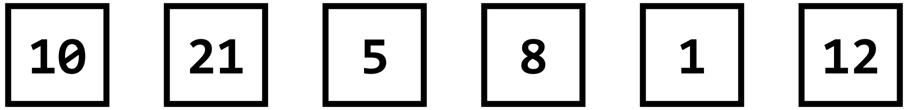
- The analysis of sorting algorithms is good for understanding algorithms
  - E.g., time complexity analysis, algorithm design, …

Sorting Algorithms

- There exists a lot of sorting algorithms
  - Comparison sorting algorithms:
    - Examples: Selection sort, Bubble sort, Insertion sort, Quick sort, …
    - Any comparison sort cannot perform better than $O(N \log N)$
  - Non-comparison sorting algorithms:
    - Examples: Radix sort, Bucket sort, Counting sort
    - They are not limited to $O(N \log N)$, but require some assumptions, e.g., digit/dictionary size, …
- In this lecture, we will learn the following **comparison-based algorithms**:
  - Simple $O(N^2)$ algorithms: Selection Sort, Bubble Sort, Insertion Sort
  - Efficient $O(N \log N)$ algorithms: Heap Sort, Quick Sort, Merge Sort
  - **Note.** For simplicity, we focus on sorting integers in the increasing order

Selection Sort

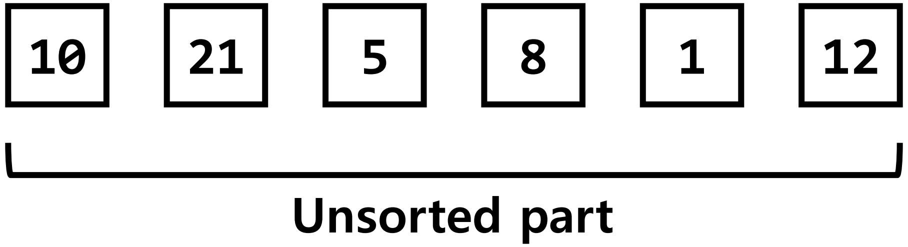
- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
    1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
        - `A[0,…,i-1]` is already sorted
        - `A[i,…,N-1]` is remaining to be sorted
    2. Select the smallest item from the unsorted part `A[i,…,N-1]`
    3. Exchange the item with the `i`-th item `A[i]`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `...`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

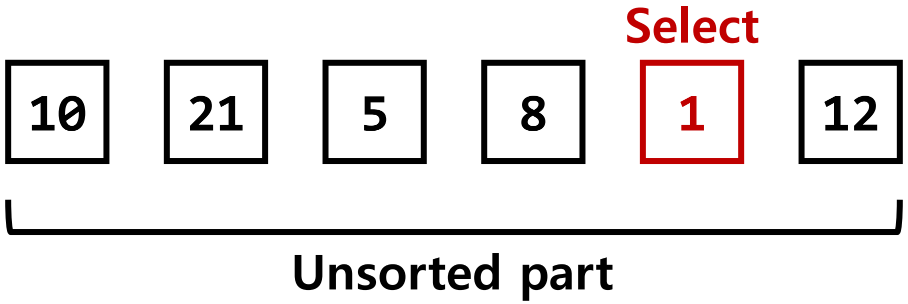
`i=0 iteration`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

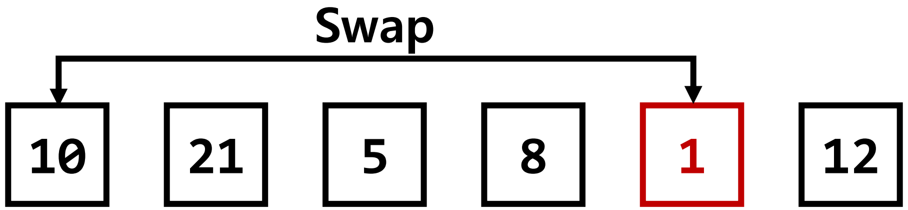
`i=0 iteration`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

`i=0 iteration`



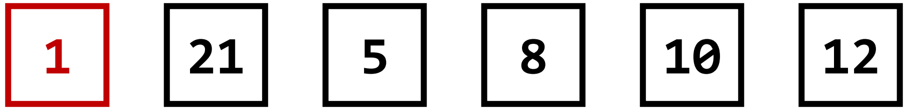
Swap

10	21	5	8	1	12

Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

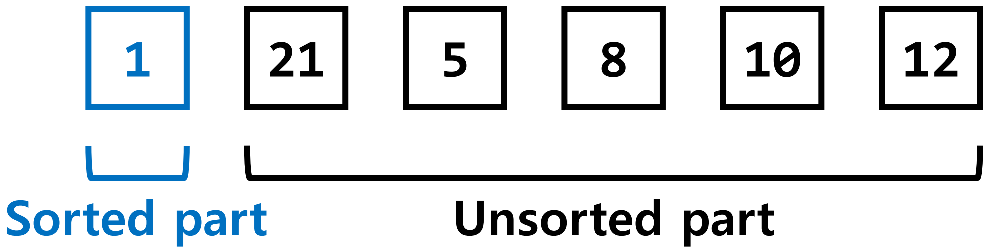
`i=0 iteration`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

`i=1 iteration`



1 21 5 8 10 12

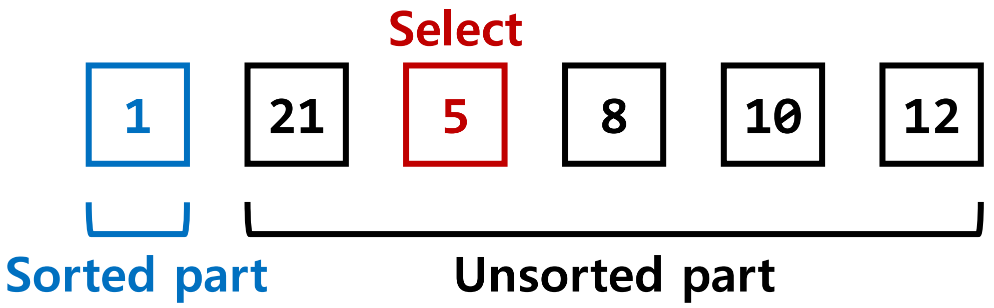
Sorted part

Unsorted part

Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

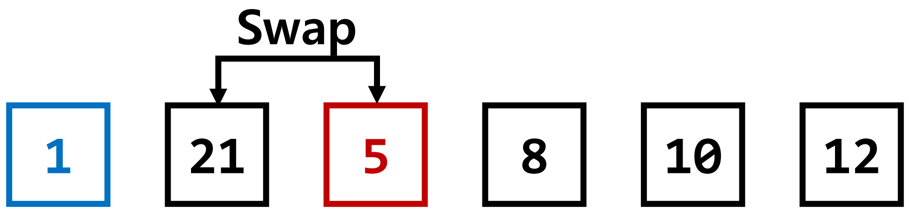
`i=1 iteration`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

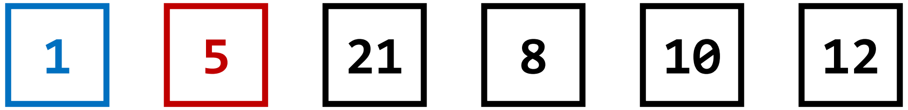
`i=1 iteration`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

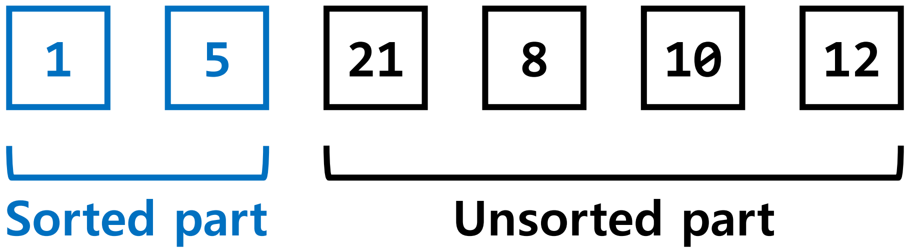
`i=1 iteration`



Selection Sort

- **Key Idea:** Select the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

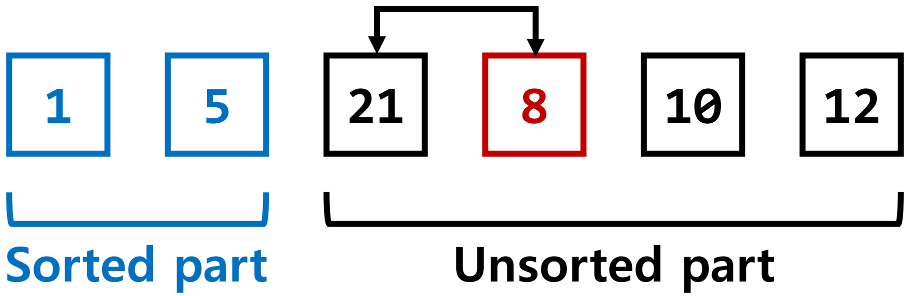
`i=2 iteration`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

`i=2 iteration`



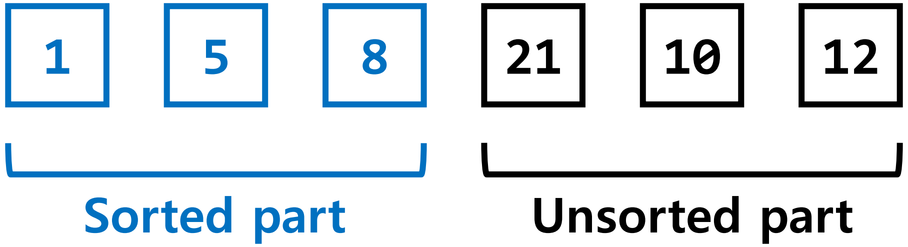
Sorted part

Unsorted part

Selection Sort

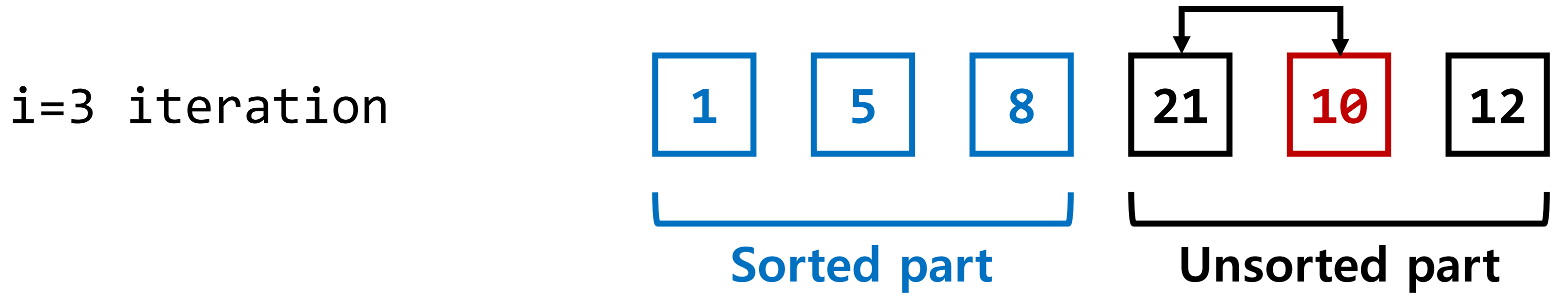
- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

`i=3 iteration`



Selection Sort

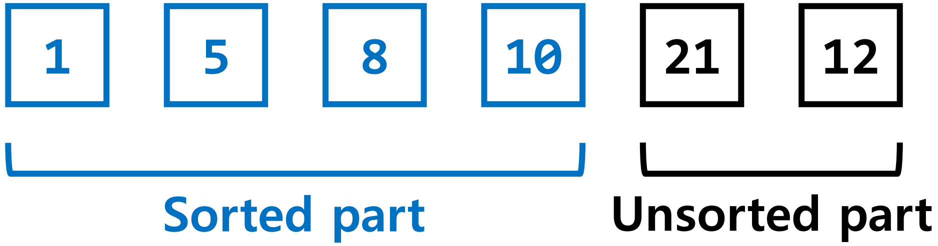
- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`



Selection Sort

- **Key Idea:** Select the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

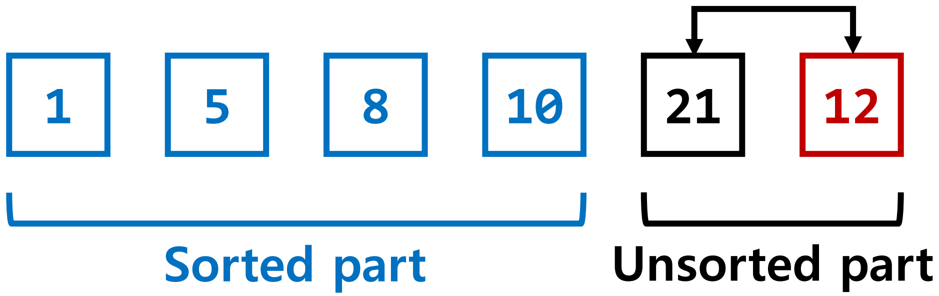
`i=4 iteration`



Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

`i=4 iteration`



1 5 8 10 21 12

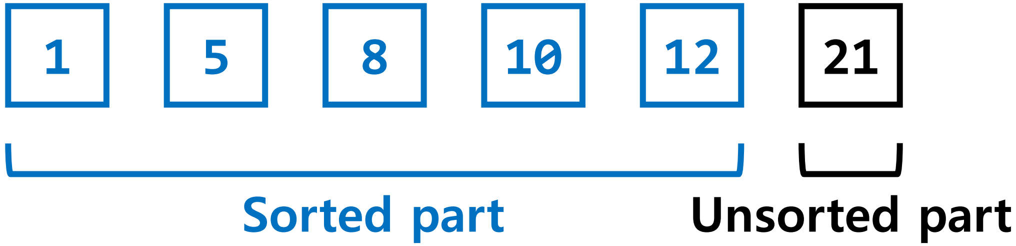
Sorted part

Unsorted part

Selection Sort

- **Key Idea:** Select the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

`i=5 iteration`

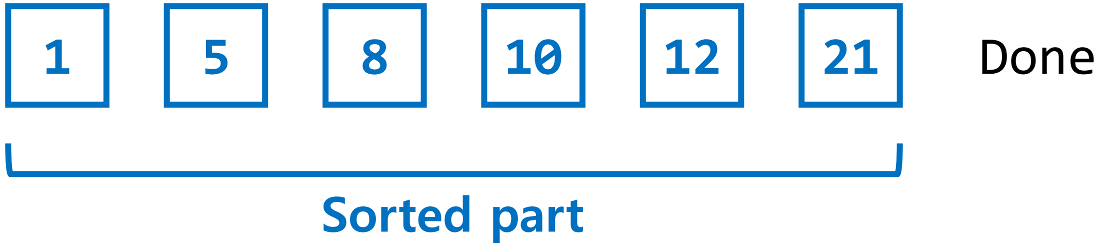


1	5	8	10	12	21

Sorted part **Unsorted part**

Selection Sort

- **Key Idea:** **Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`


1 5 8 10 12 21 Done

Sorted part

Selection Sort

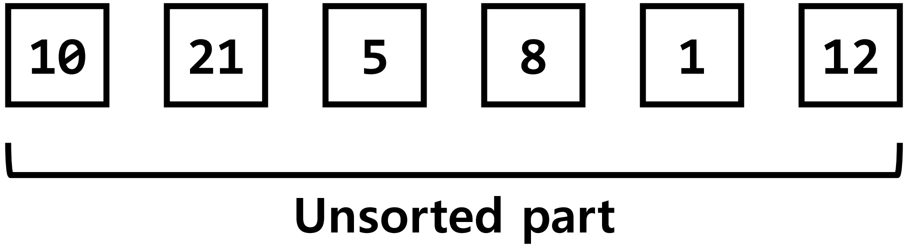
- **Key Idea: Select** the smallest item from the unsorted part
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted
     - `A[i,…,N-1]` is remaining to be sorted
  2. Select the smallest item from the unsorted part `A[i,…,N-1]`
  3. Exchange the item with the `i`-th item `A[i]`

- **Algorithm analysis**
  - Time complexity = the number of comparisons = (N-1) + (N-2) + … = $O(N^2)$
    - $O(N^2)$ even if the list is already sorted
  - Space complexity = the number of additionally required variables = O(1)

