Chapter 3: Sorting Algorithms

Algorithms

Stability and in-place sorting | Bubble | Insertion | Selection | Shell | Merge | Quick | Heapsort | Counting and radix | Timsort | Comparison

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/algorithms/slides/Chapter_03_Sorting_Algorithms.html

Learning Outcomes

Concepts

Evaluate a sort by time, extra memory, **stability**, whether it is **in place**, and whether it is comparison-based.

Simple algorithms

Trace bubble, insertion, selection and Shell sort on an array step by step and implement them in Java.

Divide and conquer, heaps

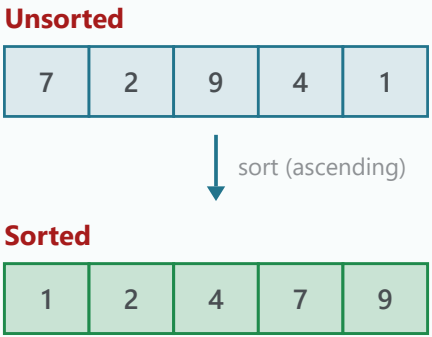
Explain how merge sort, quicksort and heapsort work; justify their O(n log n) and worst-case behavior.

Linear time and choosing

Explain the comparison lower bound and counting and radix sort; choose the right algorithm for a problem.

Sorting Algorithms

- From shopping lists to social media posts, the digital world is full of **data**.
- To use data efficiently we need to keep it **organized**.
- **Sorting algorithms** arrange the items of a list according to some **criterion**:
  - numeric value,
  - alphabetical order,
  - date …
- The order can be **ascending** (smallest first) or **descending**.
- Formally: the input is $\langle a_1, a_2, \ldots, a_n \rangle$; the output is a **permutation** of the same numbers with $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.



Sorted data makes many tasks easy: **binary search** in O(log n), finding duplicates, merging, the median, the k smallest or largest items.

Kinds of Sorting Algorithms

Different sorting algorithms work on different principles:

Algorithm	Key idea
Bubble sort	Compares adjacent items and swaps them if they are out of order.
Insertion sort	Inserts the next item into its place in the sorted part.
Selection sort	Finds the smallest item and moves it to the front, then sorts the rest.
Shell sort	Insertion sort that starts with distant items and shrinks the gap.
Merge sort	Splits the array in half, sorts the halves, then merges them.
Quicksort	Partitions around a pivot and sorts the two parts recursively.
Heapsort	Builds a max-heap and repeatedly moves the largest item to the end.
Counting / Radix sort	No comparisons; counts the keys or distributes them digit by digit.
Timsort	A hybrid of insertion sort and merge sort; the object sort of Java and Python.

Evaluating a Sorting Algorithm

Time

Comparisons and swaps (moves) in the best, average and worst case: $\Theta(n^2)$, $\Theta(n \log n)$, $\Theta(n + k)$ ...

Extra memory

Space needed beyond the input. An algorithm that uses $O(1)$ extra memory sorts **in place**.

Stability

An algorithm is stable if items with equal keys **keep their relative order**.

Comparison-based?

Algorithms that only ask "$a \leq b$?" need $\Omega(n \log n)$ comparisons in the worst case. Counting and radix sort escape this bound.

Adaptivity

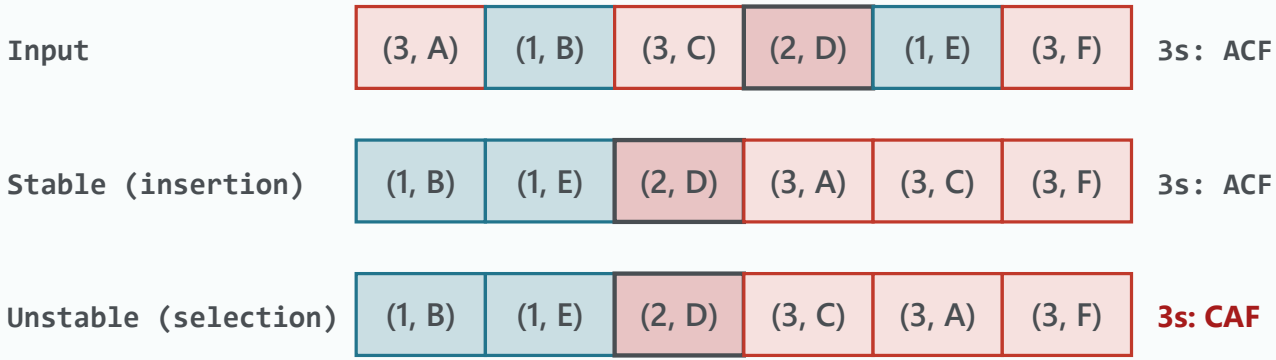
An algorithm that runs faster on nearly sorted input is **adaptive**: insertion sort, bubble sort with a flag, Timsort.

Online

An algorithm that can sort data as it arrives is **online**: insertion sort.

These bounds count one comparison as $O(1)$; for strings of length L each comparison can cost $O(L)$ (see Chapters 10–11).

Stability: An Example

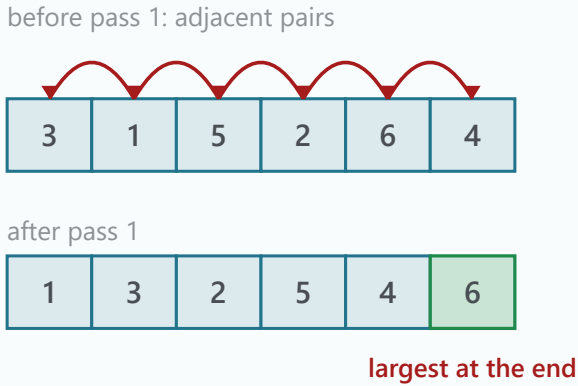


Records are sorted by the **number** (the key) only; the letter identifies the record. A stable algorithm keeps the records with key 3 in input order, **A, C, F**; the long-distance swaps of selection sort turn this into **C, A, F**. (This input is the example from the simulator lab.)

Stability matters in **multi-key sorting**: sort students by name, then (with a stable algorithm) by department, and each department stays alphabetical. Radix sort needs a stable inner sort for the same reason.

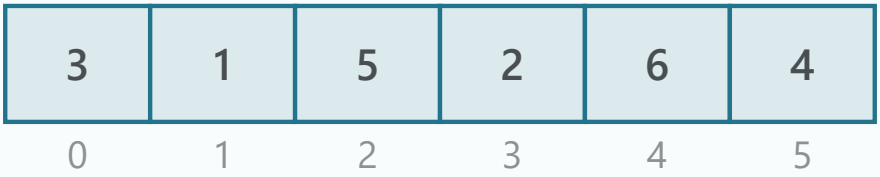
Bubble Sort

- On each pass the **larger items** move toward the end of the array, rising like bubbles in water.
- **Adjacent** items are compared in pairs.
- If the left one is larger, they are exchanged (**swap**).
- After pass i, the i largest items are **in their final places**; the next pass is one item shorter.
- If a pass makes no swaps, the array is sorted → **early exit**.



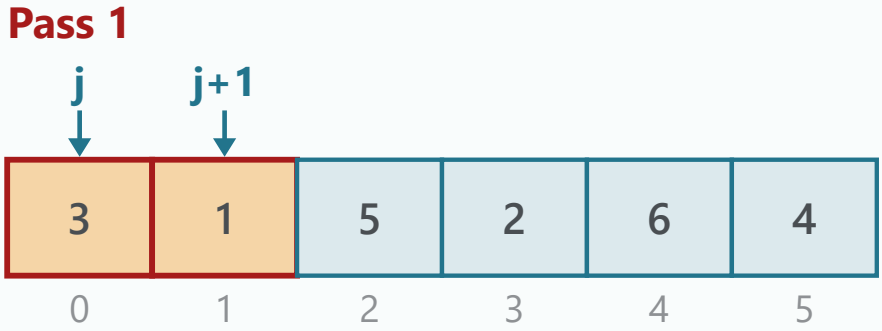
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

Start



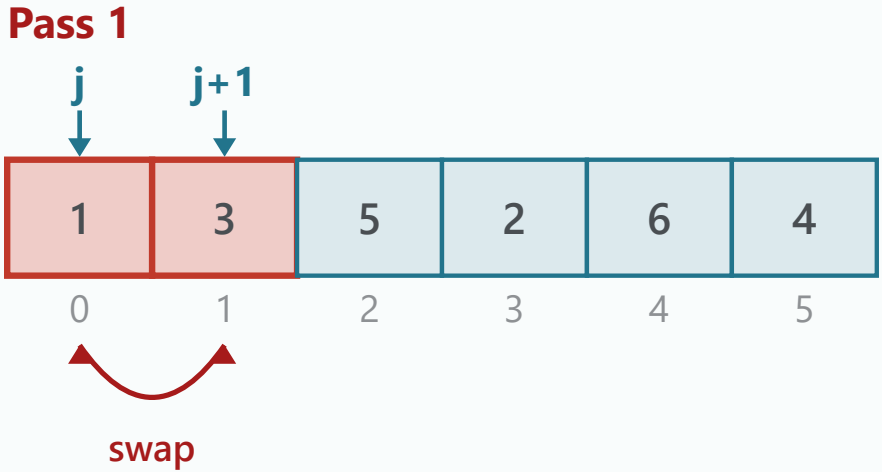
Array: [3, 1, 5, 2, 6, 4]. Each pass compares adjacent pairs from left to right; the larger one moves right.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]



Pass 1, j = 0: 3 > 1? **Yes** → swap.

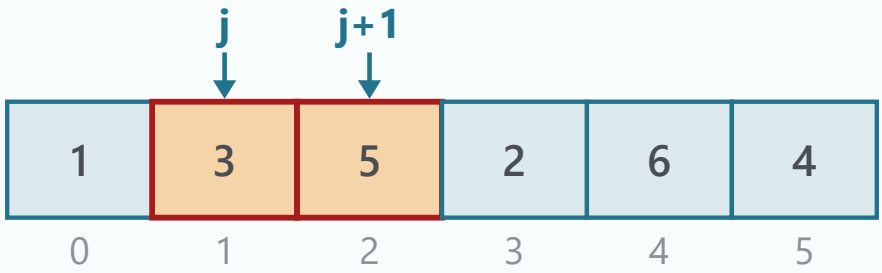
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]



Swap: the array is now [1, 3, 5, 2, 6, 4].

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

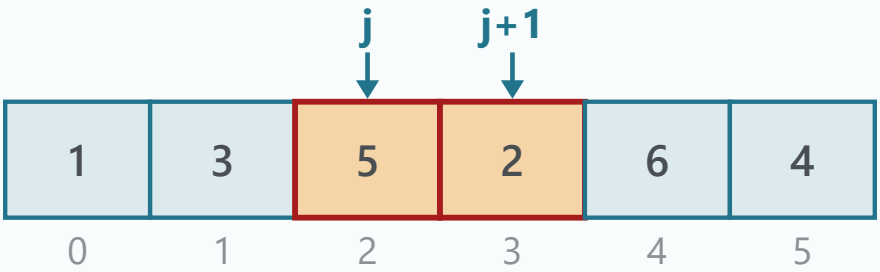
Pass 1



Pass 1, j = 1: 3 > 5? No → stays.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

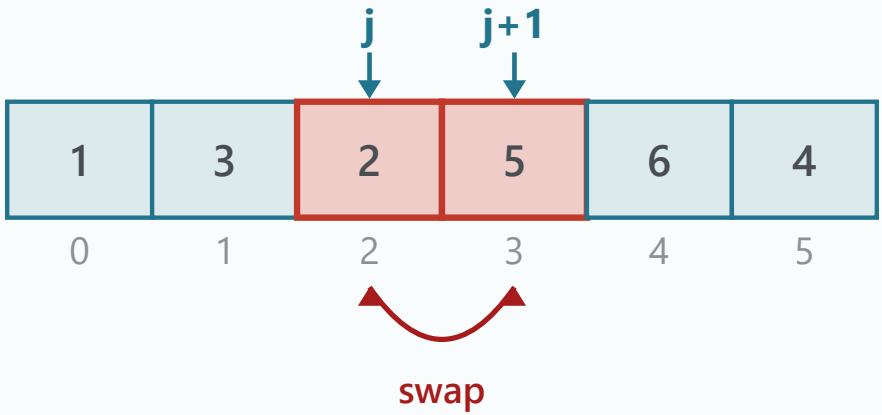
Pass 1



Pass 1, j = 2: 5 > 2? **Yes** → swap.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

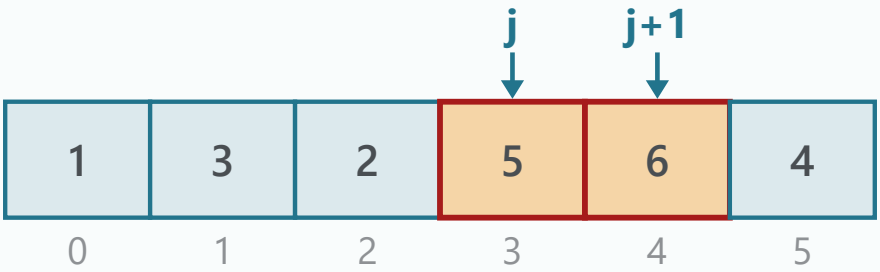
Pass 1



Swap: the array is now [1, 3, 2, 5, 6, 4].