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=0 iteration`



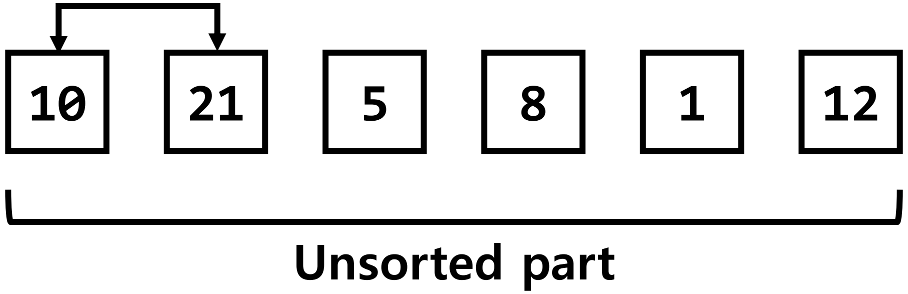
10	21	5	8	1	12

Unsorted part

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-1-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=0 iteration`



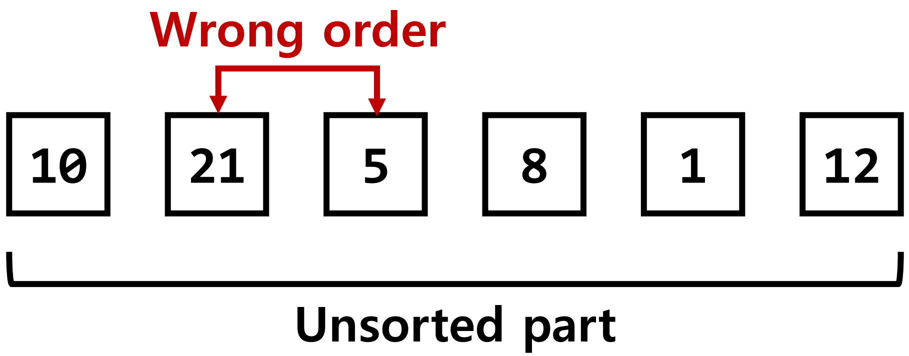
10	21	5	8	1	12

Unsorted part

Bubble Sort

- **Key Idea:** **Swap adjacent items** if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

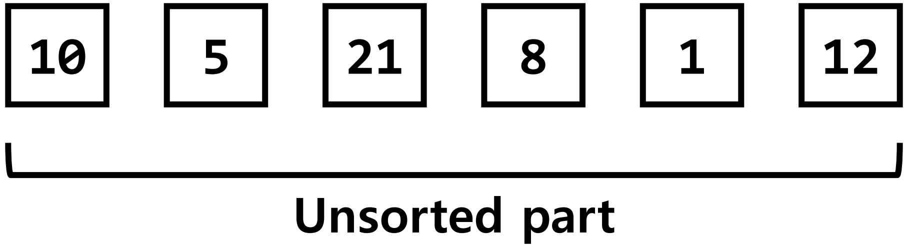
`i=0 iteration`



Bubble Sort

- **Key Idea:** **Swap adjacent items** if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=0 iteration`



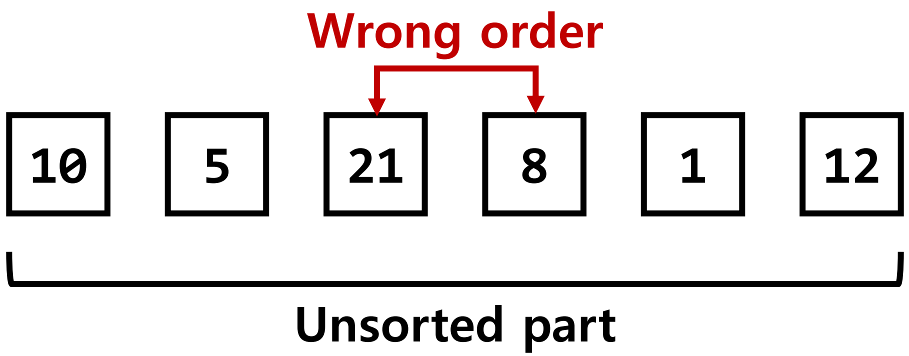
10	5	21	8	1	12

Unsorted part

Bubble Sort

- **Key Idea: Swap adjacent items** if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

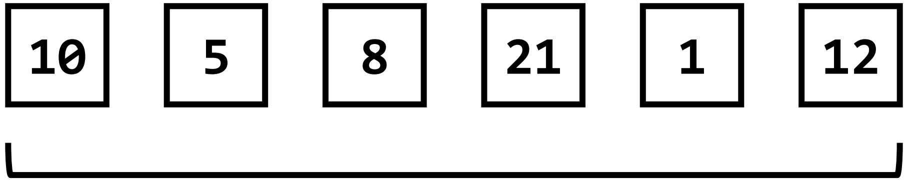
`i=0 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=0 iteration`

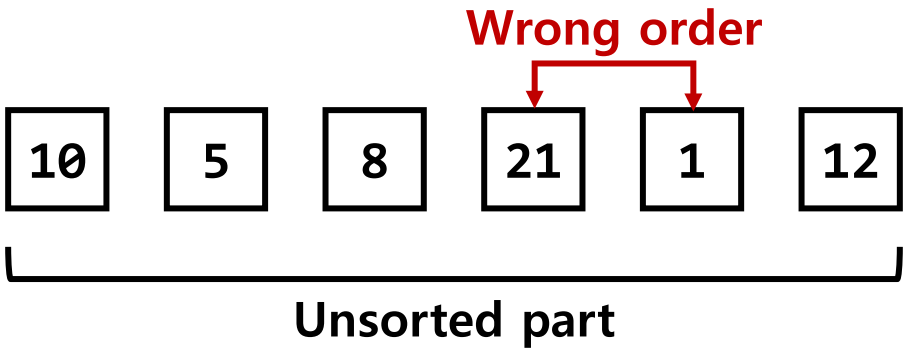


Unsorted part

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=0 iteration`



Wrong order

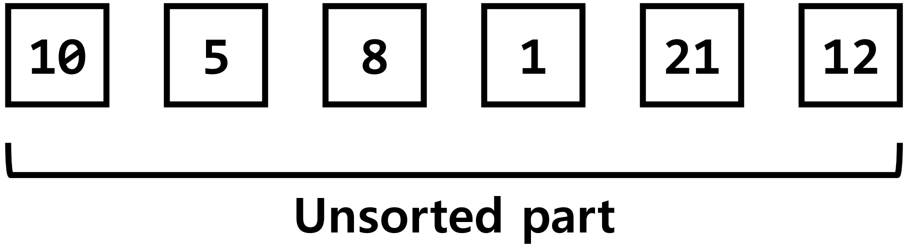
10	5	8	21	1	12

Unsorted part

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=0 iteration`



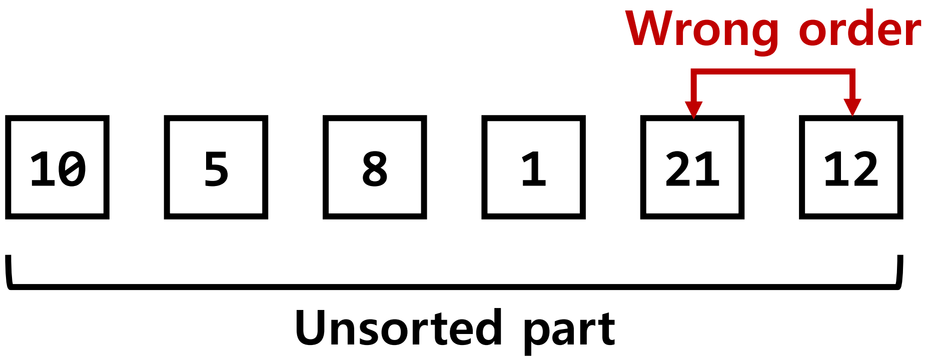
10	5	8	1	21	12

Unsorted part

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

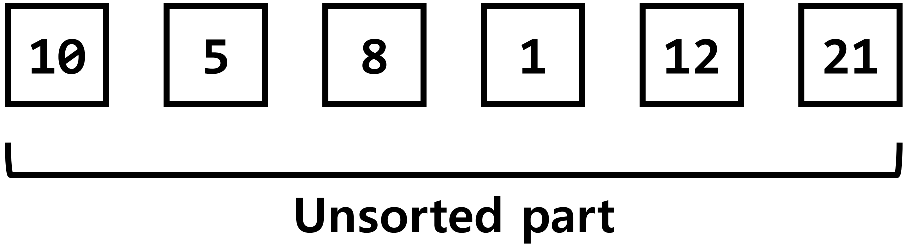
`i=0 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=0 iteration`

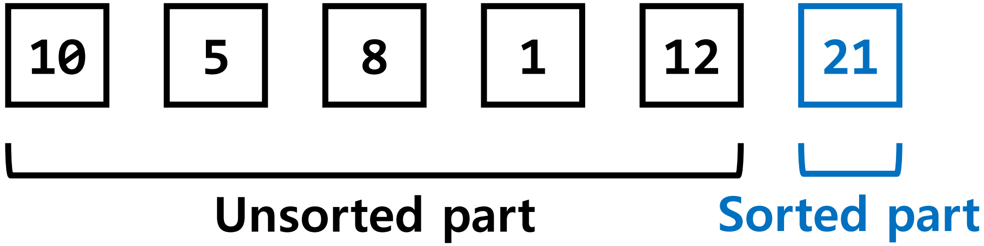


Unsorted part

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

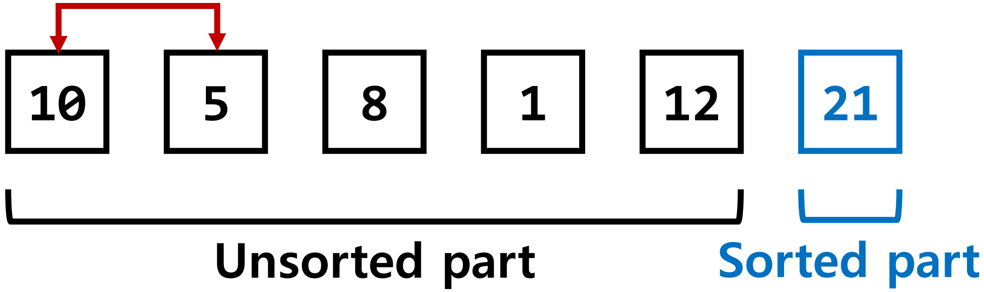
`i=1 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

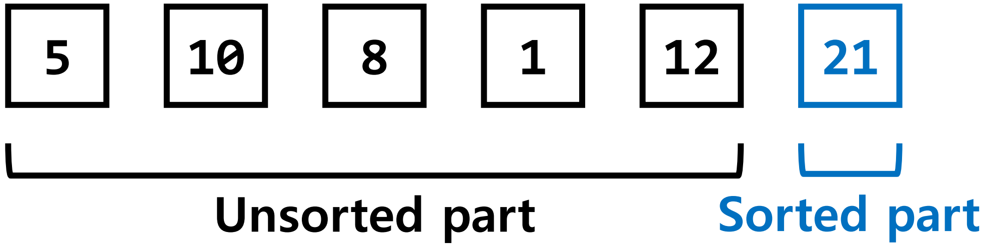
`i=1 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

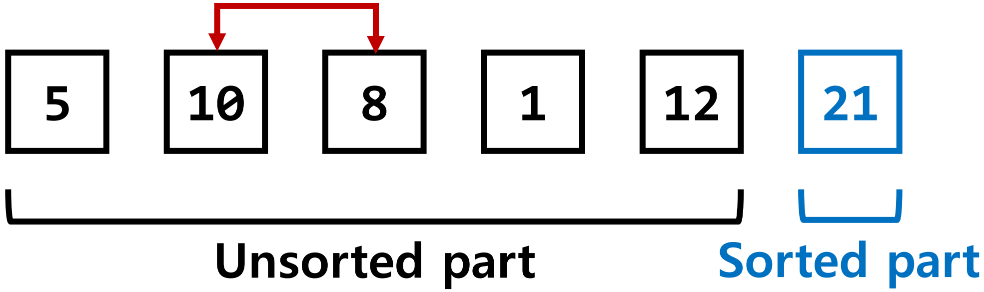
`i=1 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

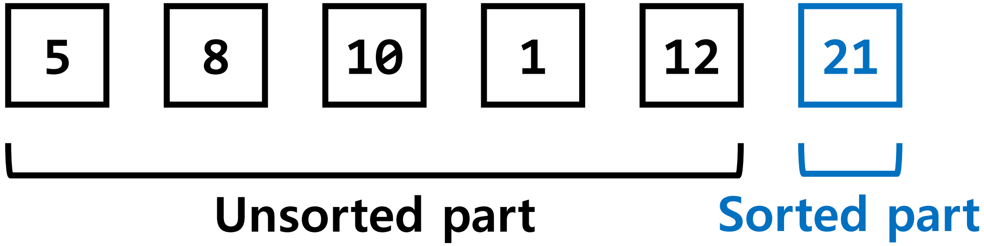
`i=1 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=1 iteration`

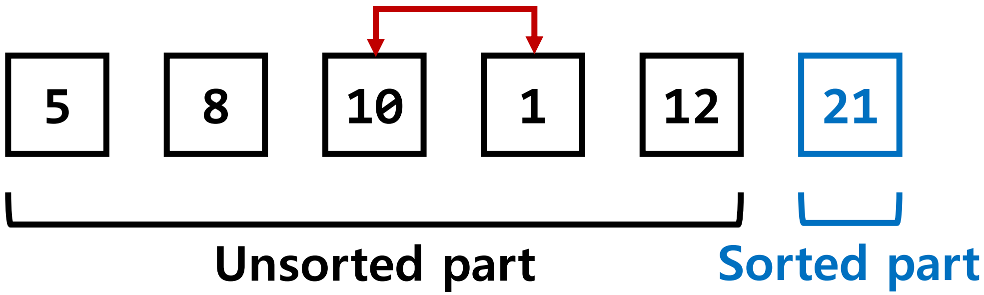


5	8	10	1	12	21

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=1 iteration`


5
8
10
1
12
21

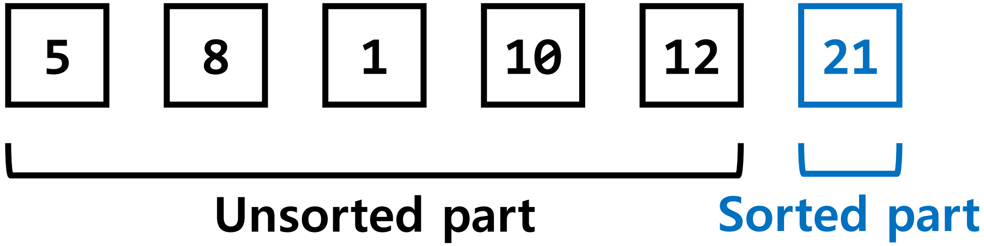
Unsorted part

Sorted part

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=1 iteration`



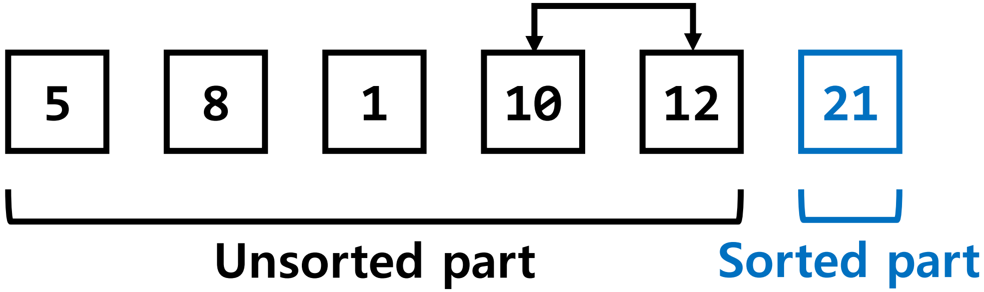
5	8	1	10	12	21

Unsorted part **Sorted part**

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

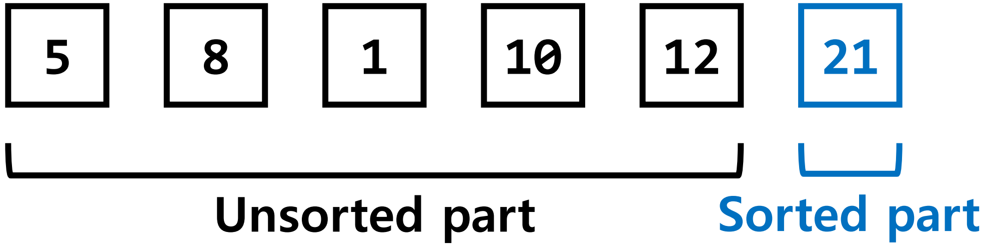
`i=1 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=1 iteration`