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

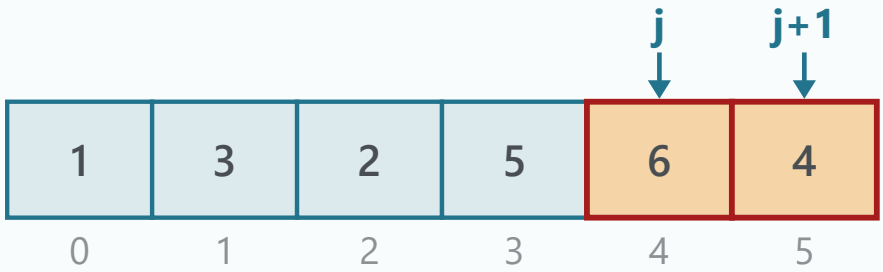
Pass 1



Pass 1, j = 3: 5 > 6? No → stays.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

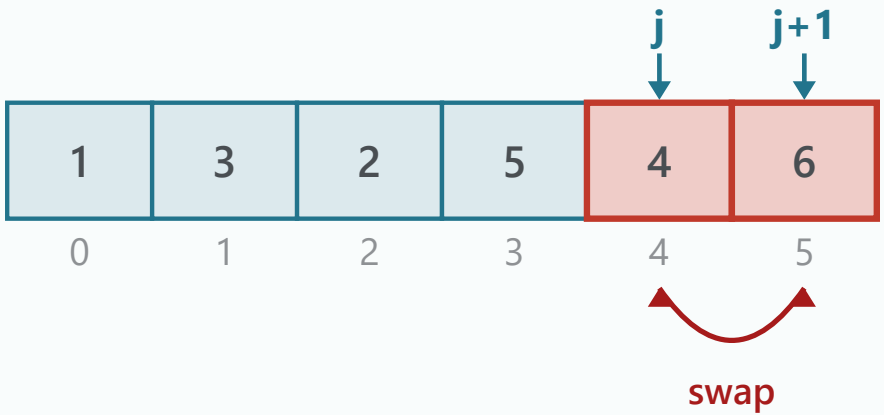
Pass 1



Pass 1, j = 4: 6 > 4? **Yes** → swap.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

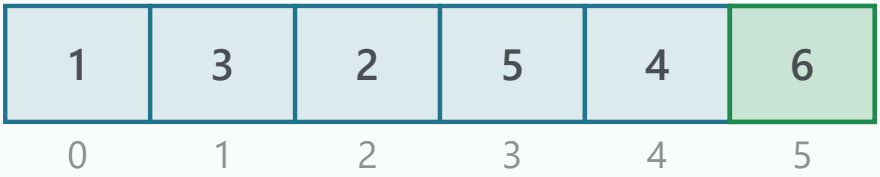
Pass 1



Swap: the array is now [1, 3, 2, 5, 4, 6].

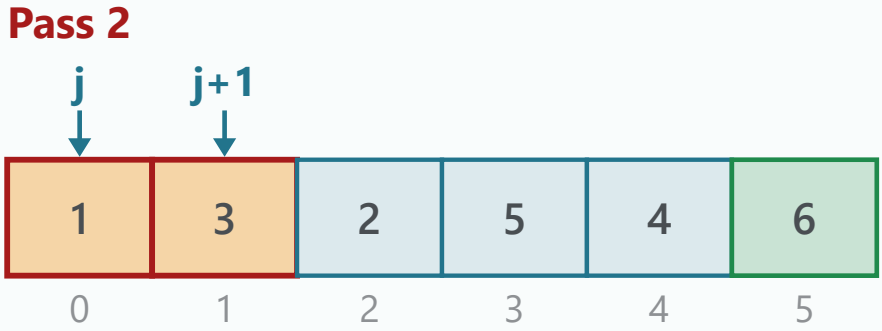
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

End of pass 1



Pass 1 done: the largest remaining item, **6**, is in its final place (index 5).

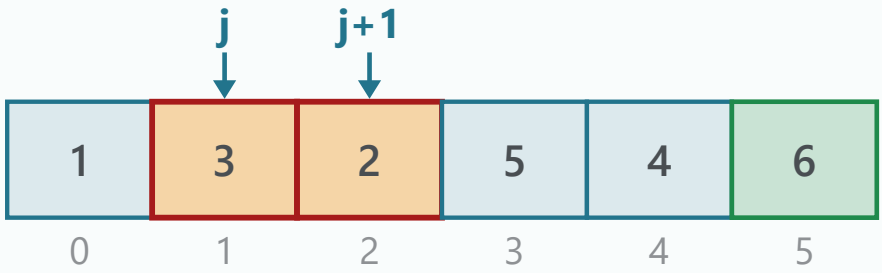
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]



Pass 2, j = 0: 1 > 3? No → stays.

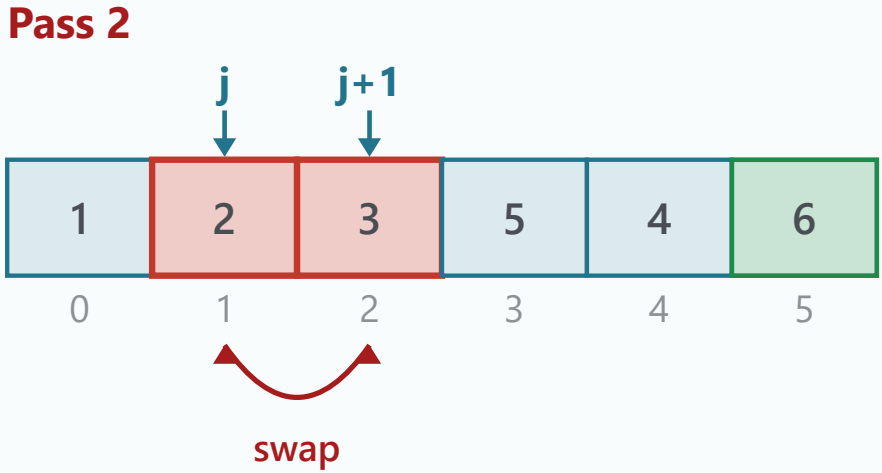
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

Pass 2



Pass 2, j = 1: 3 > 2? **Yes** → swap.

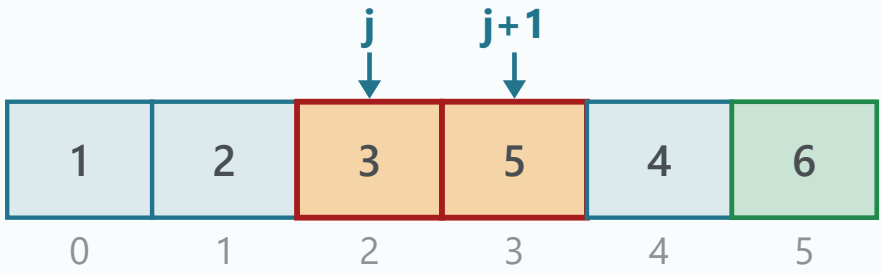
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]



Swap: the array is now [1, 2, 3, 5, 4, 6].

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

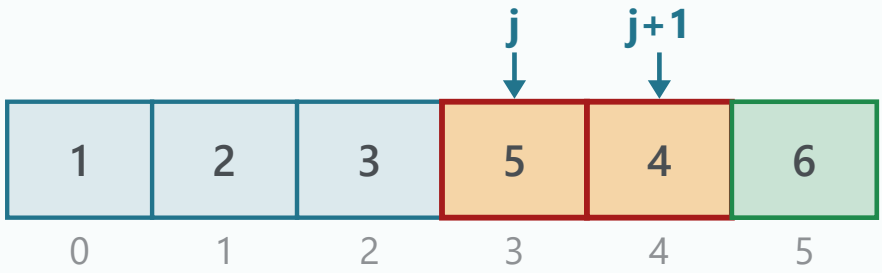
Pass 2



Pass 2, j = 2: 3 > 5? No → stays.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

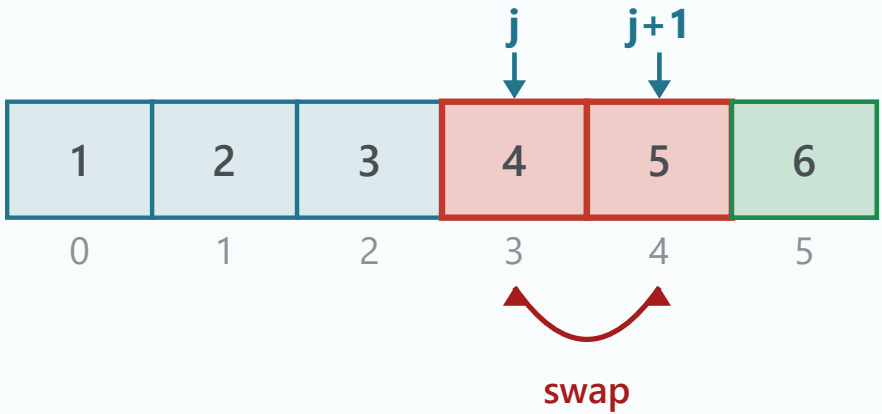
Pass 2



Pass 2, j = 3: 5 > 4? **Yes** → swap.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

Pass 2



Swap: the array is now [1, 2, 3, 4, 5, 6].

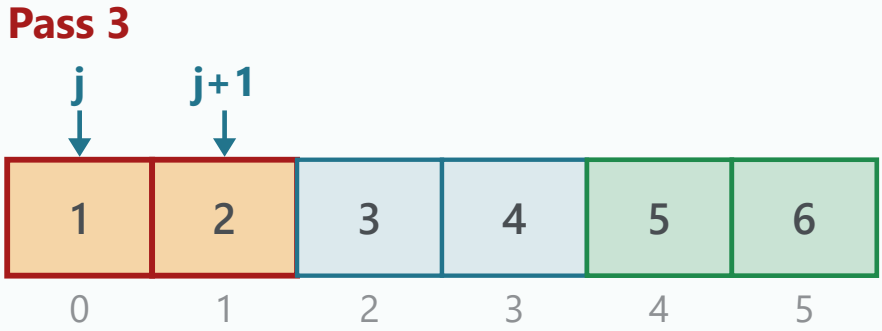
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

End of pass 2



Pass 2 done: the largest remaining item, **5**, is in its final place (index 4).

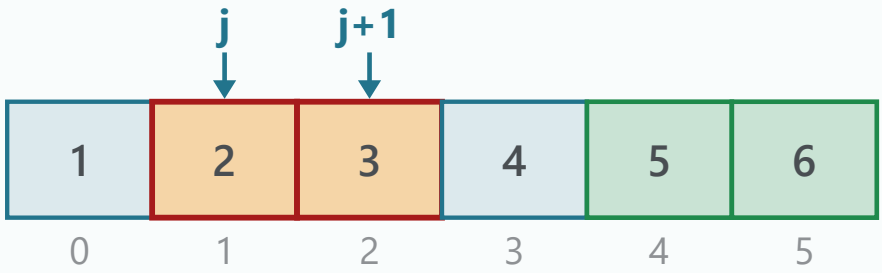
Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]



Pass 3, j = 0: 1 > 2? No → stays.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

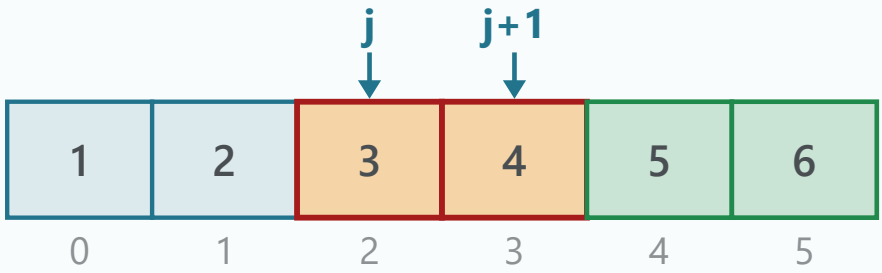
Pass 3



Pass 3, j = 1: 2 > 3? No → stays.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

Pass 3



Pass 3, j = 2: 3 > 4? No → stays.

Bubble Sort Step by Step: [3, 1, 5, 2, 6, 4]

End of pass 3



Pass 3 made no swaps → the array is **sorted** and the algorithm stops early. Result: [1, 2, 3, 4, 5, 6].

Bubble Sort: Java Code

```java
boolean swapped;
for(int i = 0; i < n - 1; i++) {
  swapped = false;
  for(int j = 0; j < n - 1 - i; j++) {
    if(arr[j] > arr[j+1]) {
      int temp = arr[j];
      arr[j] = arr[j+1];
      arr[j+1] = temp;
      swapped = true;
    }
  }
  if(swapped == false) break;
}
```

- **Outer loop** (i): at most n − 1 passes.
- **Inner loop** (j): adjacent pairs in the unsorted part; the last i items are already in place, hence `j < n − 1 − i`.
- The swap takes three assignments using `temp`.
- The `swapped` flag: if a whole pass makes no swap, the array is sorted → `break`.

Because the test is `>` (not `≥`), equal items are never swapped: bubble sort is **stable**.

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
4    for(int j = 0; j < n - 1 - i; j++) {
5      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```

arr[]	5	1	9	2	10
	0	1	2	3	4

n = 5. The flag `swapped` is declared.

Step 1 / 19

VARIABLES

n = 5 | i = – | j = – | swapped = –

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
▶ for(int i = 0; i < n - 1; i++) {
▶   swapped = false;
4   for(int j = 0; j < n - 1 - i; j++) {
5     if(arr[j] > arr[j+1]) {
6       int temp = arr[j];
7       arr[j] = arr[j+1];
8       arr[j+1] = temp;
9       swapped = true;
10    }
11  }
12  if(swapped == false) break;
13 }
```



Outer loop i = 0: the flag is set to `false`.

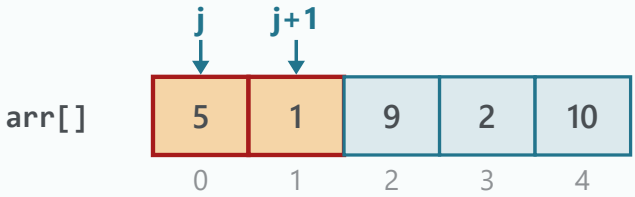
Step 2 / 19

VARIABLES

n = 5 | i = 0 | j = – | swapped = false

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 0: arr[0] = 5 > arr[1] = 1? **Yes**.

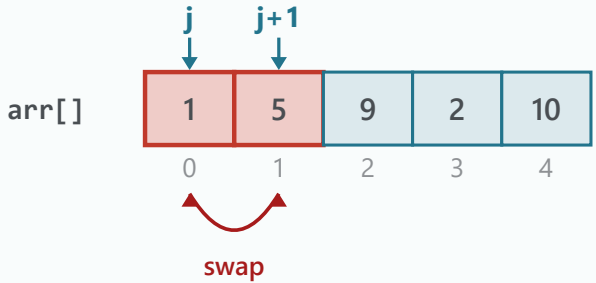
Step 3 / 19

VARIABLES

`n = 5` `i = 0` `j = 0` `swapped = false`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
4    for(int j = 0; j < n - 1 - i; j++) {
5      if(arr[j] > arr[j+1]) {
▶        int temp = arr[j];
▶        arr[j] = arr[j+1];
▶        arr[j+1] = temp;
▶        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



Swapped via `temp` = 5; the flag becomes `true`.

Step 4 / 19

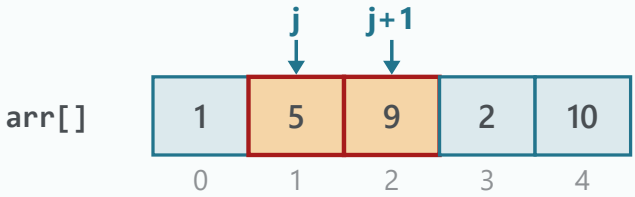
VARIABLES

`n = 5` `i = 0` `j = 0` `temp = 5`

`swapped = true`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 1: arr[1] = 5 > arr[2] = 9? No.

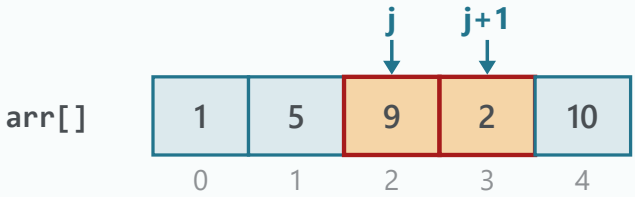
Step 5 / 19

VARIABLES

n = 5 | i = 0 | j = 1 | swapped = true

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 2: arr[2] = 9 > arr[3] = 2? **Yes**.

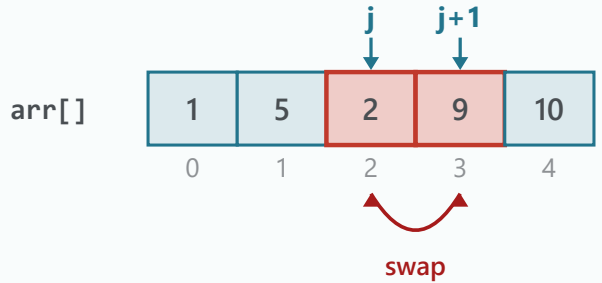
Step 6 / 19

VARIABLES

`n = 5` `i = 0` `j = 2` `swapped = true`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
4    for(int j = 0; j < n - 1 - i; j++) {
5      if(arr[j] > arr[j+1]) {
▶        int temp = arr[j];
▶        arr[j] = arr[j+1];
▶        arr[j+1] = temp;
▶        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



Swapped via `temp` = 9; the flag becomes `true`.

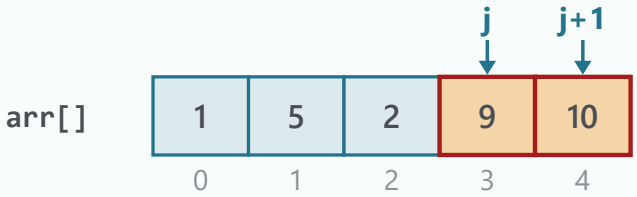
Step 7 / 19

VARIABLES

`n = 5` `i = 0` `j = 2` `temp = 9` `swapped = true`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 3: arr[3] = 9 > arr[4] = 10? No.

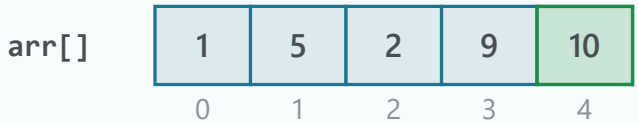
Step 8 / 19

VARIABLES

n = 5 | i = 0 | j = 3 | swapped = true

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
4    for(int j = 0; j < n - 1 - i; j++) {
5      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
▶    if(swapped == false) break;
13 }
```



End of pass: 10 is in place. Flag true → continue.

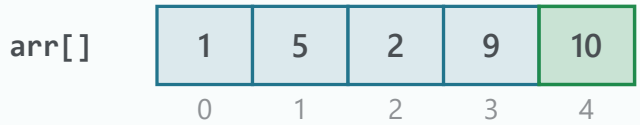
Step 9 / 19

VARIABLES

`n = 5` `i = 0` `j = –` `swapped = true`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
▶ for(int i = 0; i < n - 1; i++) {
▶   swapped = false;
4   for(int j = 0; j < n - 1 - i; j++) {
5     if(arr[j] > arr[j+1]) {
6       int temp = arr[j];
7       arr[j] = arr[j+1];
8       arr[j+1] = temp;
9       swapped = true;
10    }
11  }
12  if(swapped == false) break;
13 }
```



Outer loop i = 1: the flag is set to `false`.

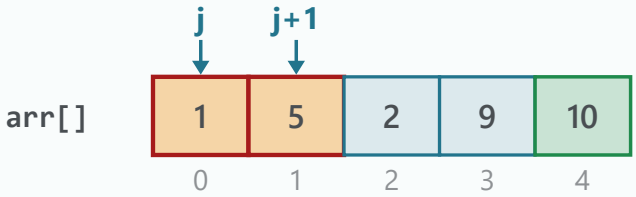
Step 10 / 19

VARIABLES

n = **5** | i = **1** | j = **–** | swapped = **false**

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 0: arr[0] = 1 > arr[1] = 5? No.

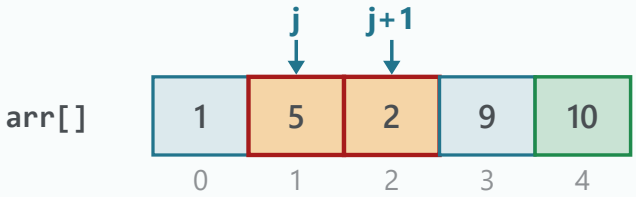
Step 11 / 19

VARIABLES

n = 5 | i = 1 | j = 0 | swapped = false

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 1: arr[1] = 5 > arr[2] = 2? **Yes**.

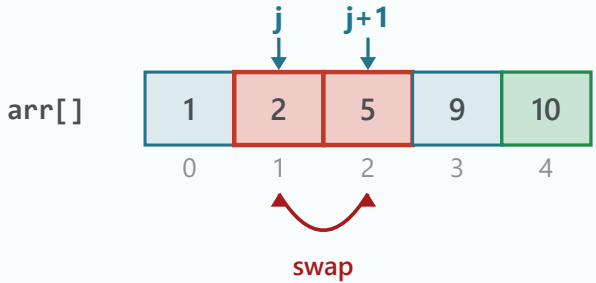
Step 12 / 19

VARIABLES

`n = 5` `i = 1` `j = 1` `swapped = false`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
4    for(int j = 0; j < n - 1 - i; j++) {
5      if(arr[j] > arr[j+1]) {
▶        int temp = arr[j];
▶        arr[j] = arr[j+1];
▶        arr[j+1] = temp;
▶        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



Swapped via `temp` = 5; the flag becomes `true`.

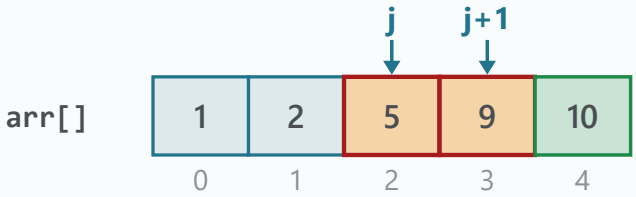
Step 13 / 19

VARIABLES

`n = 5` `i = 1` `j = 1` `temp = 5` `swapped = true`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 2: arr[2] = 5 > arr[3] = 9? No.

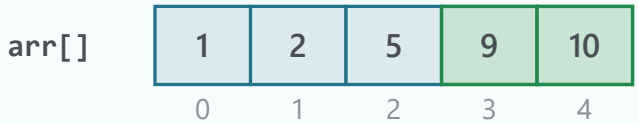
Step 14 / 19

VARIABLES

n = **5** | i = **1** | j = **2** | swapped = **true**

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
4    for(int j = 0; j < n - 1 - i; j++) {
5      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
▶    if(swapped == false) break;
13 }
```



End of pass: 9 is in place. Flag true → continue.

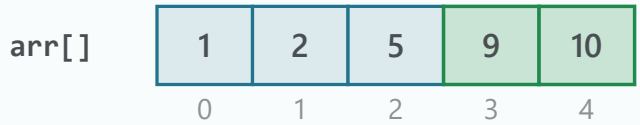
Step 15 / 19

VARIABLES

`n = 5` `i = 1` `j = –` `swapped = true`

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
▶ for(int i = 0; i < n - 1; i++) {
▶   swapped = false;
4   for(int j = 0; j < n - 1 - i; j++) {
5     if(arr[j] > arr[j+1]) {
6       int temp = arr[j];
7       arr[j] = arr[j+1];
8       arr[j+1] = temp;
9       swapped = true;
10    }
11  }
12  if(swapped == false) break;
13 }
```



Outer loop i = 2: the flag is set to `false`.

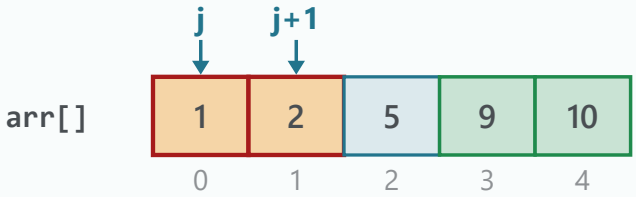
Step 16 / 19

VARIABLES

n = 5 | i = 2 | j = – | swapped = false

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 0: arr[0] = 1 > arr[1] = 2? No.

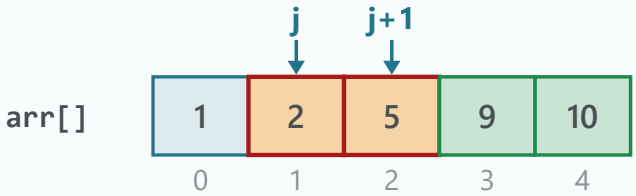
Step 17 / 19

VARIABLES

n = 5 | i = 2 | j = 0 | swapped = false

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
▶    for(int j = 0; j < n - 1 - i; j++) {
▶      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
12   if(swapped == false) break;
13 }
```



j = 1: arr[1] = 2 > arr[2] = 5? No.