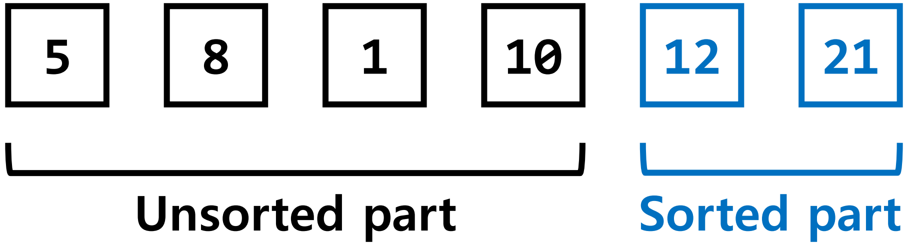
5 8 1 10 12 21

Unsorted part **Sorted part**

Bubble Sort

- **Key Idea:** **Swap adjacent items** if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

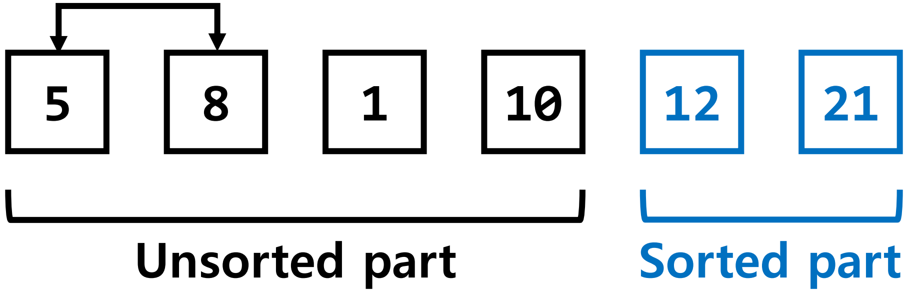
`i=2 iteration`



Bubble Sort

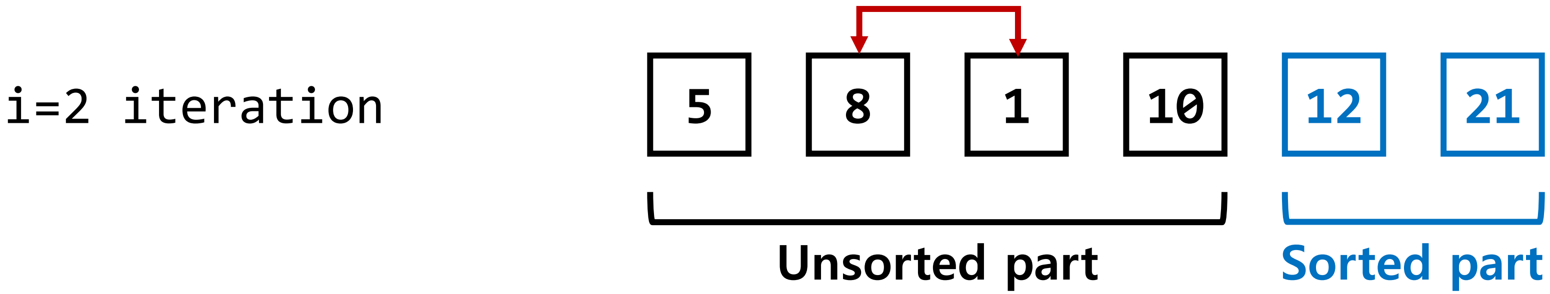
- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=2 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order



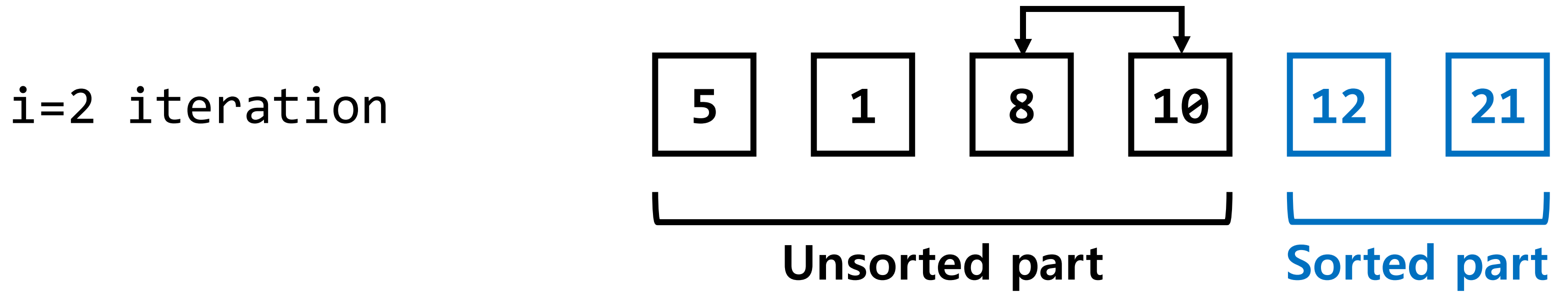
`i=2 iteration`

5	8	1	10	12	21

Unsorted part **Sorted part**

Bubble Sort

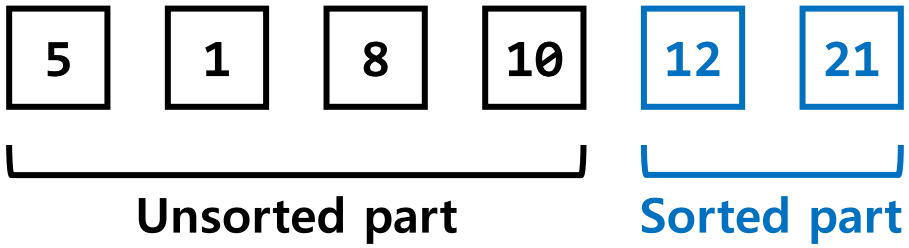
- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

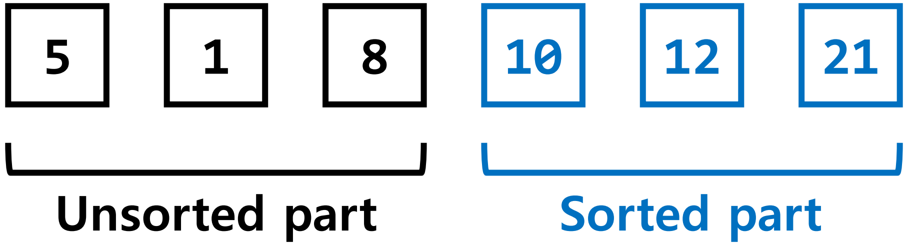
`i=2 iteration`



Bubble Sort

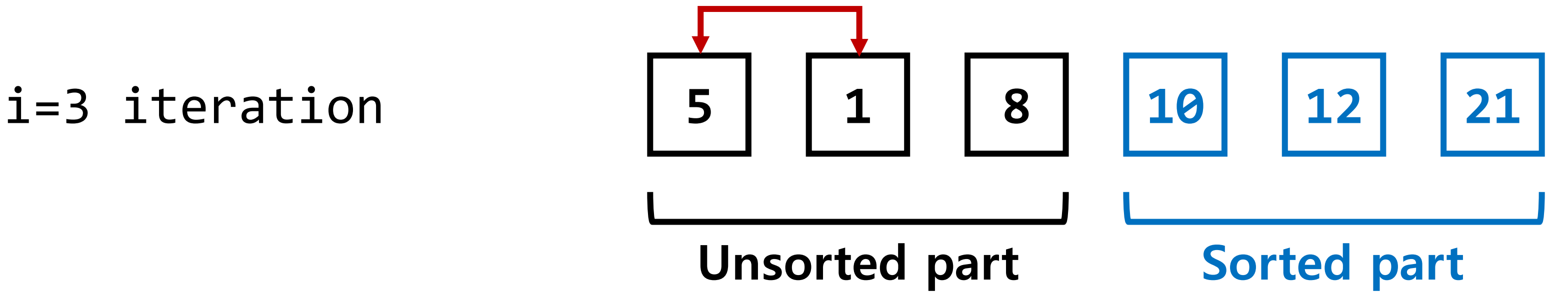
- **Key Idea:** **Swap adjacent items** if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=3 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order



`i=3 iteration`

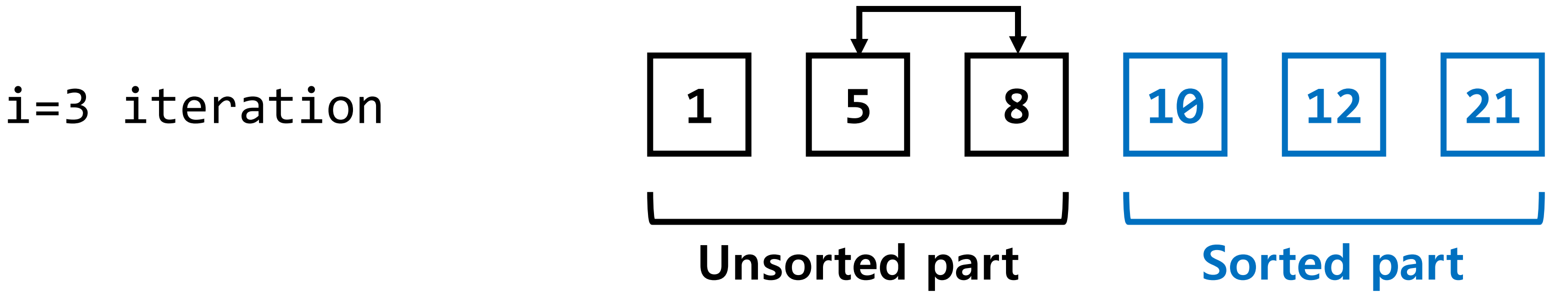
5 1 8 10 12 21

Unsorted part

Sorted part

Bubble Sort

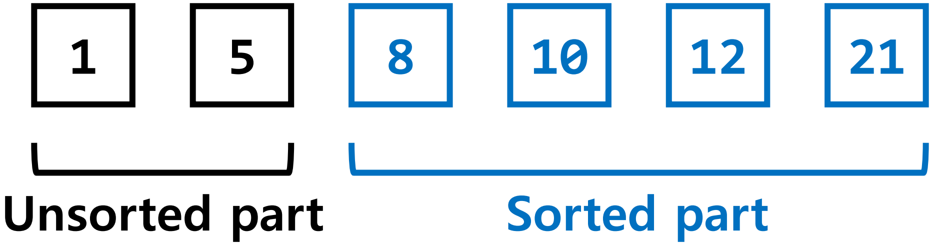
- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

`i=4 iteration`



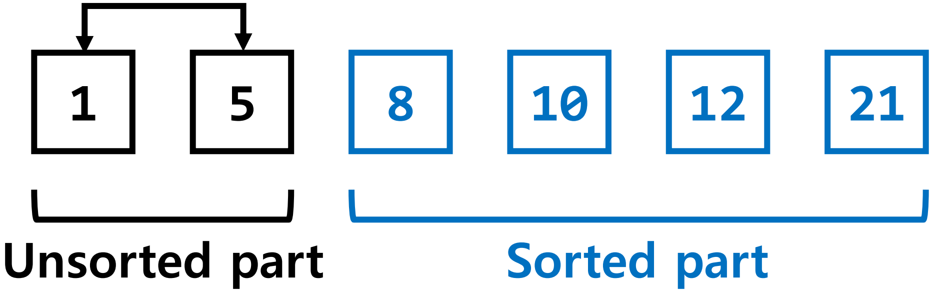
1	5	8	10	12	21

Unsorted part **Sorted part**

Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

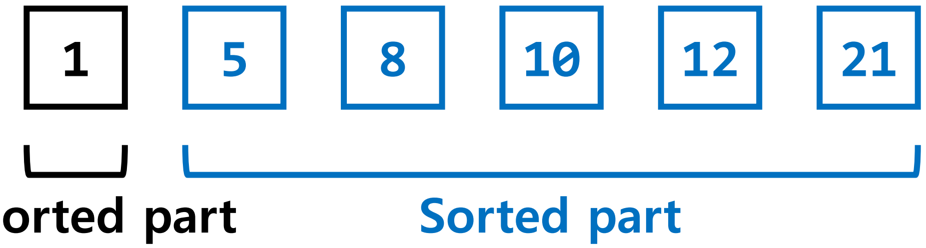
`i=4 iteration`



Bubble Sort

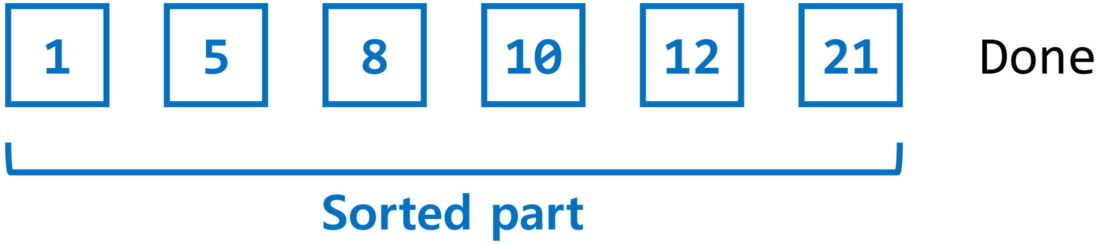
- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
    1. Divide the list of items `A[]` into two parts: `A[0,…,N-1-i]`…

`i=5 iteration`



Bubble Sort

- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-i-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order



1	5	8	10	12	21	Done

Sorted part

Bubble Sort

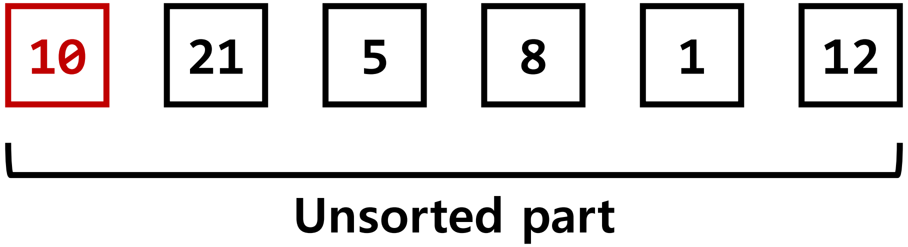
- **Key Idea:** Swap adjacent items if they are in the wrong order
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,N-1-1]` and `A[N-i,…,N-1]`
     - `A[0,…,N-i-1]` is remaining to be sorted
     - `A[N-i,…,N-1]` is already sorted
  2. From left to right, swap adjacent items if they are in the wrong order

- **Algorithm analysis**
  - Time complexity = the number of comparisons = (N-1) + (N-2) + … = $O(N^2)$
    - $O(N^2)$ even if the list is already sorted
  - Space complexity = the number of additionally required variables = O(1)

Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

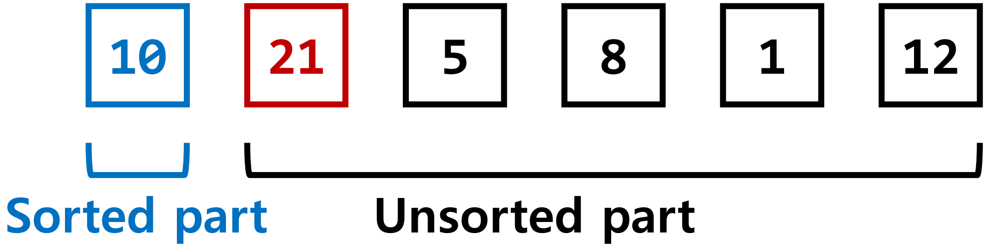
`i=0 iteration`



Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=1 iteration`



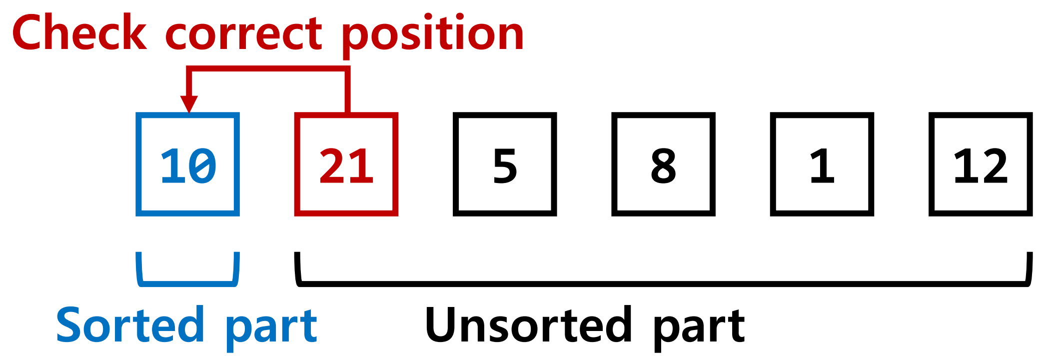
10	21	5	8	1	12

Sorted part **Unsorted part**

Insertion Sort

- **Key Idea: Insert** i-th item at the correct position
- At the i-th iteration (i = 0, 1, . . ., N-1),
  1. Divide the list of items A[ ] into two parts: A[0,…,i-1] and A[i,…,N-1]
     - A[0,…,i-1] is already sorted among A[0], A[1], …, A[i-1] items
     - A[i,…,N-1] is remaining to be sorted
  2. Insert i-th item A[i] at the correct position in A[0,…,i]

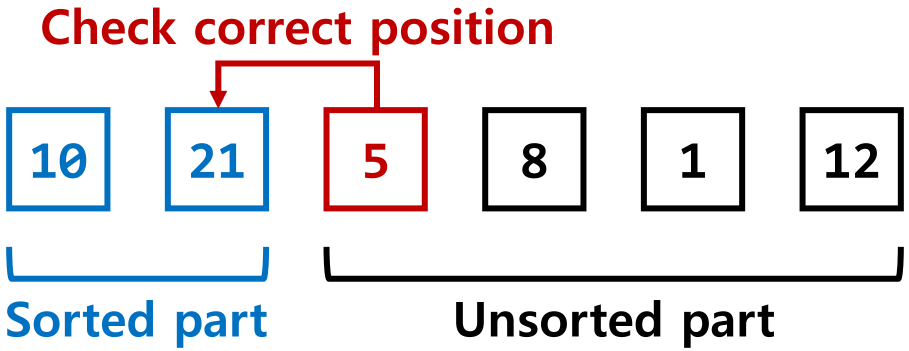
i=1 iteration



Insertion Sort

- **Key Idea: Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

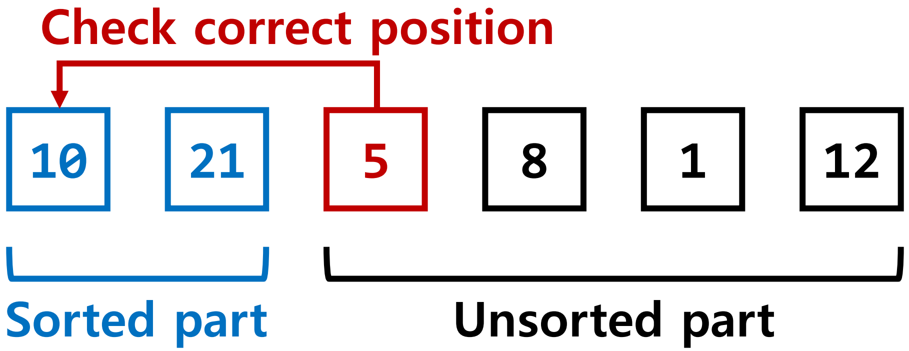
`i=2 iteration`



Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

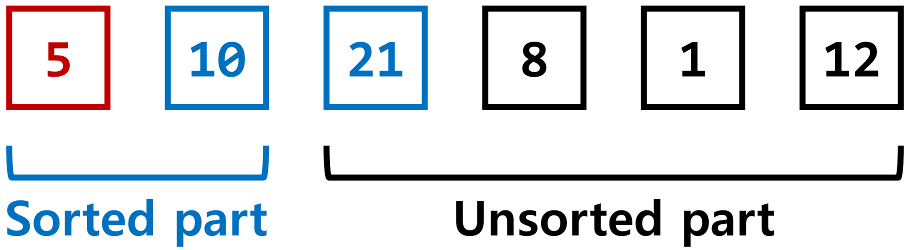
`i=2 iteration`



Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

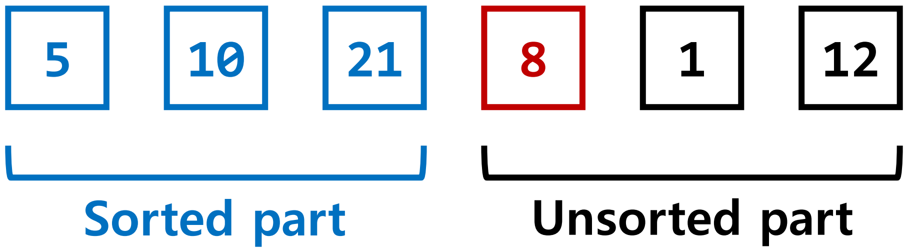
`i=2 iteration`



Insertion Sort

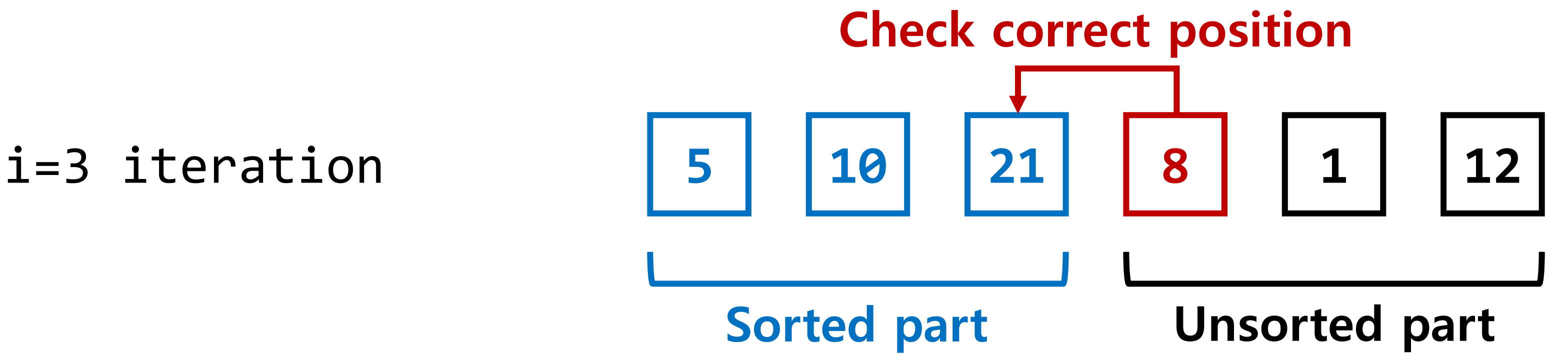
- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=3 iteration`



Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`



`i=3 iteration`

Check correct position

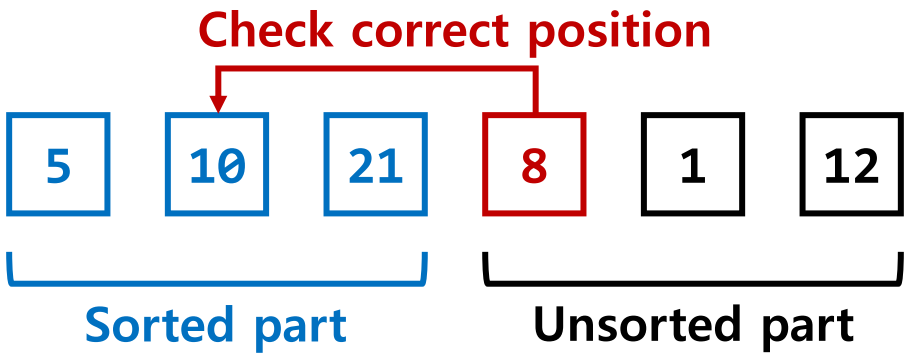
5	10	21	8	1	12

Sorted part **Unsorted part**

Insertion Sort

- **Key Idea: Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=3 iteration`



Check correct position

5 10 21 8 1 12

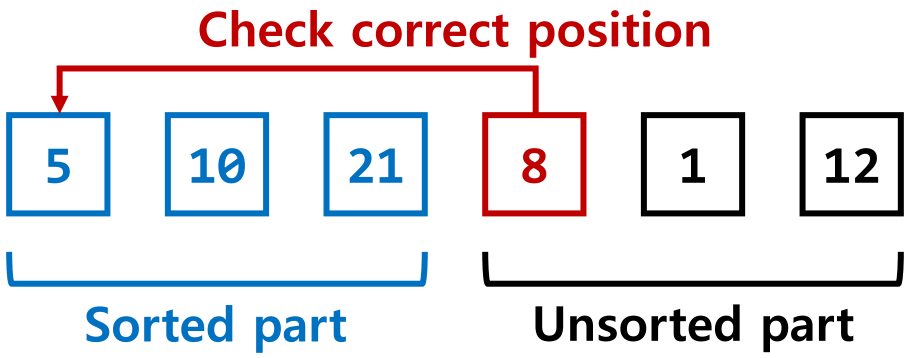
Sorted part

Unsorted part

Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=3 iteration`



Check correct position

5	10	21	8	1	12

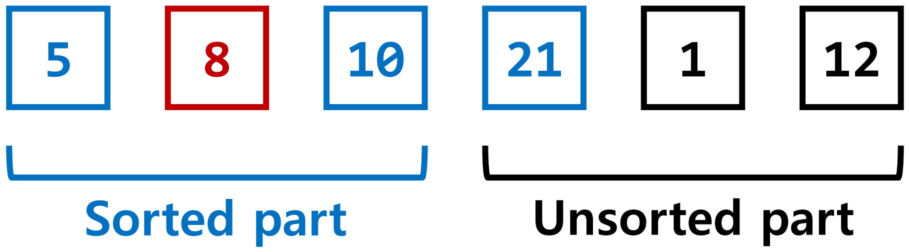
Sorted part

Unsorted part

Insertion Sort

- **Key Idea: Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=3 iteration`



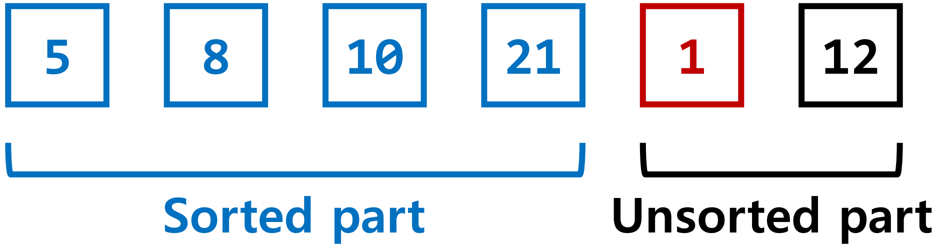
5	8	10	21	1	12

Sorted part **Unsorted part**

Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=4 iteration`



5 8 10 21 1 12

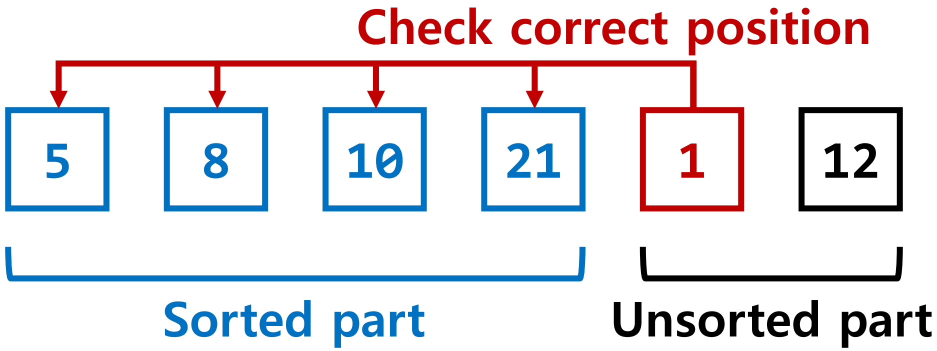
Sorted part

Unsorted part

Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

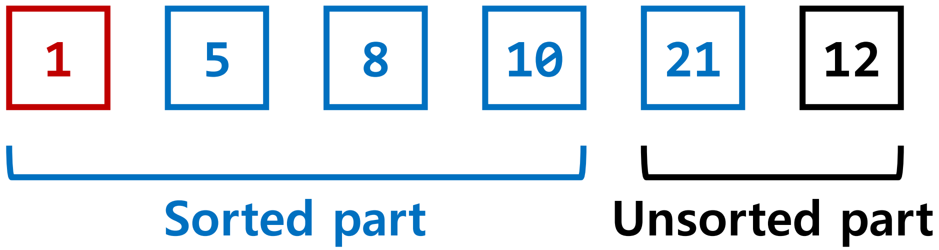
`i=4 iteration`



Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
    1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
        - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
        - `A[i,…,N-1]` is remaining to be sorted
    2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

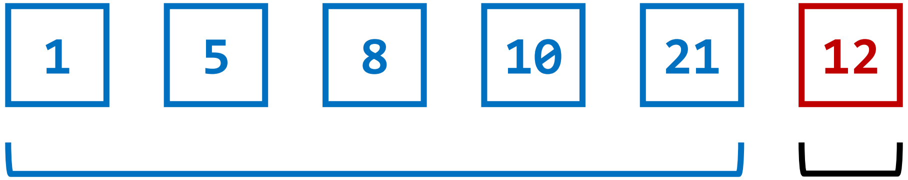
`i=4 iteration`



Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=5 iteration`

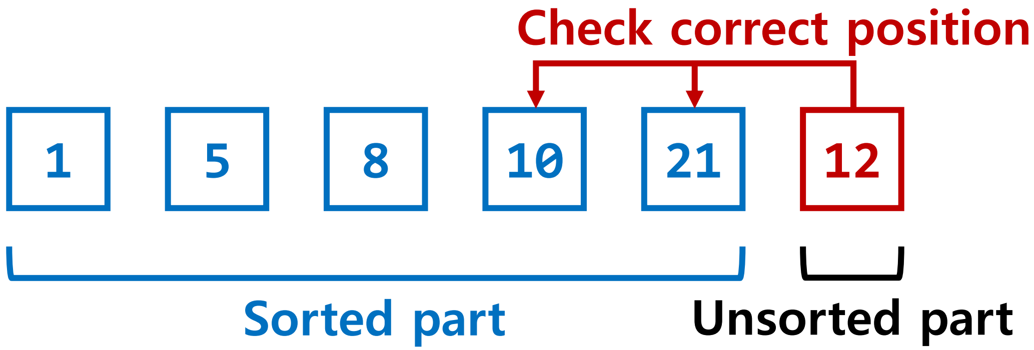


Sorted part **Unsorted part**

Insertion Sort

- **Key Idea: Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i` = `0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

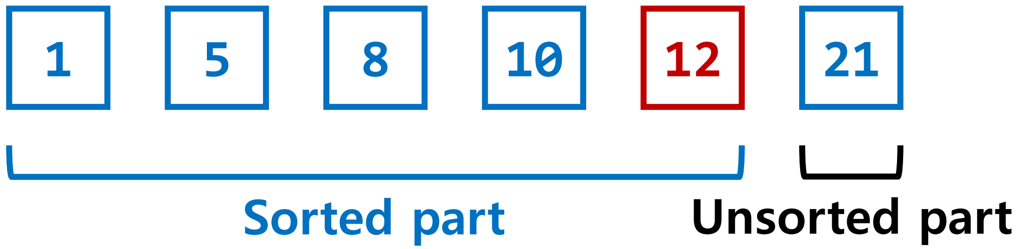
`i=5 iteration`



Insertion Sort

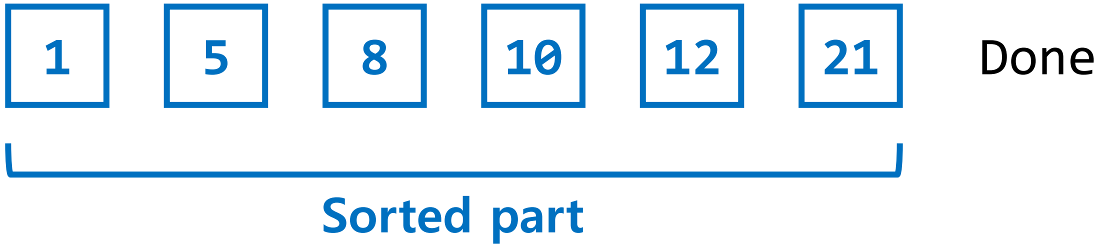
- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[ ]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

`i=5 iteration`



Insertion Sort

- **Key Idea:** **Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`



Insertion Sort

- **Key Idea: Insert** `i`-th item at the correct position
- At the `i`-th iteration (`i = 0`, `1`, `. . .`, `N-1`),
  1. Divide the list of items `A[]` into two parts: `A[0,…,i-1]` and `A[i,…,N-1]`
     - `A[0,…,i-1]` is already sorted among `A[0]`, `A[1]`, …, `A[i-1]` items
     - `A[i,…,N-1]` is remaining to be sorted
  2. Insert `i`-th item `A[i]` at the correct position in `A[0,…,i]`

- **Algorithm analysis**
  - Time complexity = the number of comparisons = (N-1) + (N-2) + … = $O(N^2)$
    - O(N) if the list is already sorted
    - It is efficient when the items are already mostly sorted
  - Space complexity = the number of additionally required variables = O(1)

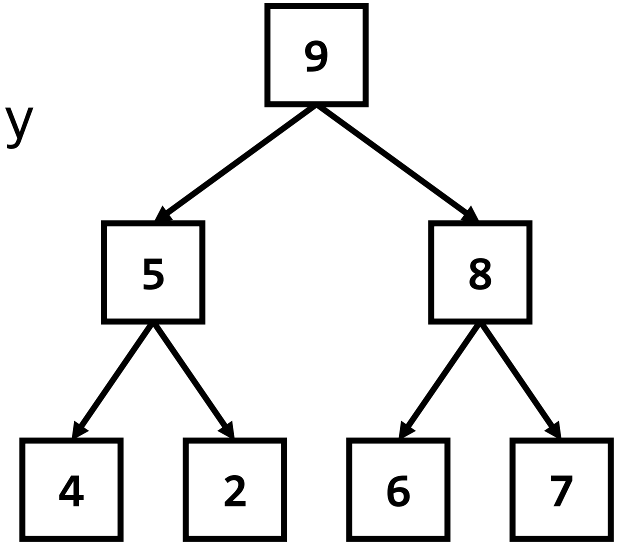
Heap Sort

- **Heap** is a complete binary tree satisfying …
  - Each node has its own **priority** (like key in BSTs)
  - Any node has a higher priority than its children:

`priority(parent) >= priority(child)`

- Why **heap** structure is useful for sorting?
  - Its root node has the highest priority
  - → You can select the minimum (or maximum) item efficiently

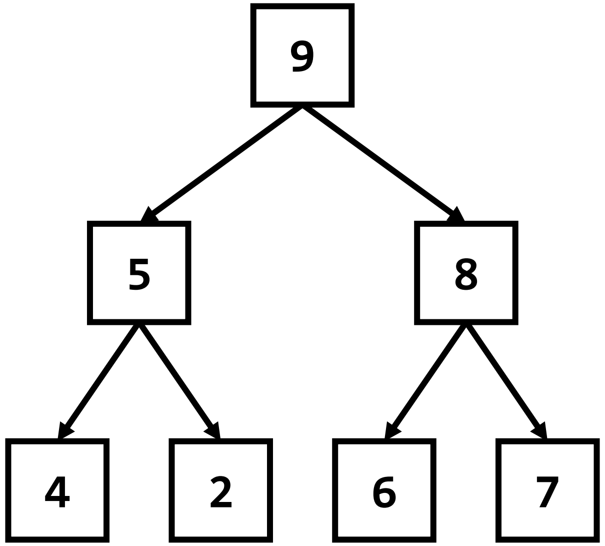
  - It is implemented by an array structure
  - → You can directly use the array of input items



Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



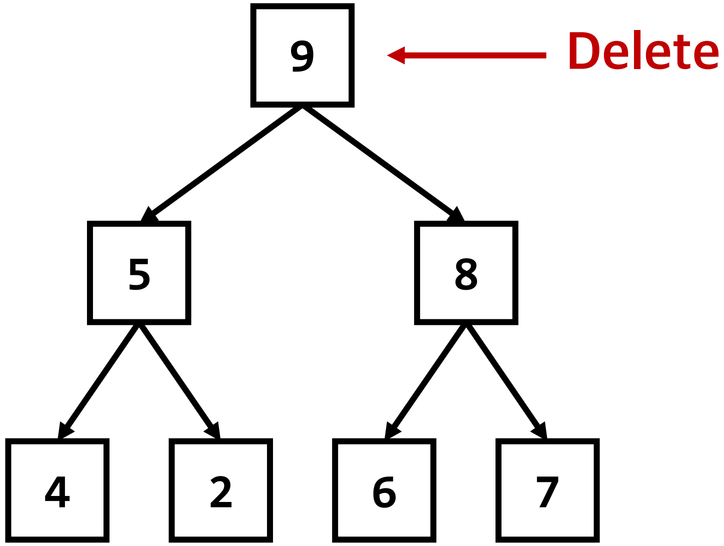
Array Representation

9	5	8	4	2	6	7

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure

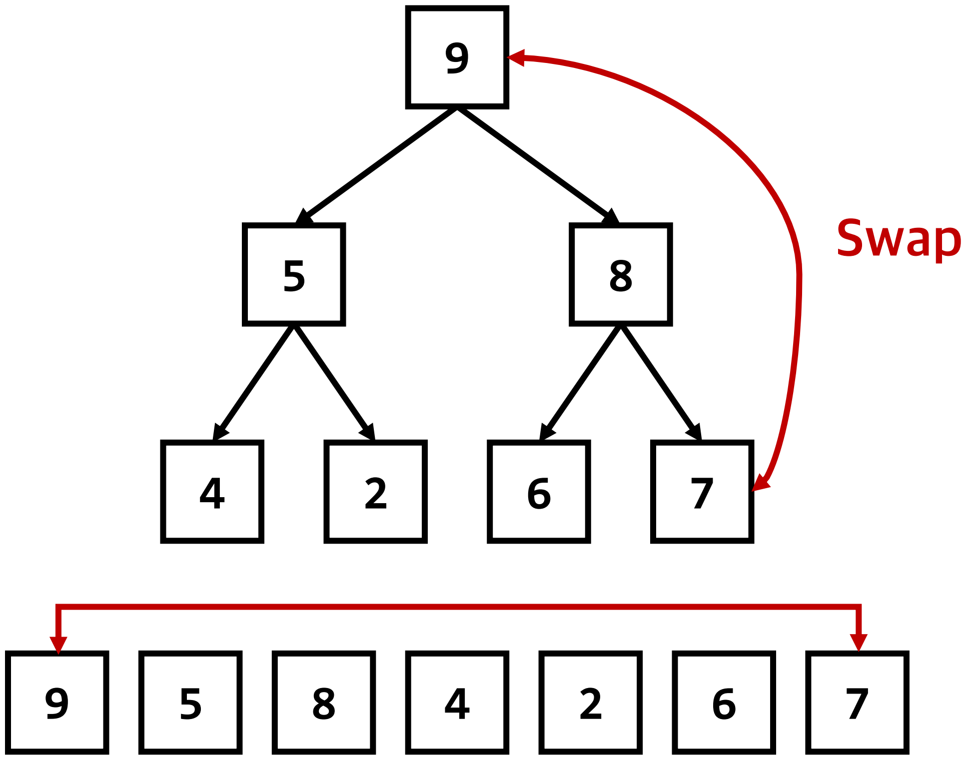


Array Representation

9	5	8	4	2	6	7

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position



Heap Structure

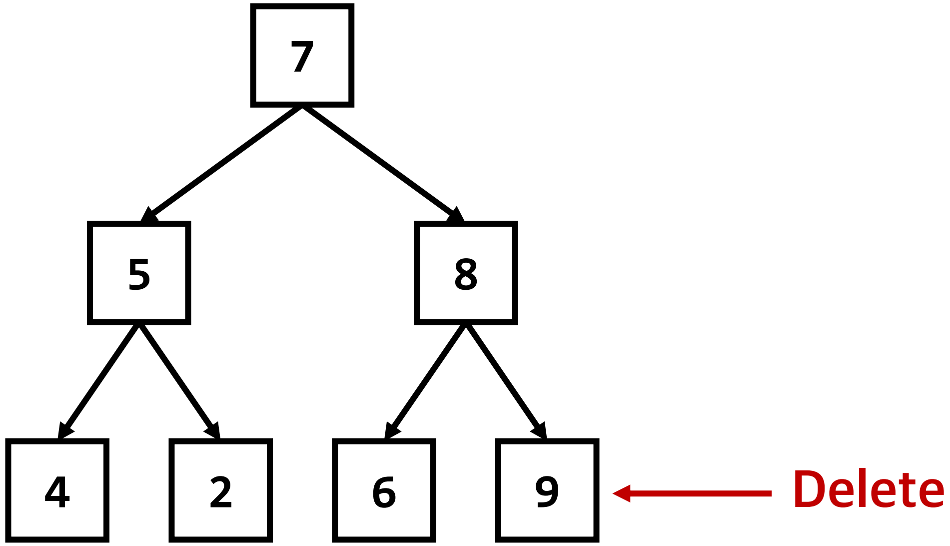
Array Representation

9	5	8	4	2	6	7

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure

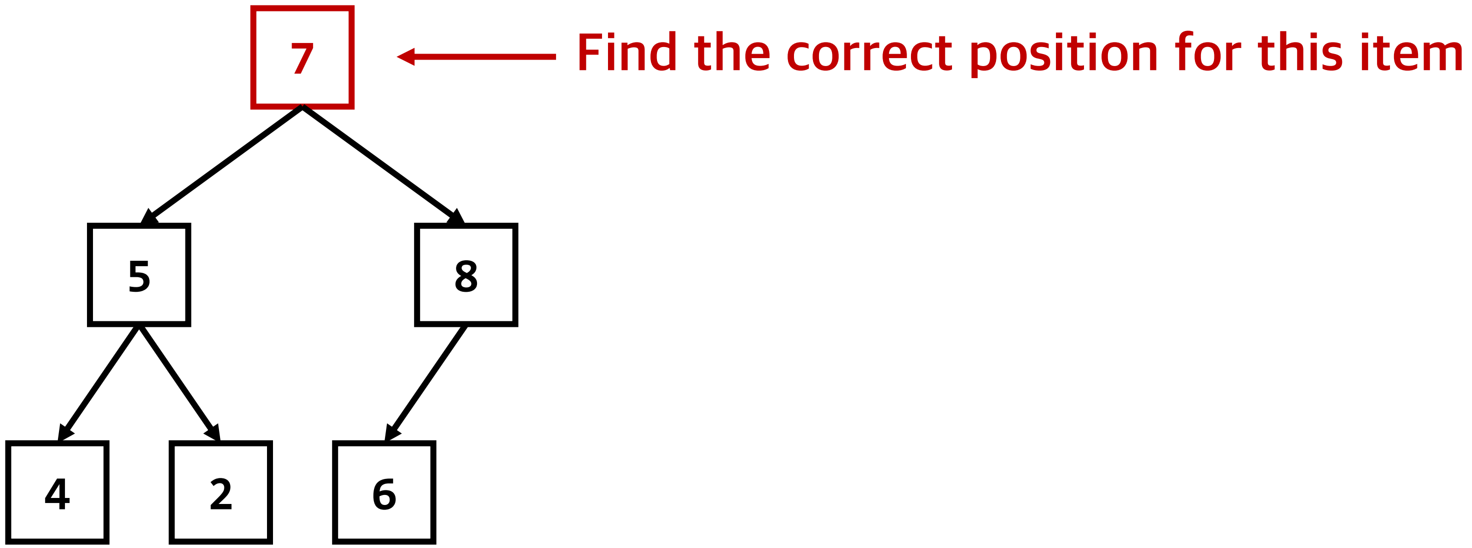


Array Representation

7	5	8	4	2	6	9

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position



Heap Structure

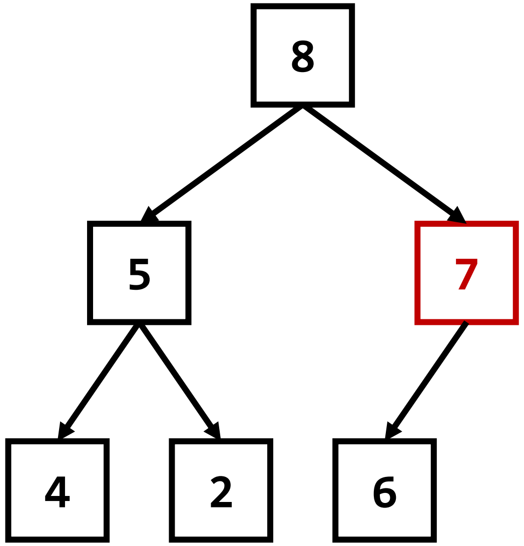
Array Representation

7	5	8	4	2	6	9

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



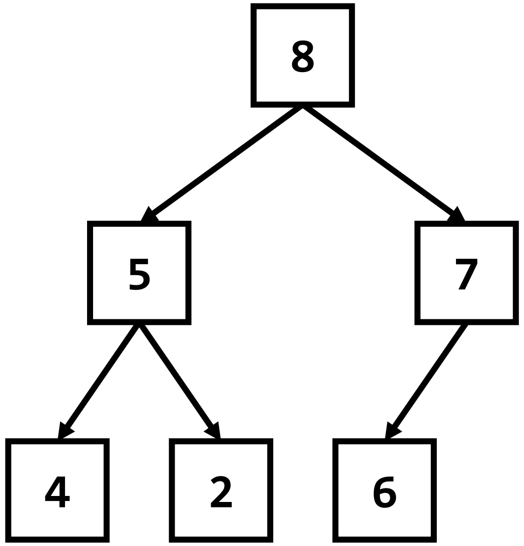
Array Representation

8	5	7	4	2	6	9

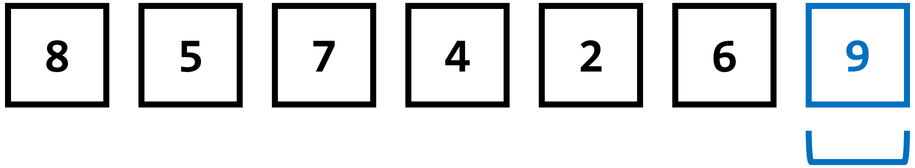
Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



Array Representation

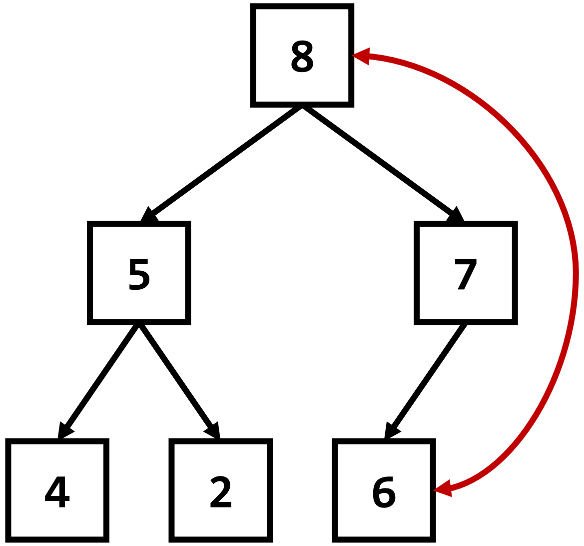


Sorted part

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



Array Representation

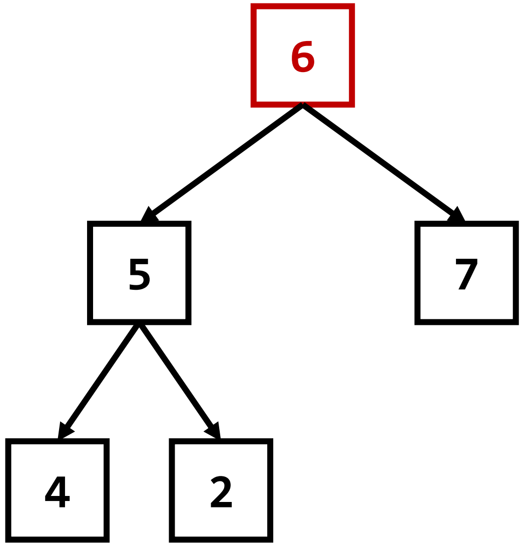
8	5	7	4	2	6	9

Sorted part

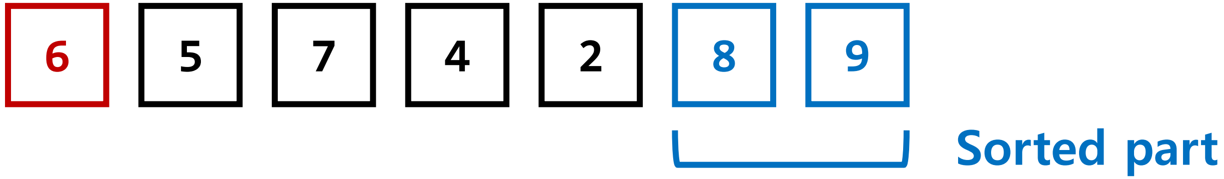
Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