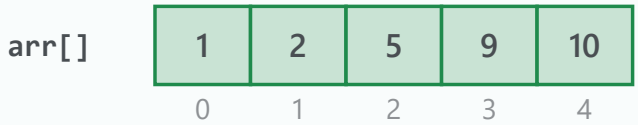
Step 18 / 19

VARIABLES

n = 5 | i = 2 | j = 1 | swapped = false

Bubble Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1  boolean swapped;
2  for(int i = 0; i < n - 1; i++) {
3    swapped = false;
4    for(int j = 0; j < n - 1 - i; j++) {
5      if(arr[j] > arr[j+1]) {
6        int temp = arr[j];
7        arr[j] = arr[j+1];
8        arr[j+1] = temp;
9        swapped = true;
10     }
11   }
▶    if(swapped == false) break;
13 }
```



No swap in this pass: `break`. The array is sorted: [1, 2, 5, 9, 10].

Step 19 / 19

VARIABLES

`n = 5` `i = 2` `j = –` `swapped = false`

Bubble Sort: Analysis

Case	Input	Comparisons	Swaps	Time
Best	sorted	$n - 1$ (one pass, with flag)	0	$\Theta(n)$
Average	random	$\approx n^2/2$	$\approx n^2/4$	$\Theta(n^2)$
Worst	reverse sorted	$n(n - 1)/2$	$n(n - 1)/2$	$\Theta(n^2)$

Pros

+ Very easy to understand and write.
+ In place ($O(1)$ extra memory) and stable.
+ With the flag, recognizes sorted input in $\Theta(n)$.

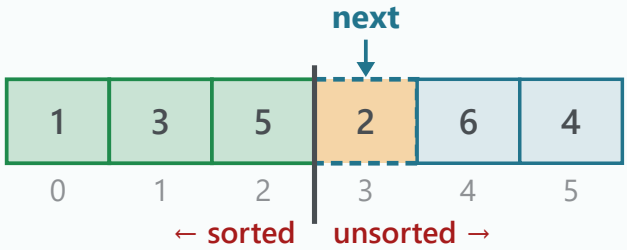
Cons

– $\Theta(n^2)$: very slow on large arrays.
– Makes many swaps (each inversion = 1 swap).
– In practice always slower than insertion sort.

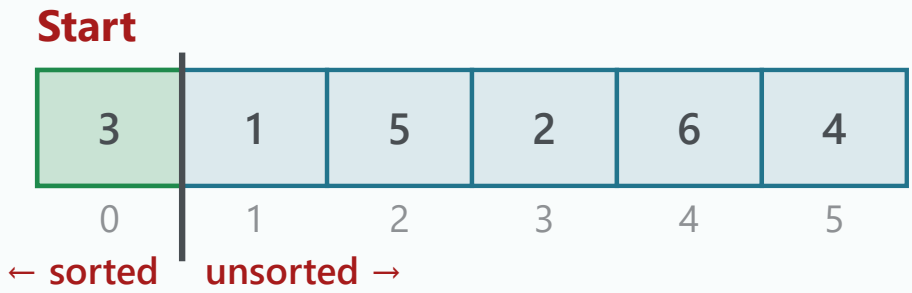
An inversion is a pair $i < j$ with $arr[i] > arr[j]$. Bubble and insertion sort fix exactly one inversion per step; a reverse-sorted array has $n(n - 1)/2$ inversions.

Insertion Sort

- Like sorting a hand of **playing cards**: each new card is slipped into its place among the sorted cards in your hand.
- The array has two parts: a **sorted part** (on the left) and an **unsorted part**.
- Take the first item of the unsorted part and insert it at its **correct position** in the sorted part.
- To make room, the larger items are **shifted** one place to the right.
- **Invariant**: at the start of each outer step, `arr[0..i-1]` is sorted.

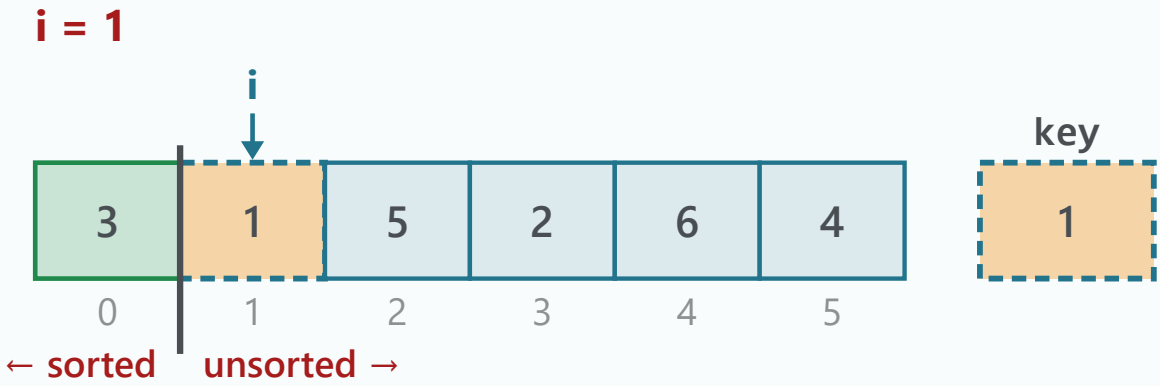


Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



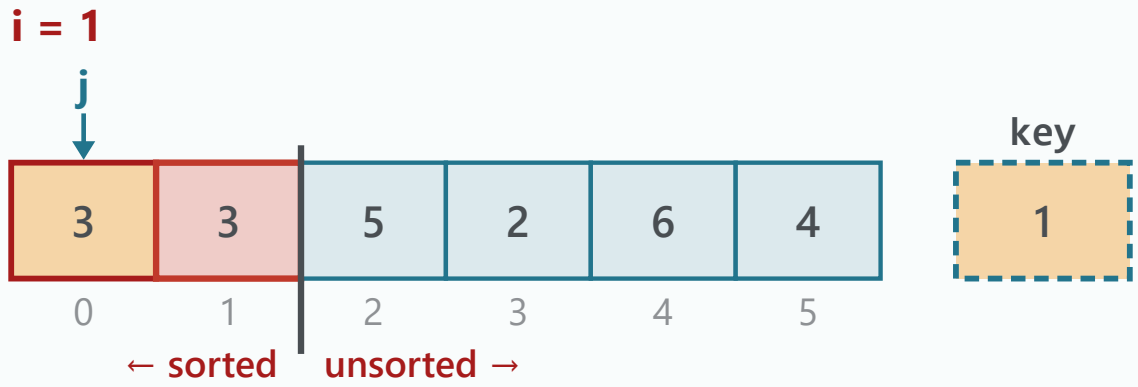
Array [3, 1, 5, 2, 6, 4]. The first item alone is the **sorted part**; the rest is unsorted.

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



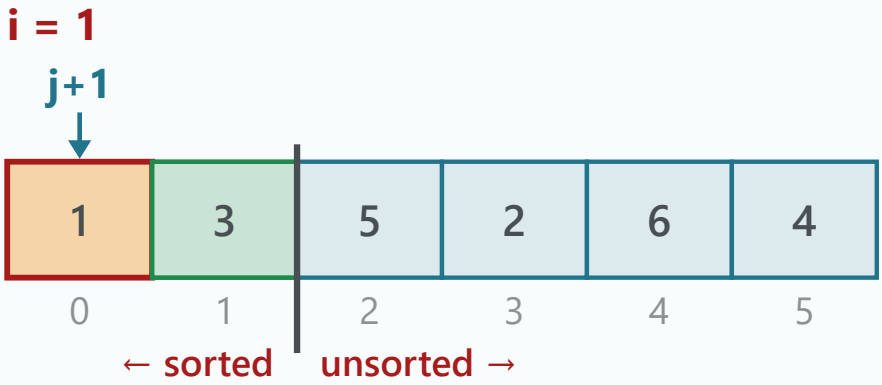
Take the first item of the unsorted part: `key` = 1.

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



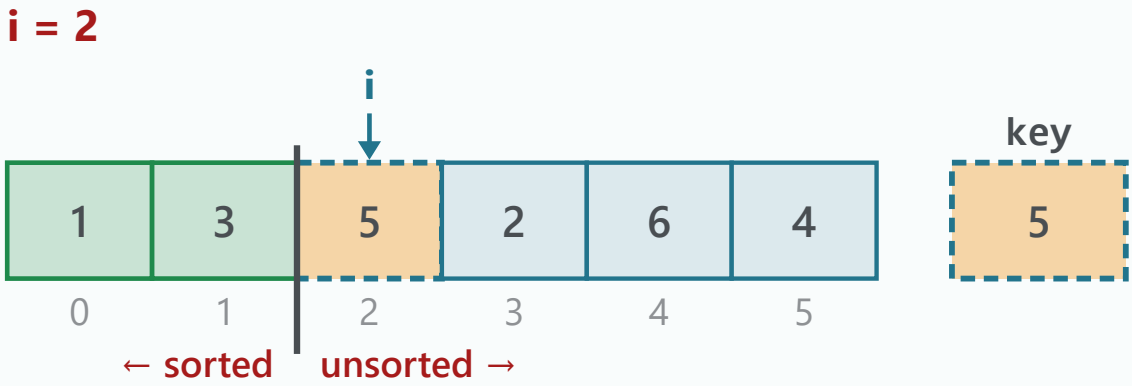
3 > 1 → 3 is **shifted** right (index 0 → 1).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



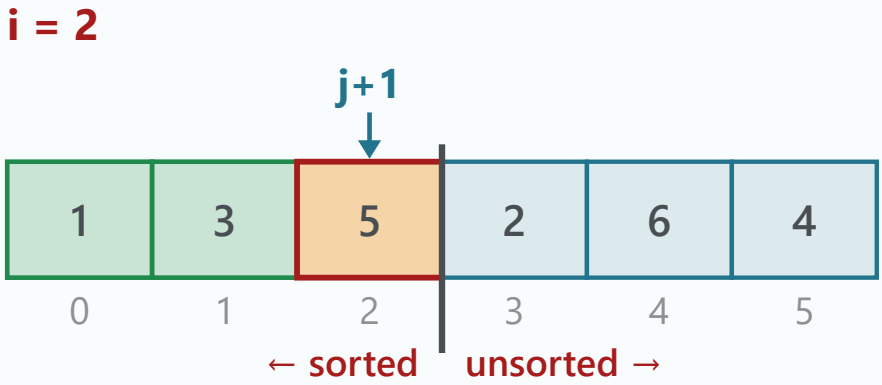
Reached the start of the array → 1 is **inserted** at index 0 (1 shift).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



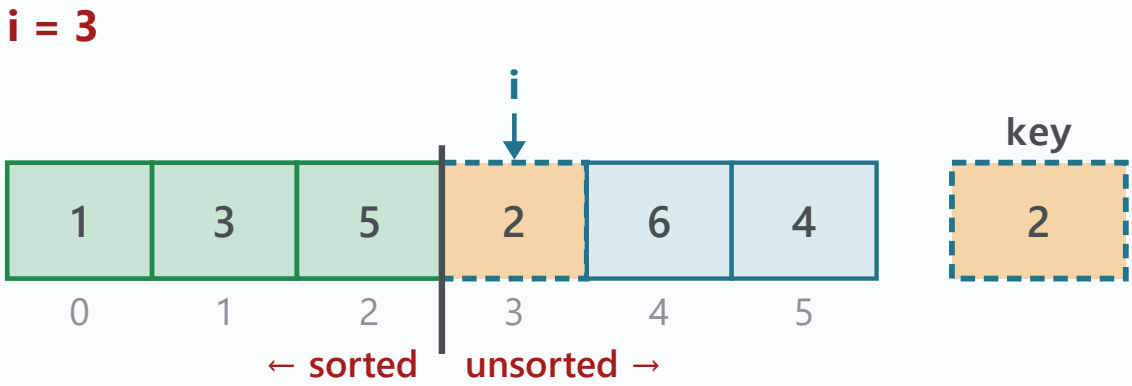