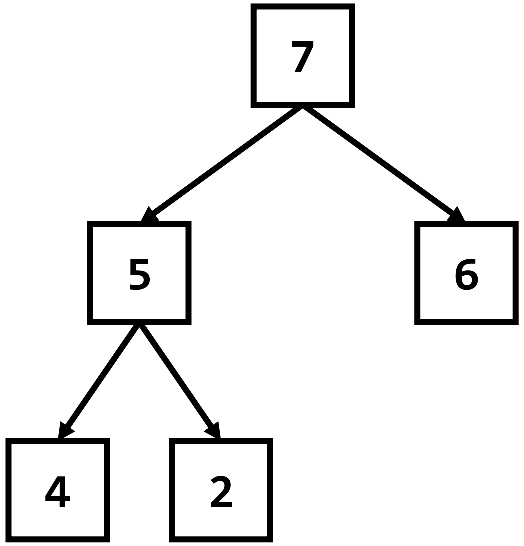
Array Representation



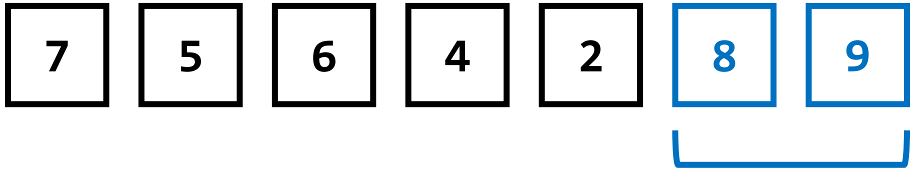
Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



Array Representation

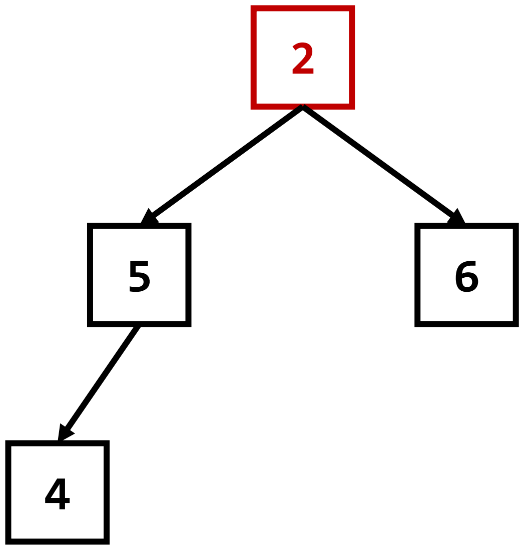


Sorted part

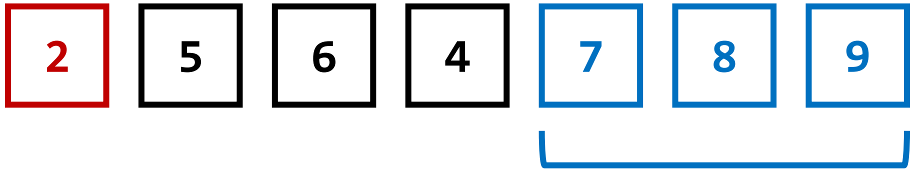
Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



Array Representation

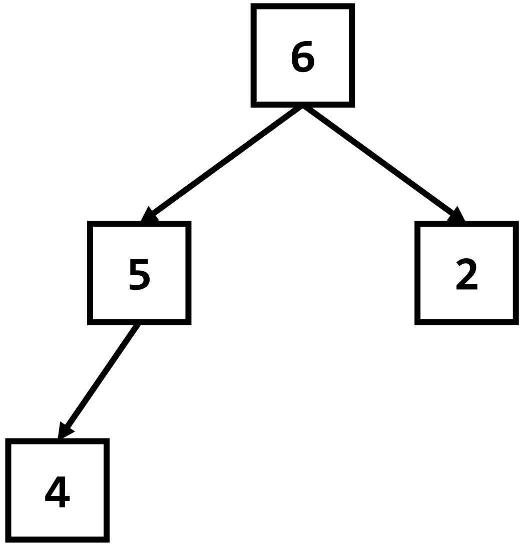


Sorted part

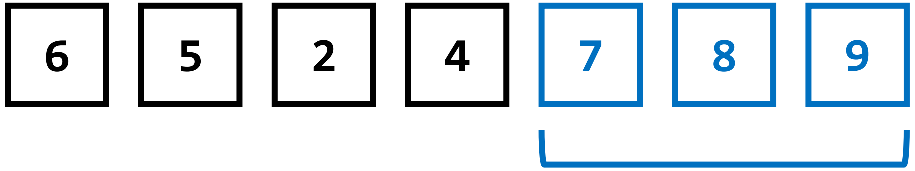
Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



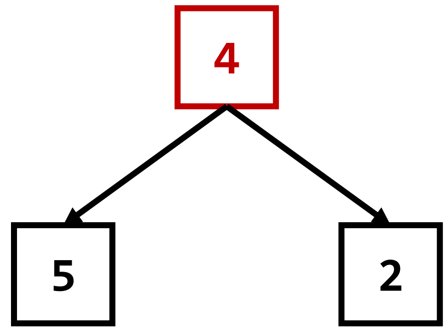
Array Representation



Sorted part

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position



Heap Structure

Array Representation

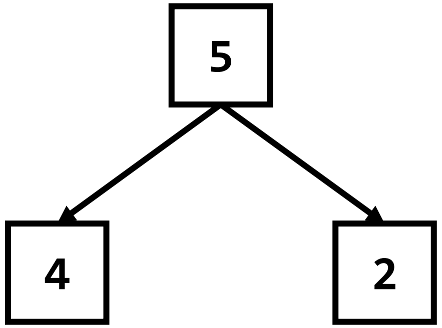
4	5	2	6	7	8	9

Sorted part

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



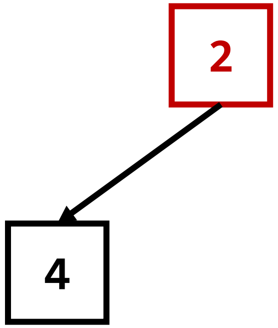
Array Representation

Sorted part

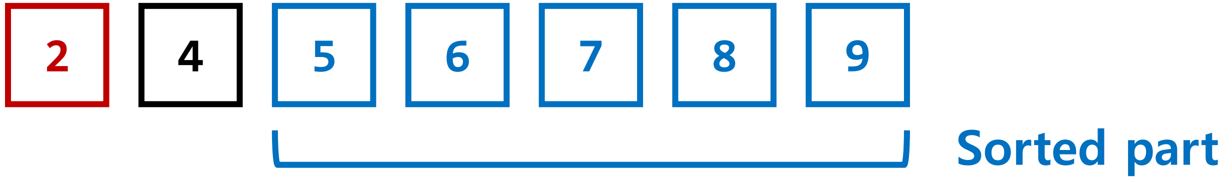
Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



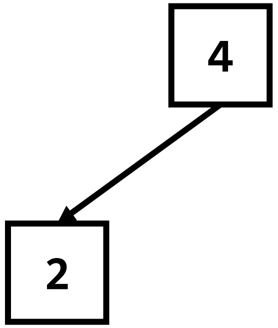
Array Representation



Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure



Array Representation

4	2	5	6	7	8	9

Sorted part

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

2

Heap Structure

Array Representation	2	4	5	6	7	8	9

$\underbrace{4\quad 5\quad 6\quad 7\quad 8\quad 9}_{\text{Sorted part}}$

Heap Sort

- How to use **heap** structure for sorting?
  - Construct **MaxHeap** using all items
  - For each iteration, delete the root item and put it to the last position

Heap Structure

Array Representation

2	4	5	6	7	8	9

Sorted part

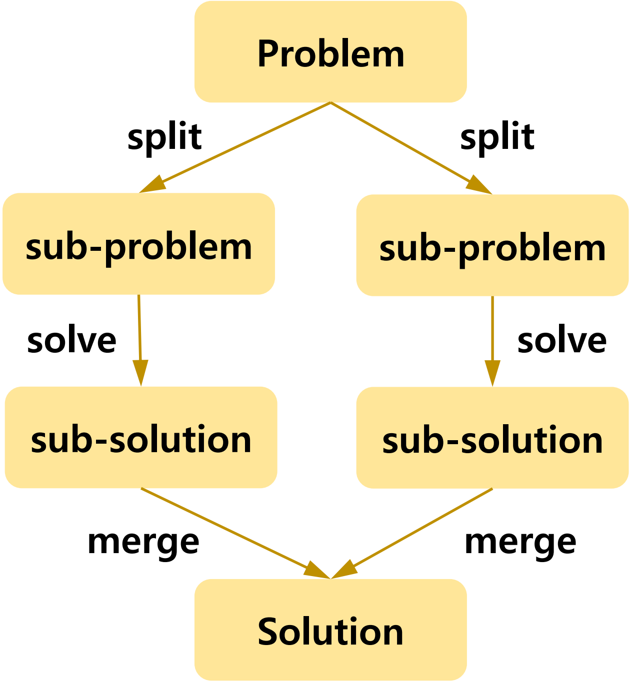
Heap Sort

- How to use **heap** structure for sorting?
    - Construct **MaxHeap** using all items
    - For each iteration, delete the root item and put it to the last position

- **Algorithm analysis**
    - Time complexity = the number of comparisons = log(N-1) + log(N-2) + … = O(NlogN)
    - Space complexity = the number of additionally required variables = O(1)

Quick & Merge Sorts

- Quick and Merge Sorts are based on Divide & Conquer (D&C) Paradigm
  - **Breaking down a problem into two or more sub-problems** of the same type
  - **Solutions to the sub-problems are combined to be a solution** to the original one



Problem

split split

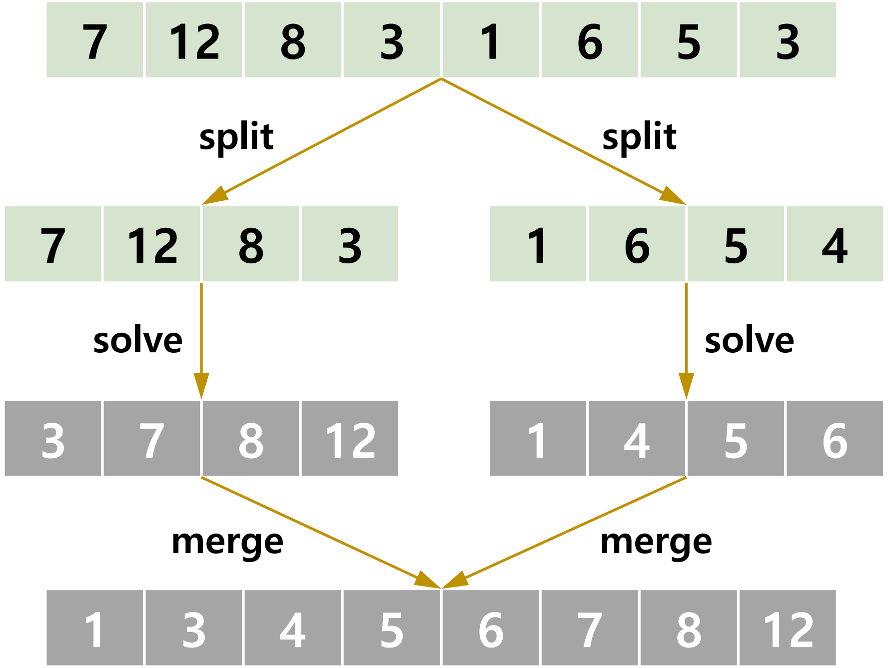
sub-problem **sub-problem**

solve solve

sub-solution **sub-solution**

merge merge

Solution



7 12 8 3 1 6 5 3

split split

7 12 8 3 | 1 6 5 4

solve solve

3 7 8 12 | 1 4 5 6

merge merge

1 3 4 5 6 7 8 12

Quick Sort

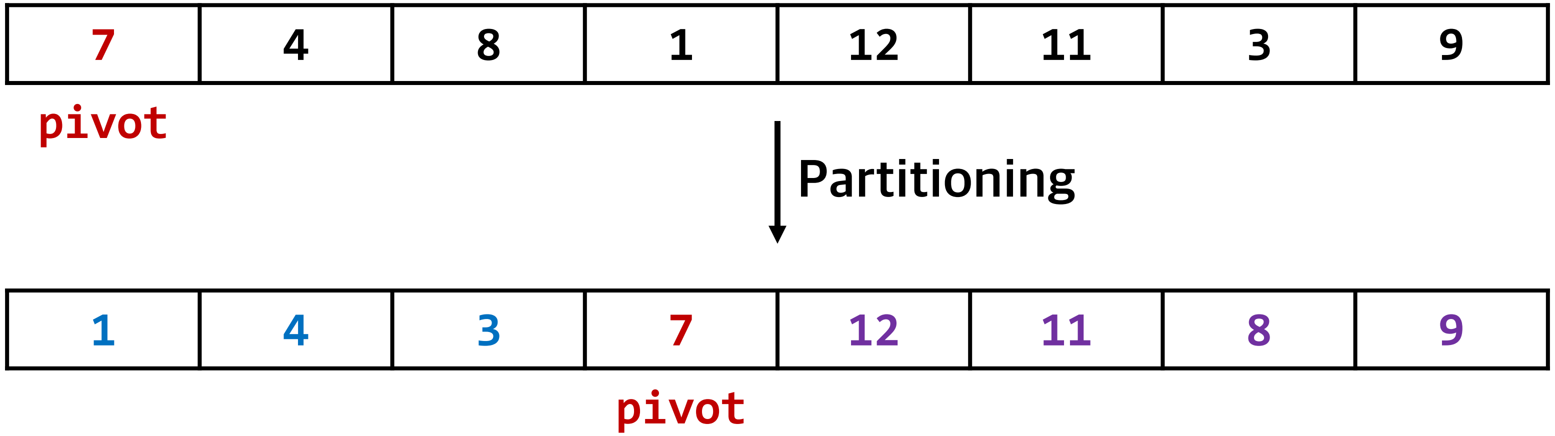
- **Key Idea:** Select a pivot and split items into two partitions using the pivot
    - **Pivot selection**: Pick an element, called a pivot, from the list

7	4	8	1	12	11	3	9

pivot

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - **Pivot selection**: Pick an element, called a pivot, from the list
  - **Partitioning**: reorder the list with the pivot
    - The elements less than the pivot come before the pivot
    - The element greater than the pivot come after the pivot