Take the first item of the unsorted part: `key` = 5.

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



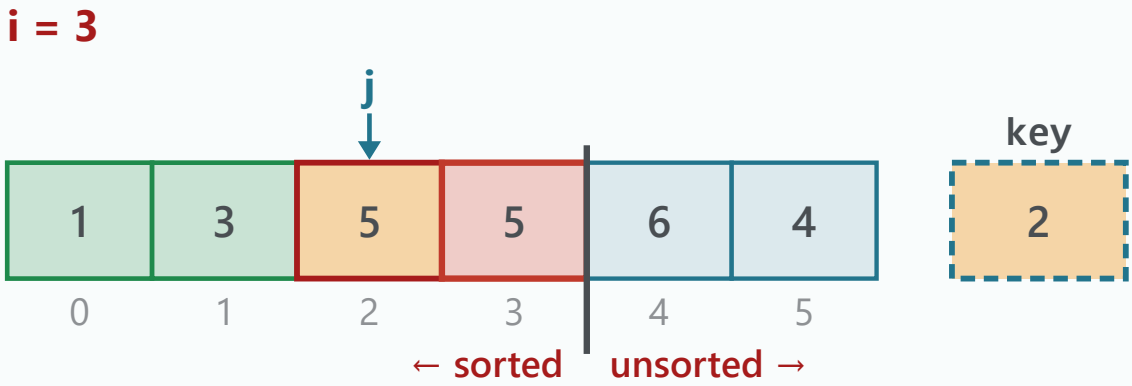
3 ≤ 5 → 5 is **inserted** at index 2 (no shift needed).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



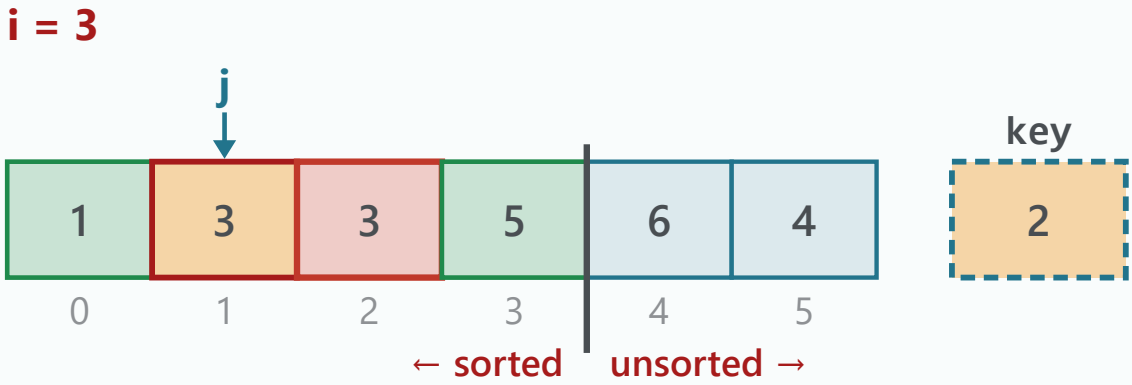
Take the first item of the unsorted part: `key` = 2.

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



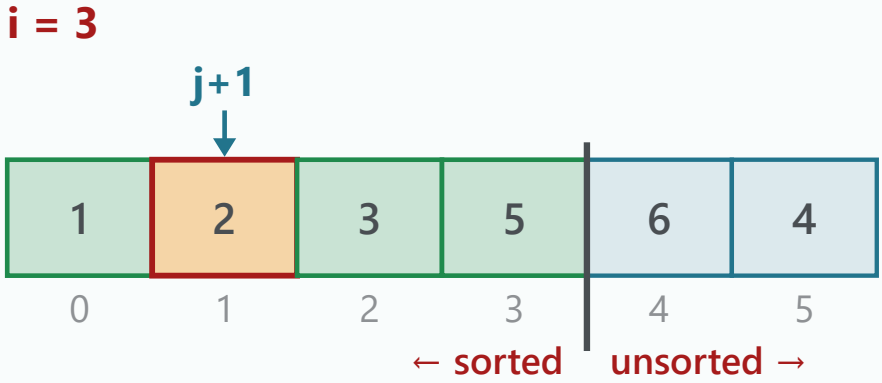
5 > 2 → 5 is **shifted** right (index 2 → 3).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



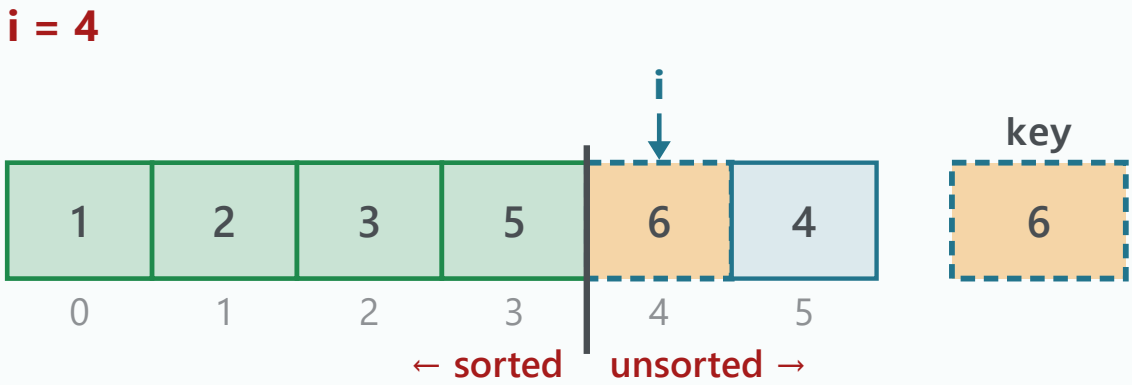
3 > 2 → 3 is **shifted** right (index 1 → 2).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



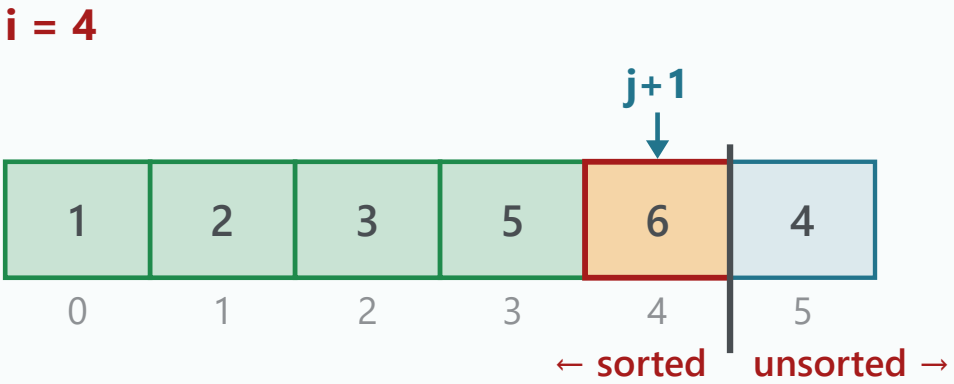
1 ≤ 2 → 2 is **inserted** at index 1 (2 shifts).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



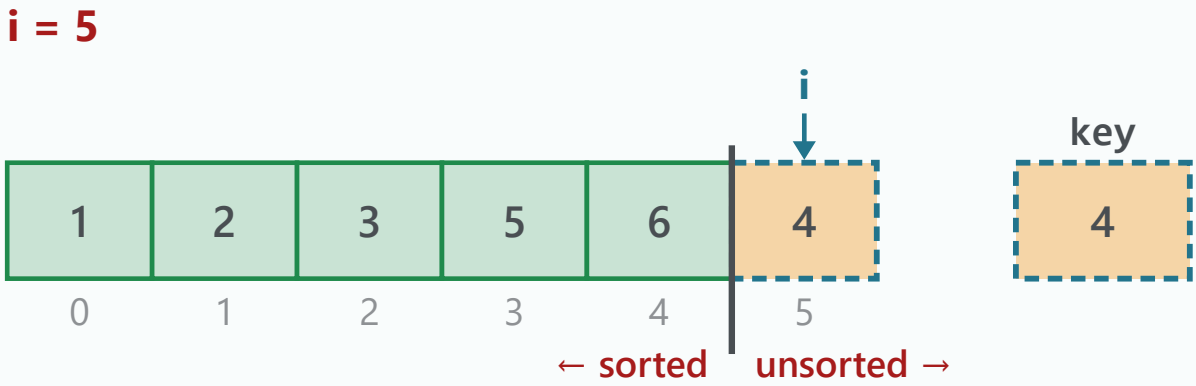
Take the first item of the unsorted part: `key` = 6.

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



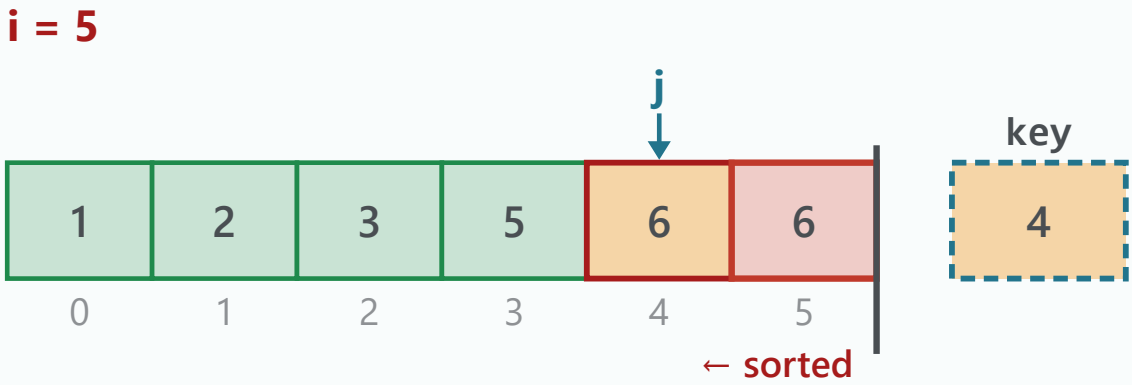
5 ≤ 6 → 6 is **inserted** at index 4 (no shift needed).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



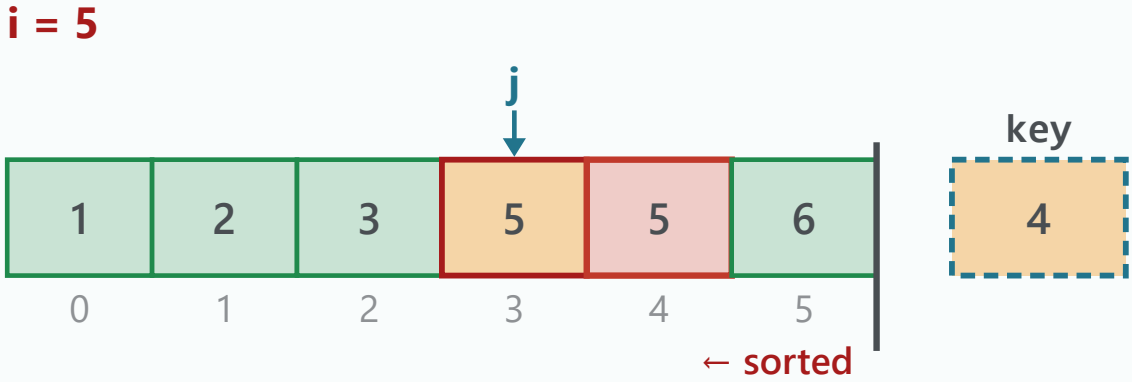
Take the first item of the unsorted part: `key` = 4.

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



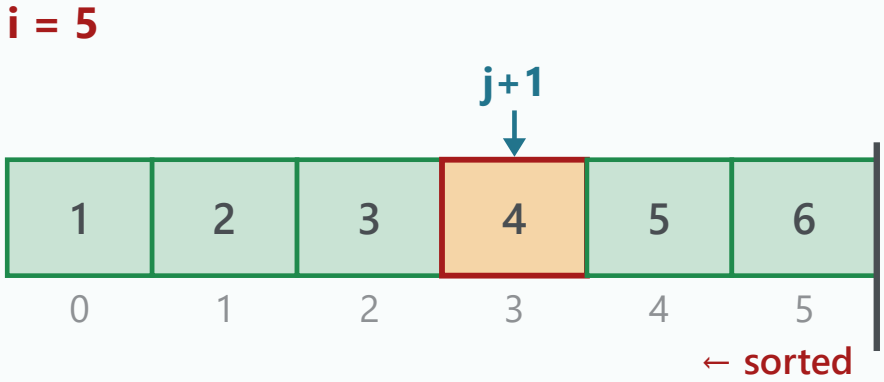
6 > 4 → 6 is **shifted** right (index 4 → 5).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



5 > 4 → 5 is **shifted** right (index 3 → 4).

Insertion Sort Step by Step: [3, 1, 5, 2, 6, 4]



3 ≤ 4 → 4 is **inserted** at index 3 (2 shifts). Sorted: [1, 2, 3, 4, 5, 6].

Insertion Sort: Java Code

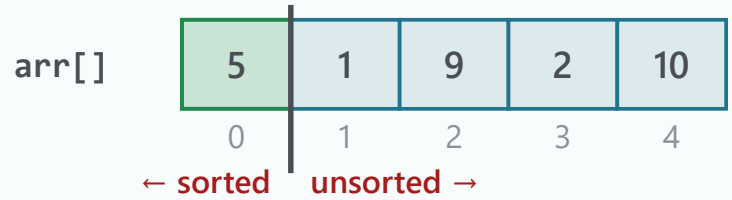
```
for(int i = 1; i < n; i++) {
  int key = arr[i];
  int j = i - 1;
  while(j >= 0 && arr[j] > key) {
    arr[j+1] = arr[j];
    j = j - 1;
  }
  arr[j+1] = key;
}
```

- Start at i = 1: the one-element `arr[0..0]` is already sorted.
- `key`: the item to insert. It is saved first because the shifts overwrite its cell.
- `while`: as long as the item on the left is larger, shift it one place right.
- When the loop ends, the free slot is `j + 1`.

It **shifts** rather than swaps: one assignment per step instead of 3. That is why insertion sort is noticeably faster than bubble sort.

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



n = 5. Sorted part: just arr[0].

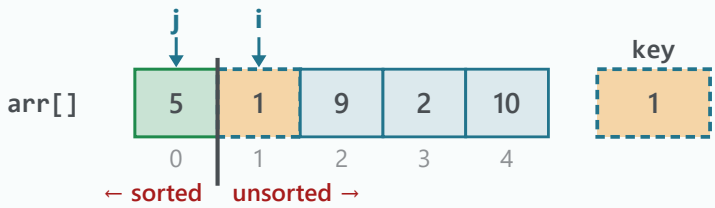
Step 1 / 20

VARIABLES

n = 5 | i = – | j = – | key = –

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
for(int i = 1; i < n; i++) {
  int key = arr[i];
  int j = i - 1;
  while(j >= 0 && arr[j] > key) {
    arr[j+1] = arr[j];
    j = j - 1;
  }
  arr[j+1] = key;
}
```



i = 1: `key` = arr[1] = 1, j = 0.