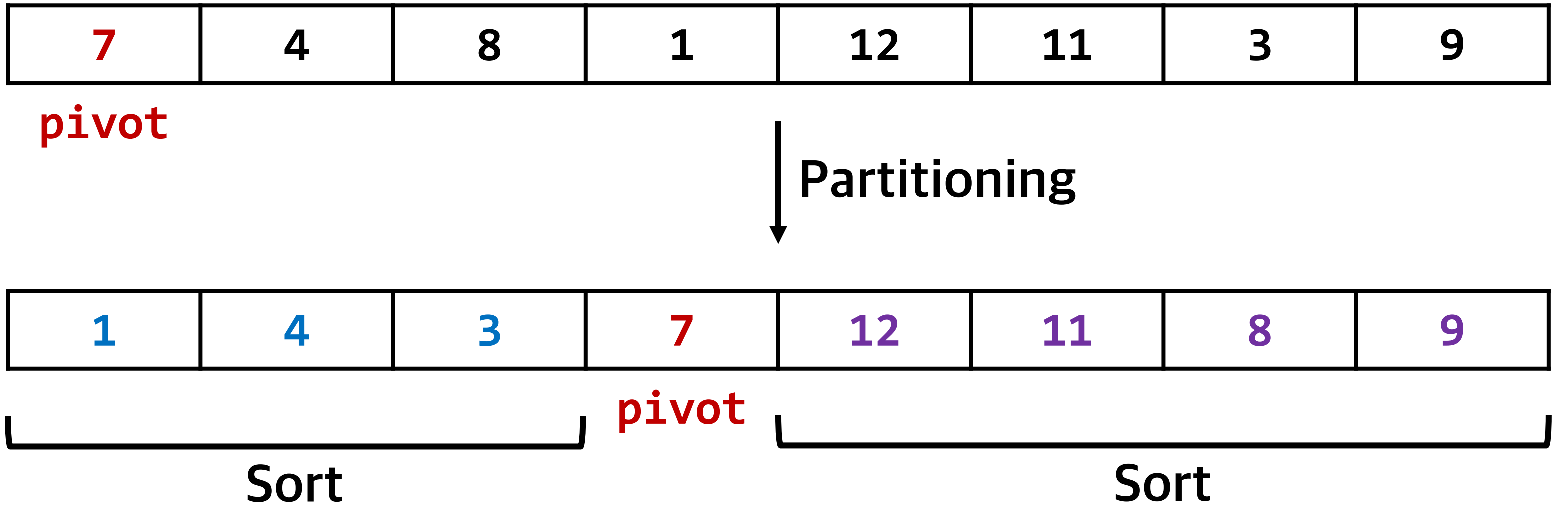


7	4	8	1	12	11	3	9

1	4	3	7	12	11	8	9

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - **Pivot selection**: Pick an element, called a pivot, from the list
  - **Partitioning**: reorder the list with the pivot
    - The elements less than the pivot come before the pivot
    - The element greater than the pivot come after the pivot



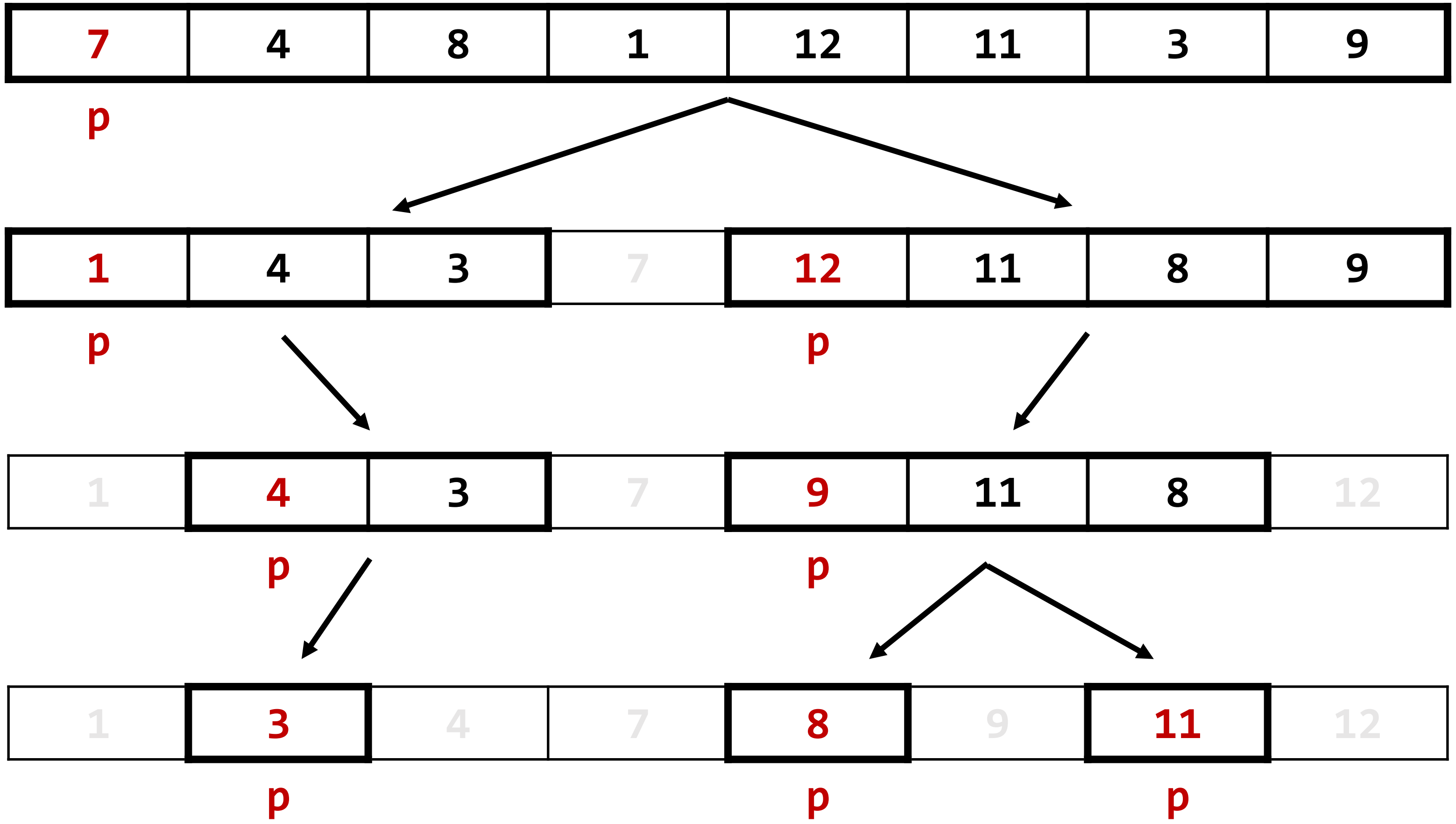
7	4	8	1	12	11	3	9

1	4	3	7	12	11	8	9

- **Recursively apply** the above steps to the sub-lists

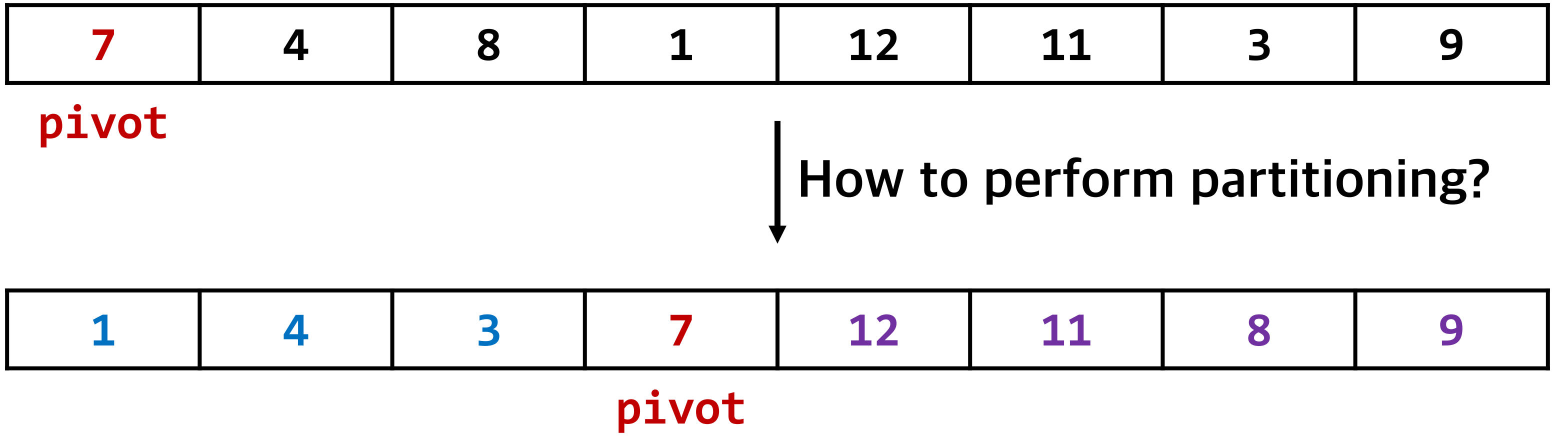
Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot



Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?



7	4	8	1	12	11	3	9

pivot

How to perform partitioning?

1	4	3	7	12	11	8	9

pivot

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)

7	4	8	1	12	11	3	9
pivot	L						

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)

7	4	8	1	12	11	3	9
pivot		L					

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)

7	4	8	1	12	11	3	9
pivot		L					R

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)

7	4	8	1	12	11	3	9
pivot		L				R	

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if **L** is placed before **R**, otherwise partitioning is completed after swap pivot and **R**

7	**4**	**8**	**1**	**12**	**11**	**3**	**9**
pivot		**L**				**R**	

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if **L** is placed before **R**, otherwise partitioning is completed after swap pivot and **R**

7	4	3	1	12	11	8	9
pivot		L				R	

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if L is placed before R, otherwise partitioning is completed after swap pivot and R

7	4	3	1	12	11	8	9
pivot			L			R	

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if L is placed before R, otherwise partitioning is completed after swap pivot and R

7	4	3	1	12	11	8	9
pivot				L		R	

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if L is placed before R, otherwise partitioning is completed after swap pivot and R

7	4	3	1	12	11	8	9
pivot				L	R		

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if L is placed before R, otherwise partitioning is completed after swap pivot and R

7	4	3	1	12	11	8	9
pivot				L R			

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if L is placed before R, otherwise partitioning is completed after swap pivot and R

7	4	3	1	12	11	8	9
pivot			R	L			

Quick Sort

- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - Select the left-most element as the pivot
  - How to perform partitioning?
    - Select **the left-most item L greater** than the pivot (check items from left to right)
    - Select **the right-most item R less** than the pivot (check items from right to left)
    - Swap them if **L** is placed before **R**, otherwise partitioning is completed after swap pivot and **R**

1	**4**	**3**	**7**	**12**	**11**	**8**	**9**

  - **Recursively apply** the above steps to the sub-lists

Quick Sort

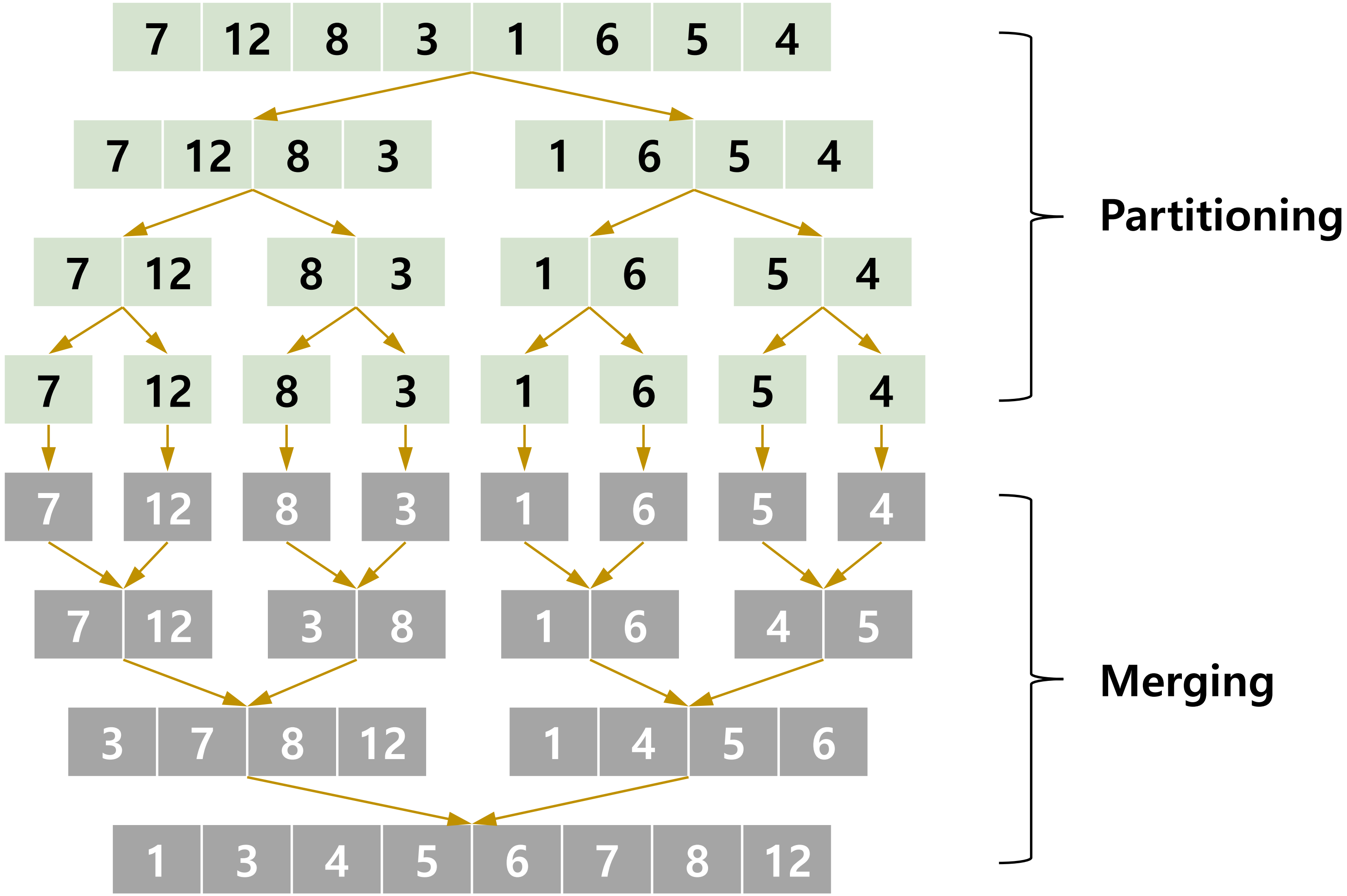
- **Key Idea:** Select a pivot and split items into two partitions using the pivot
  - **Pivot selection**: Pick an element, called a pivot, from the list
  - **Partitioning**: reorder the list with the pivot
    - The elements less than the pivot come before the pivot
    - The element greater than the pivot come after the pivot
  - **Recursively apply** the above steps to the sub-lists

- **Algorithm analysis**
  - Time complexity = O(N logN) on average & O(N^2) in the worst case
    - In each step, pivot selection = O(1) & partitioning = O(N)
    - Recursion depth = O(log N) on average & O(N) in the worst case
    - The worst case is when partitioned sub-lists are extremely skewed

  - Space complexity = the number of additionally required variables = O(1)