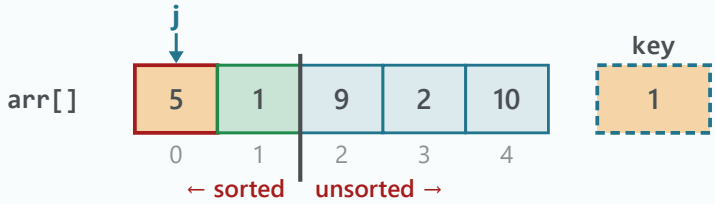
Step 2 / 20

VARIABLES

`n = 5` `i = 1` `j = 0` `key = 1`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
▶   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



j = 0 ≥ 0 and arr[0] = 5 > 1 → enter the loop.

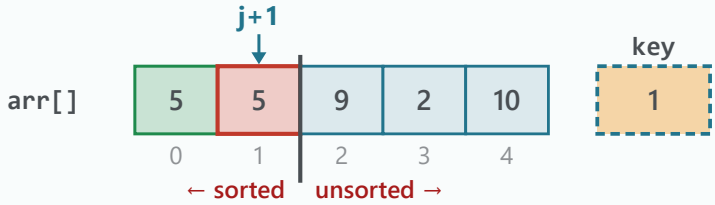
Step 3 / 20

VARIABLES

n = 5 | i = 1 | j = 0 | key = 1

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
▶     arr[j+1] = arr[j];
▶     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



arr[1] = arr[0] = 5 (shift right), j = −1.

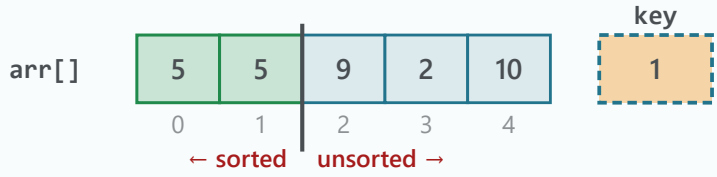
Step 4 / 20

VARIABLES

n = 5 | i = 1 | j = −1 | key = 1

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
▶   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



j = −1: the loop ends.

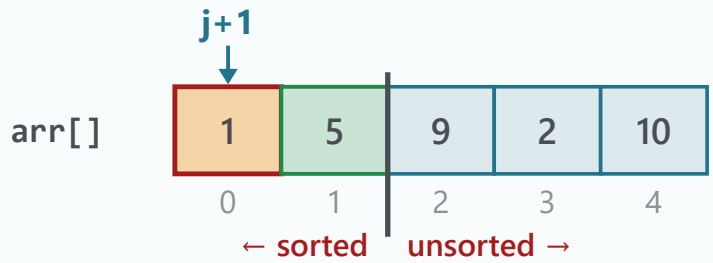
Step 5 / 20

VARIABLES

n = 5 | i = 1 | j = -1 | key = 1

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
▶   arr[j+1] = key;
9 }
```



arr[0] = key = 1. Sorted part: [1, 5].

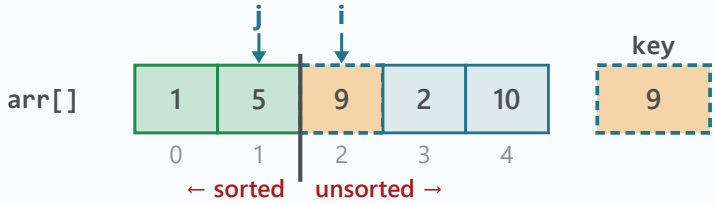
Step 6 / 20

VARIABLES

`n = 5` `i = 1` `j = -1` `key = 1`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
for(int i = 1; i < n; i++) {
  int key = arr[i];
  int j = i - 1;
  while(j >= 0 && arr[j] > key) {
    arr[j+1] = arr[j];
    j = j - 1;
  }
  arr[j+1] = key;
}
```



i = 2: `key` = arr[2] = 9, j = 1.

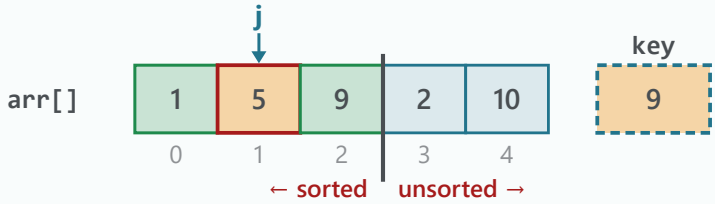
Step 7 / 20

VARIABLES

`n = 5` `i = 2` `j = 1` `key = 9`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
▶   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



arr[1] = 5 ≤ 9: the loop ends.

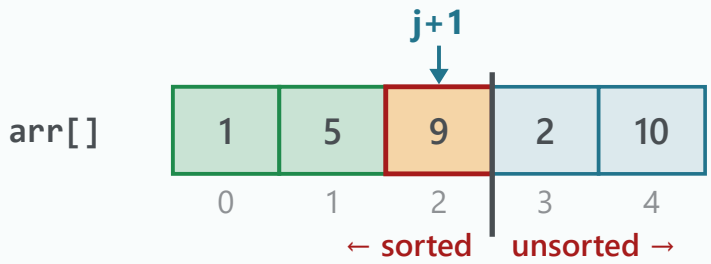
Step 8 / 20

VARIABLES

`n = 5` `i = 2` `j = 1` `key = 9`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
▶   arr[j+1] = key;
9 }
```



arr[2] = key = 9. Sorted part: [1, 5, 9].

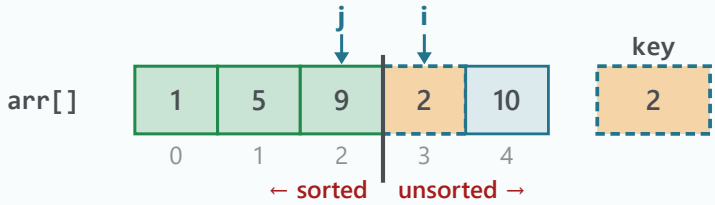
Step 9 / 20

VARIABLES

`n = 5` `i = 2` `j = 1` `key = 9`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
for(int i = 1; i < n; i++) {
  int key = arr[i];
  int j = i - 1;
  while(j >= 0 && arr[j] > key) {
    arr[j+1] = arr[j];
    j = j - 1;
  }
  arr[j+1] = key;
}
```



i = 3: `key` = arr[3] = 2, j = 2.

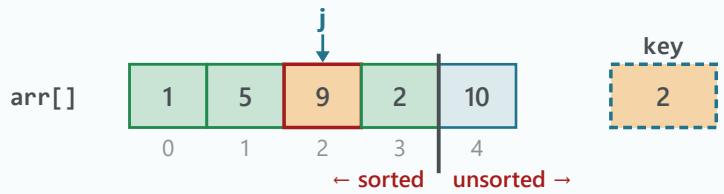
Step 10 / 20

VARIABLES

`n = 5` `i = 3` `j = 2` `key = 2`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
▶   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



j = 2 ≥ 0 and arr[2] = 9 > 2 → enter the loop.

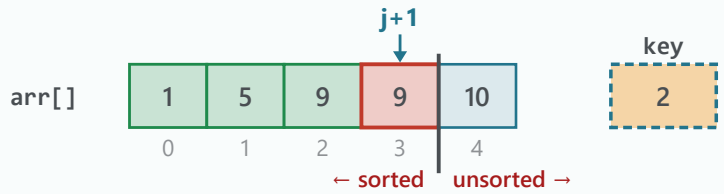
Step 11 / 20

VARIABLES

n = 5 | i = 3 | j = 2 | key = 2

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
      arr[j+1] = arr[j];
      j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



arr[3] = arr[2] = 9 (shift right), j = 1.

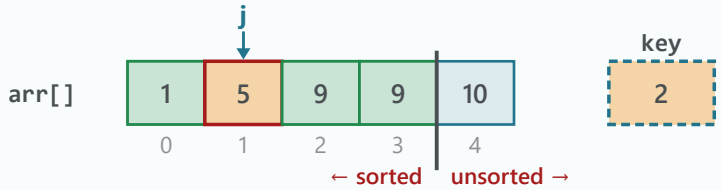
Step 12 / 20

VARIABLES

n = 5 | i = 3 | j = 1 | key = 2

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
▶   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



j = 1 ≥ 0 and arr[1] = 5 > 2 → enter the loop.

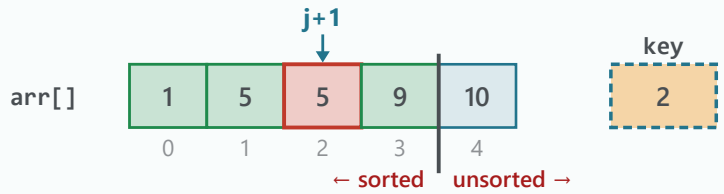
Step 13 / 20

VARIABLES

n = 5 | i = 3 | j = 1 | key = 2

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
      arr[j+1] = arr[j];
      j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



arr[2] = arr[1] = 5 (shift right), j = 0.

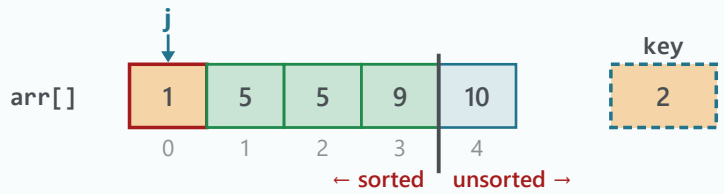
Step 14 / 20

VARIABLES

n = 5 | i = 3 | j = 0 | key = 2

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
▶   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



arr[0] = 1 ≤ 2: the loop ends.

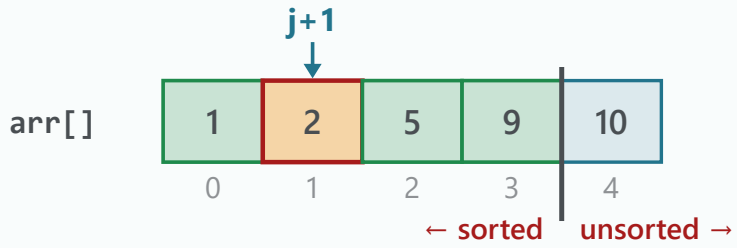
Step 15 / 20

VARIABLES

`n = 5` `i = 3` `j = 0` `key = 2`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
▶   arr[j+1] = key;
9 }
```



arr[1] = key = 2. Sorted part: [1, 2, 5, 9].

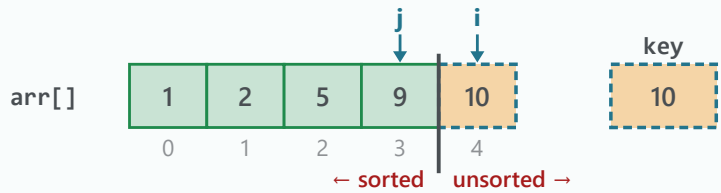
Step 16 / 20

VARIABLES

n = 5 | i = 3 | j = 0 | key = 2

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
for(int i = 1; i < n; i++) {
  int key = arr[i];
  int j = i - 1;
  while(j >= 0 && arr[j] > key) {
    arr[j+1] = arr[j];
    j = j - 1;
  }
  arr[j+1] = key;
}
```



i = 4: `key` = arr[4] = 10, j = 3.

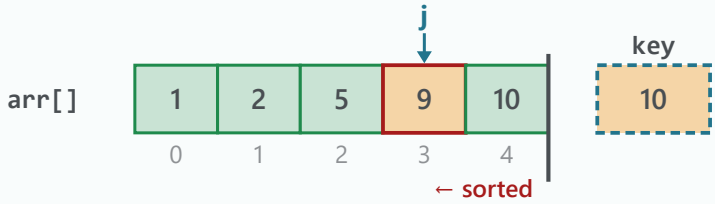
Step 17 / 20

VARIABLES

`n = 5` `i = 4` `j = 3` `key = 10`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
▶   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
8   arr[j+1] = key;
9 }
```



arr[3] = 9 ≤ 10: the loop ends.

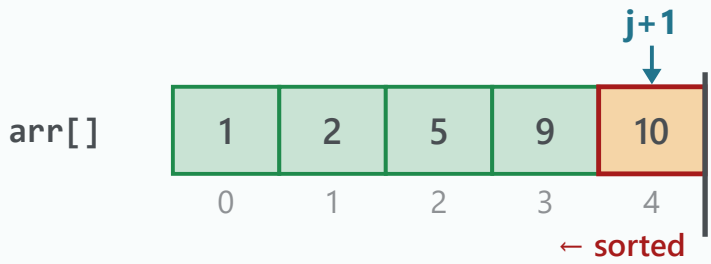
Step 18 / 20

VARIABLES

`n = 5` `i = 4` `j = 3` `key = 10`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
1 for(int i = 1; i < n; i++) {
2   int key = arr[i];
3   int j = i - 1;
4   while(j >= 0 && arr[j] > key) {
5     arr[j+1] = arr[j];
6     j = j - 1;
7   }
▶   arr[j+1] = key;
9 }
```



arr[4] = key = 10. Sorted part: [1, 2, 5, 9, 10].

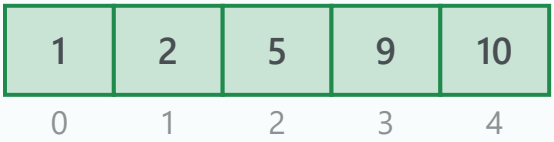
Step 19 / 20

VARIABLES

`n = 5` `i = 4` `j = 3` `key = 10`

Insertion Sort: Code Trace, arr = [5, 1, 9, 2, 10]

```
for(int i = 1; i < n; i++) {
  int key = arr[i];
  int j = i - 1;
  while(j >= 0 && arr[j] > key) {
    arr[j+1] = arr[j];
    j = j - 1;
  }
  arr[j+1] = key;
}
```



i = 5 is not < n → the loop ends. Result: [1, 2, 5, 9, 10].

Step 20 / 20

VARIABLES

n = 5 | i = 5 | j = – | key = –

Insertion Sort: Analysis

Case	Input	Comparisons	Shifts	Time
Best	sorted	$n - 1$	0	$\Theta(n)$
Average	random	$\approx n^2/4$	$\approx n^2/4$	$\Theta(n^2)$
Worst	reverse sorted	$n(n - 1)/2$	$n(n - 1)/2$	$\Theta(n^2)$

Pros

+ Stable, in place, online.
+ Adaptive: time $\Theta(n +$ number of inversions$)$.
+ Among the fastest on small arrays ($n \lesssim$ 20–50).

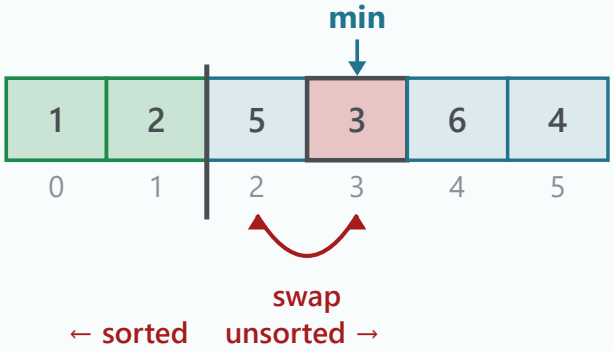
Cons

– $\Theta(n^2)$ on large random arrays.
– An item moves only one cell per step; a distant item needs many shifts (→ Shell sort).

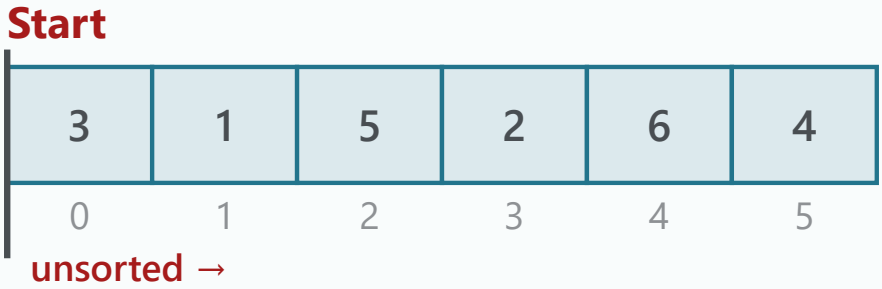
Over all inputs we can only say $O(n^2)$ and $\Omega(n)$; there is no single Θ. Timsort and many libraries hand small subarrays in quicksort and merge sort over to insertion sort.

Selection Sort

- The array has two parts: **sorted** and **unsorted**.
- At each step:
  - Find the **smallest** item in the unsorted part.
  - Swap it with the item at the **front** of the unsorted part.
  - After the swap it joins the sorted part.
- n − 1 passes in total; **at most one** swap per pass.

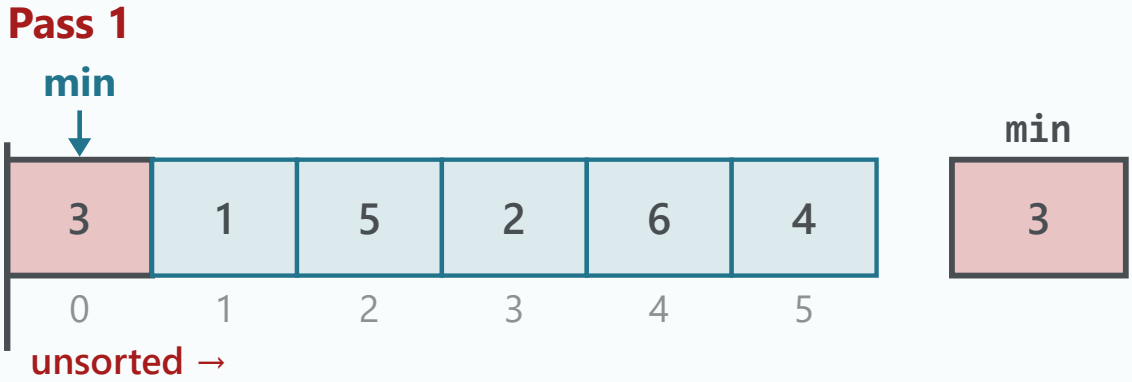


Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



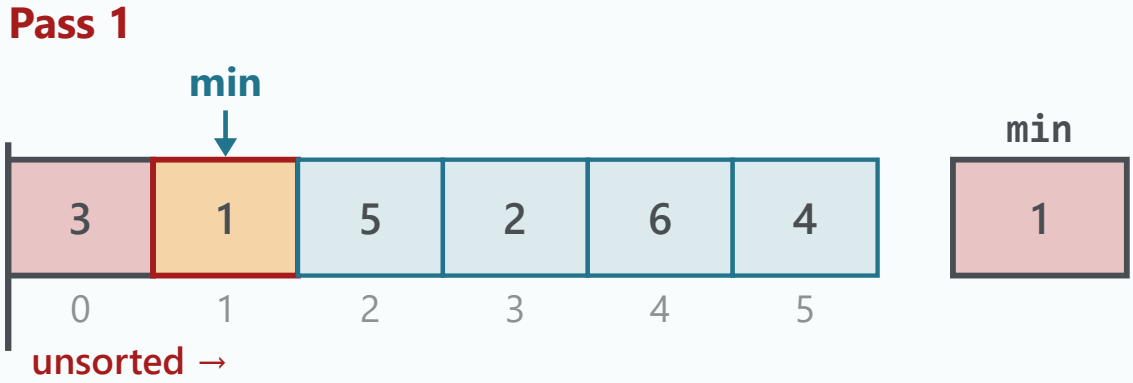
Array [3, 1, 5, 2, 6, 4], length = 6. The sorted part starts empty.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



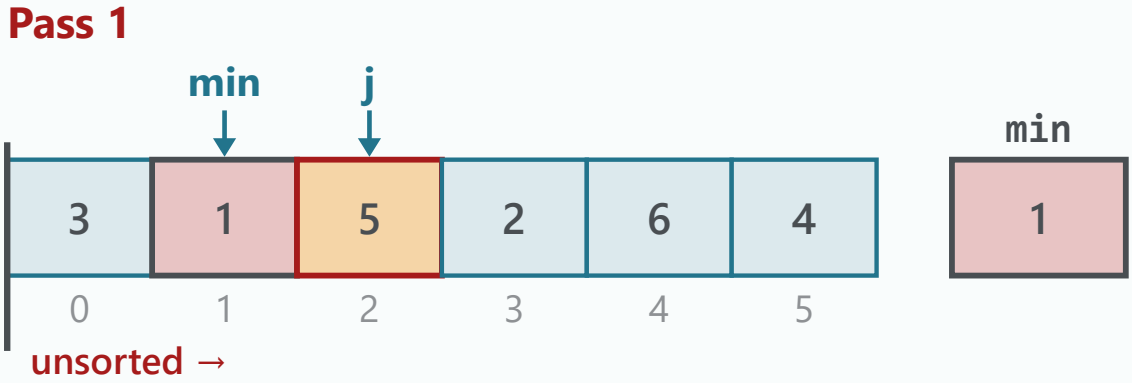
Pass 1: the first unsorted item is the tentative minimum: min = 3.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



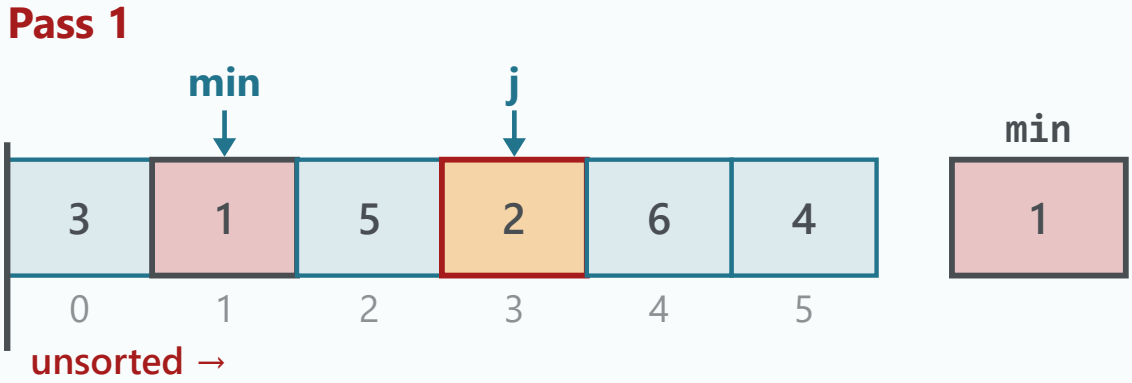
1 < min (3)? **Yes** → min = 1.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



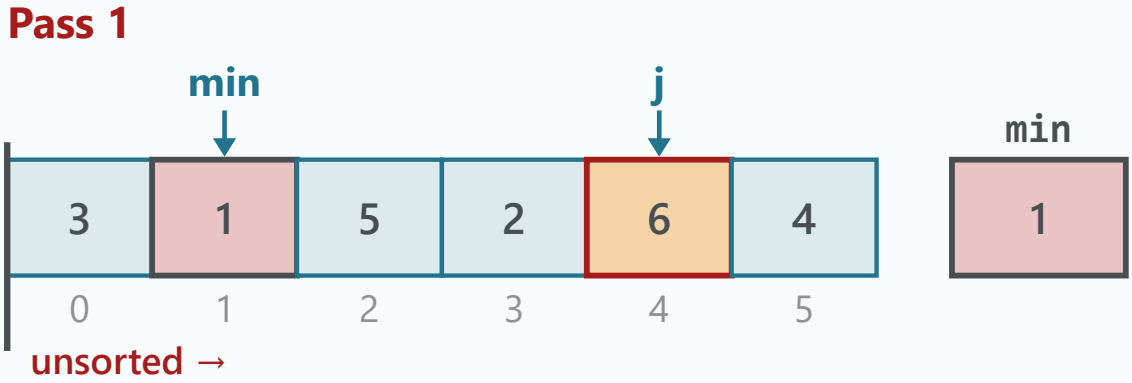
5 < min (1)? No.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