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them



7 12 8 3 1 6 5 4

7 12 8 3 | 1 6 5 4

7 12 | 8 3 | 1 6 | 5 4

7 | 12 | 8 | 3 | 1 | 6 | 5 | 4

7 | 12 | 8 | 3 | 1 | 6 | 5 | 4

7 12 | 3 8 | 1 6 | 4 5

3 7 8 12 | 1 4 5 6

1 3 4 5 6 7 8 12

Partitioning

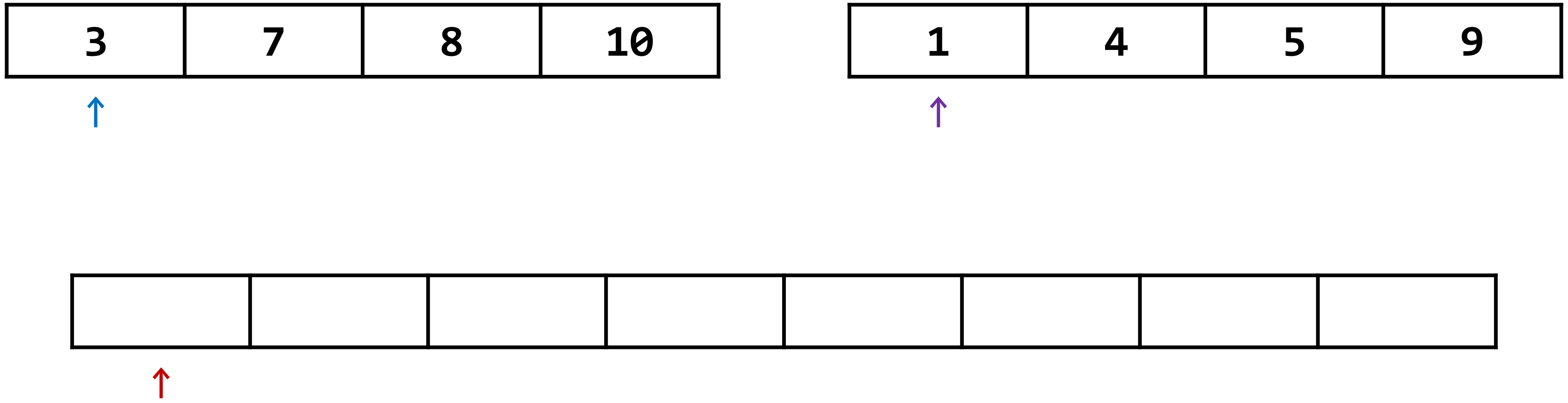
Merging

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



3	7	8	10

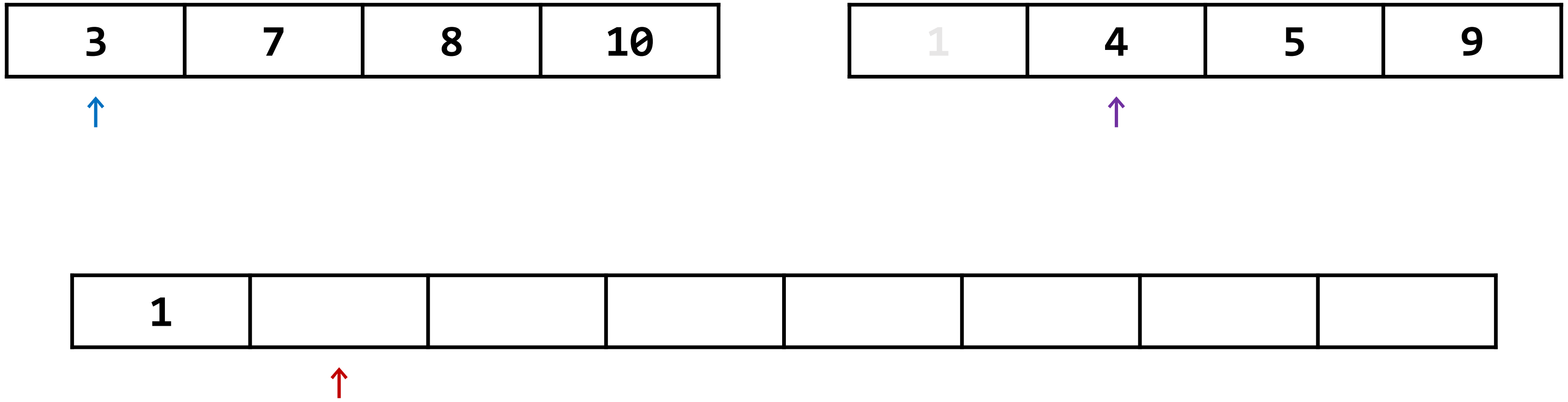
1	4	5	9

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



3	7	8	10

1	4	5	9

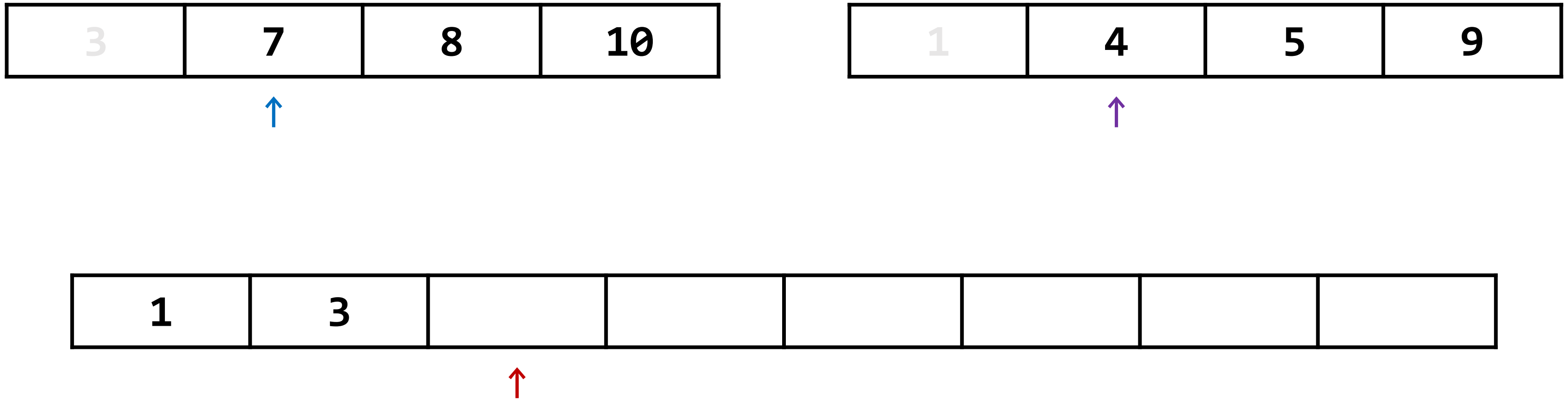
1							

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



3	7	8	10

1	4	5	9

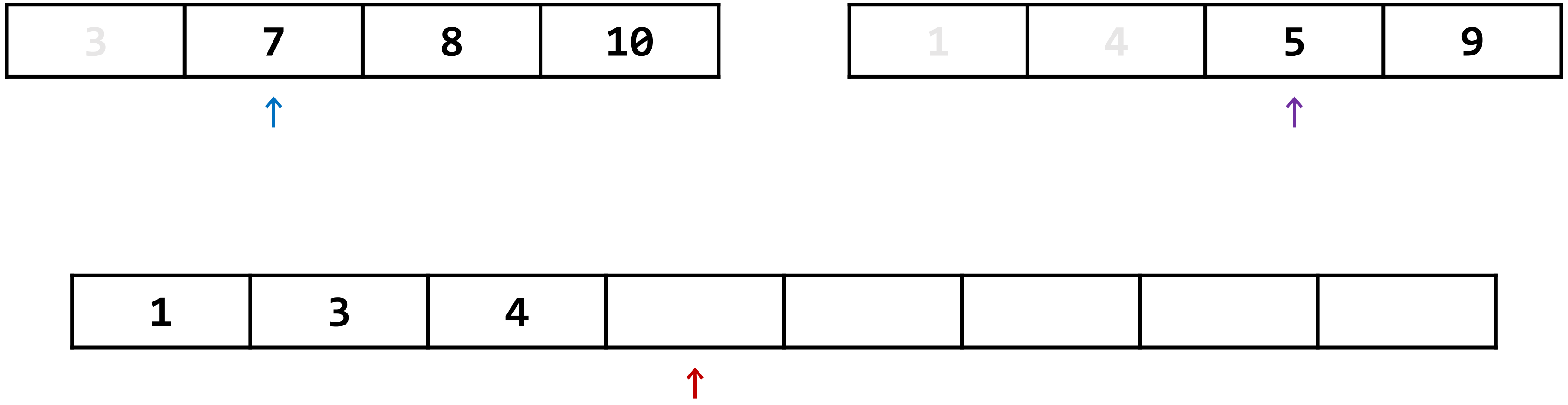
1	3						

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



3	7	8	10

1	4	5	9

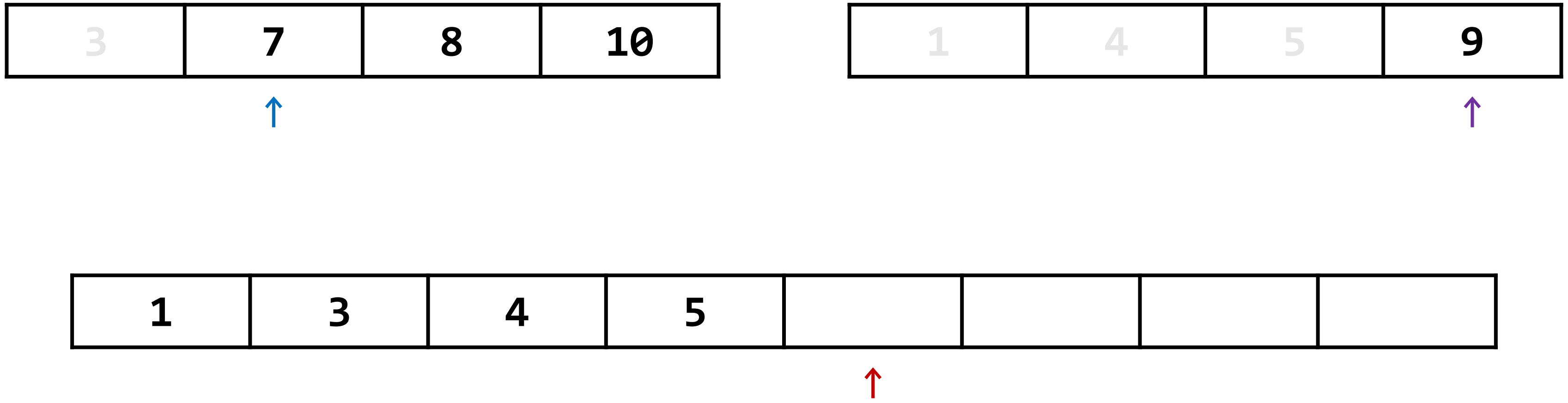
1	3	4					

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



3	7	8	10

1	4	5	9

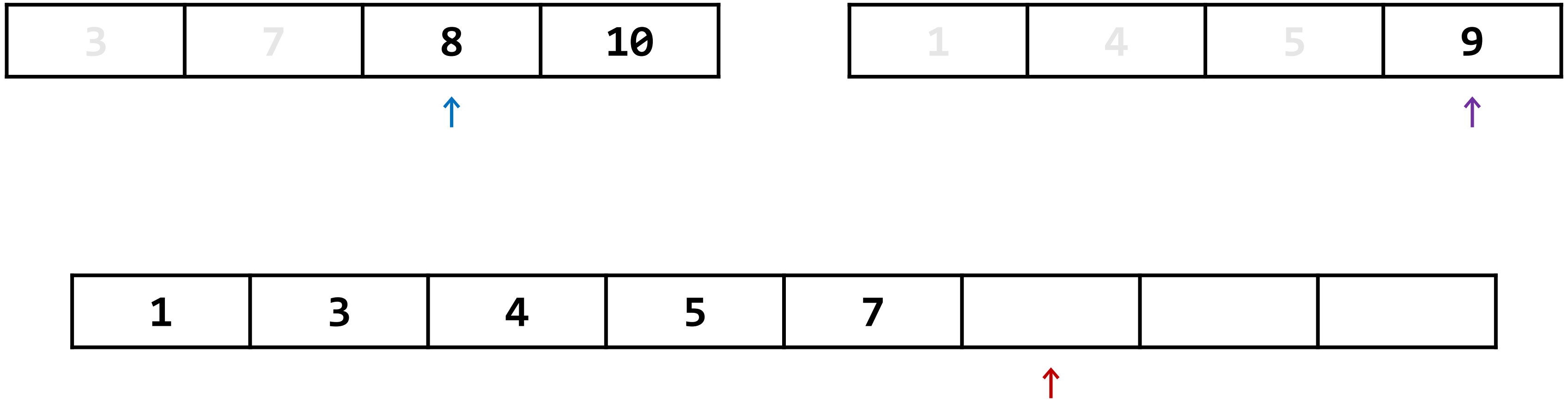
1	3	4	5				

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



3	7	8	10

1	4	5	9

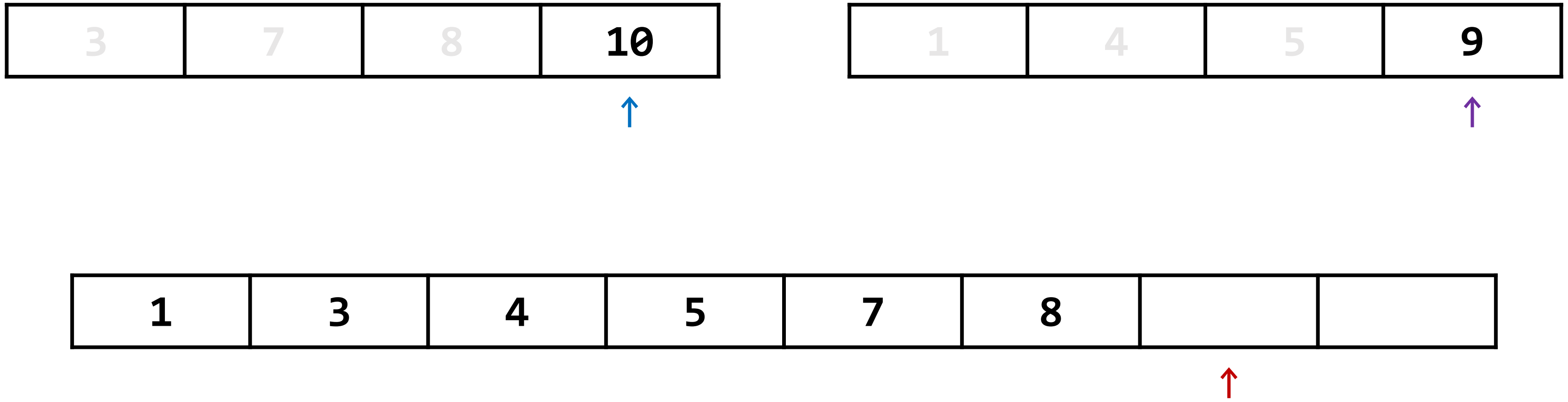
1	3	4	5	7			

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



3	7	8	**10**

1	4	5	**9**

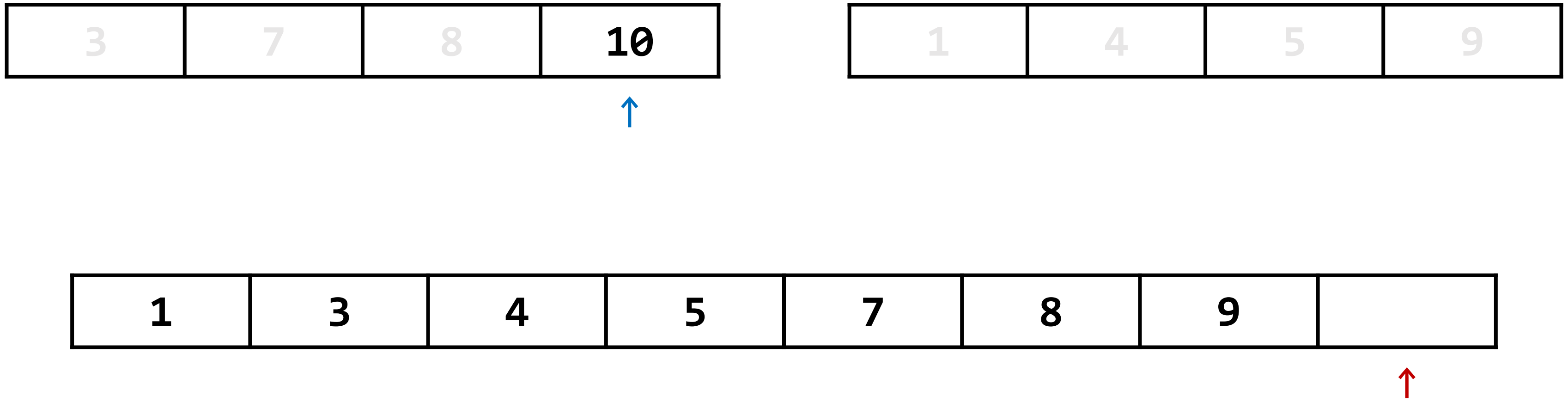
1	3	4	5	7	8		

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one



Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

(Q) How to combine two sorted sub-lists?

- The first element should be either the first item of one sub-list or that of another one

3	7	8	10

1	4	5	9

Merging a sorted list of N items and a sorted list of M items → **O(N+M)**

1	3	4	5	7	8	9	10

Merge Sort

- **Key Idea:** Split items half and half first, and then merge them
  1. **Divide**: Split the items into two half sub-lists
  2. **Conquer**: Recursively sort the two sub-lists
  3. **Combine**: Merge the two sorted sub-lists into one

- **Algorithm analysis**
  - Time complexity = O(N logN) in both the best and worst cases
    - Recursion depth = O(log N)
    - Total complexity = N + 2*(N/2) + 4*(N/4) + … = N * logN

  - Space complexity = the number of additionally required variables = O(N)
    - It requires an auxiliary array to store the merged list

Comparison of Sorting Algorithms

Algorithm	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Any Questions?