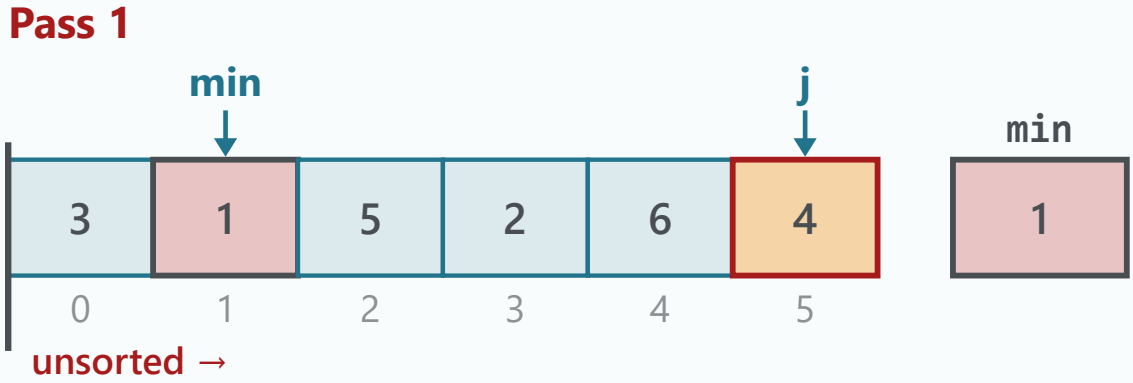
2 < min (1)? No.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



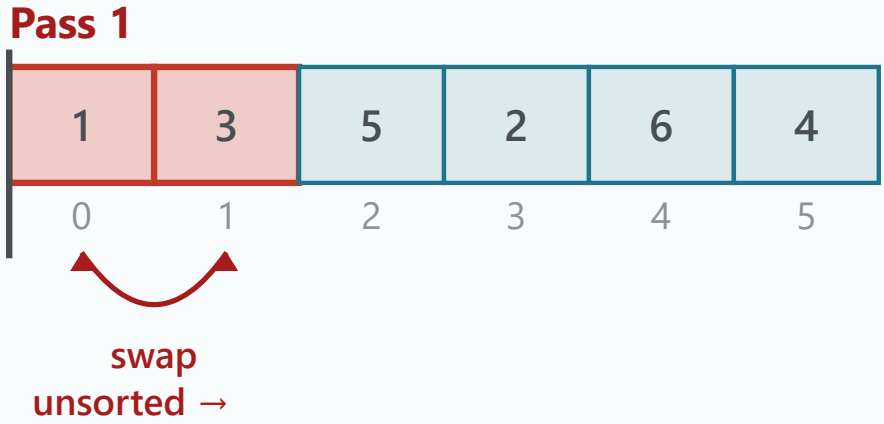
6 < min (1)? No.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



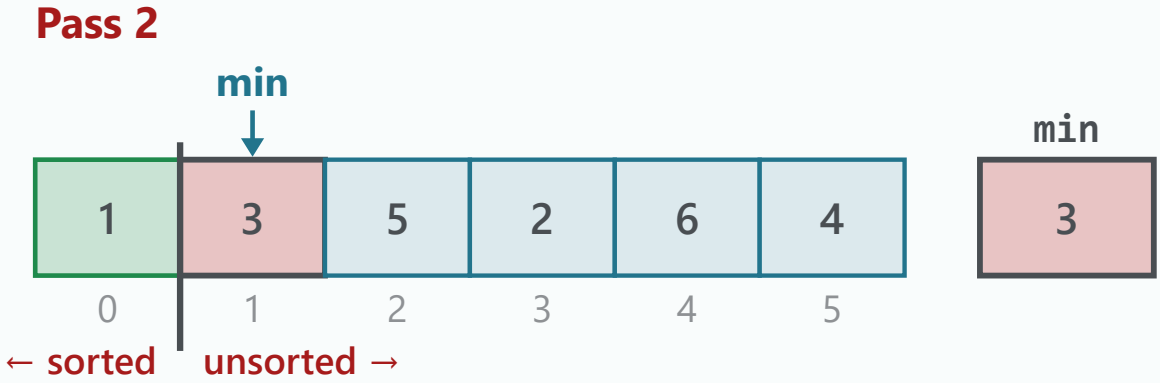
4 < min (1)? No.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



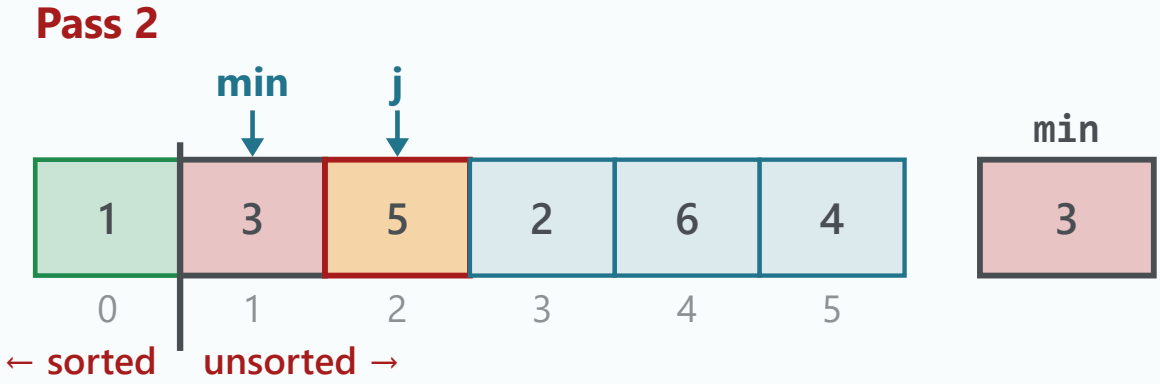
The minimum (1) is swapped with the front of the unsorted part → [1, 3, 5, 2, 6, 4]; the sorted part grows by one.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



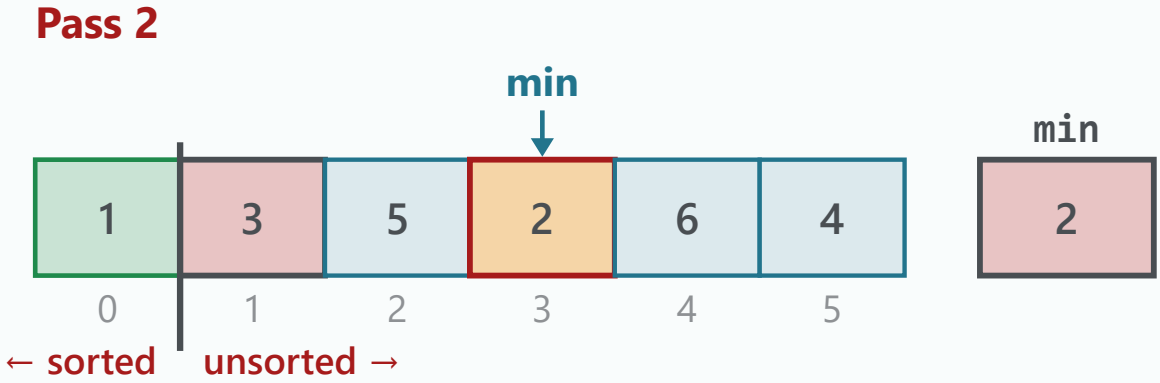
Pass 2: the first unsorted item is the tentative minimum: min = 3.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



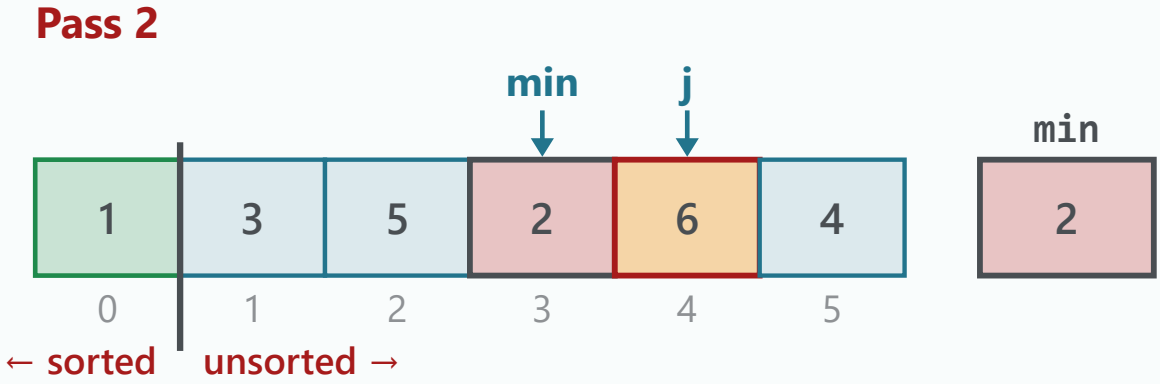
5 < min (3)? No.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



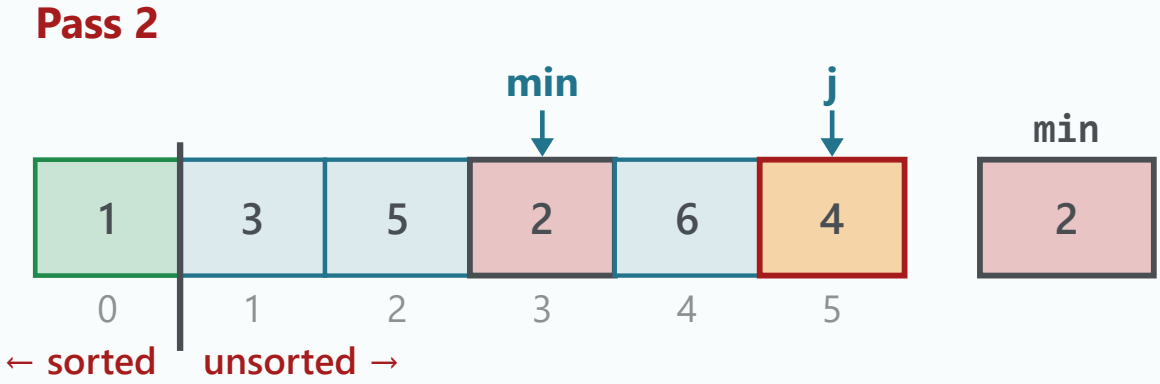
2 < min (3)? **Yes** → min = 2.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



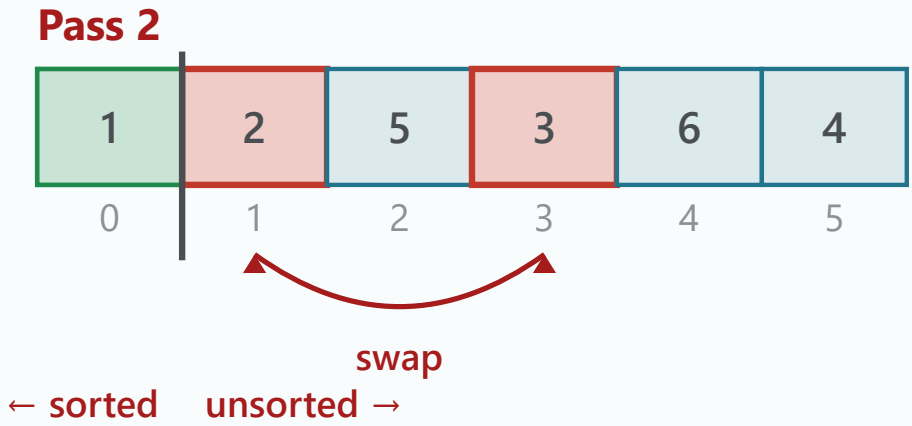
6 < min (2)? No.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



4 < min (2)? No.

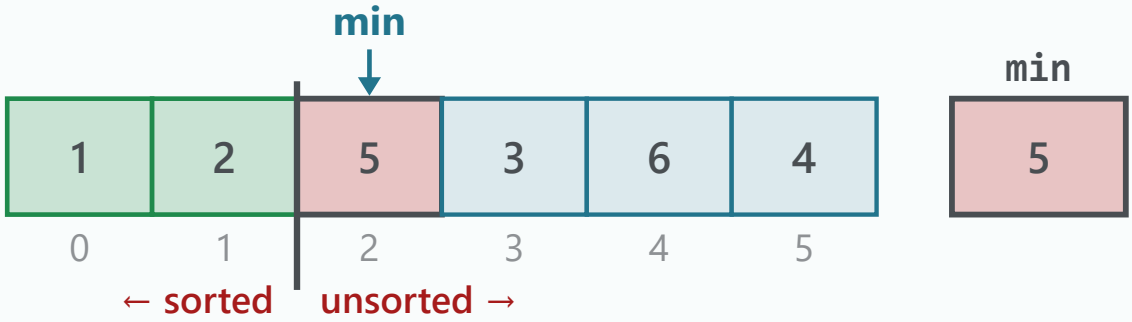
Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



The minimum (2) is swapped with the front of the unsorted part → [1, 2, 5, 3, 6, 4]; the sorted part grows by one.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

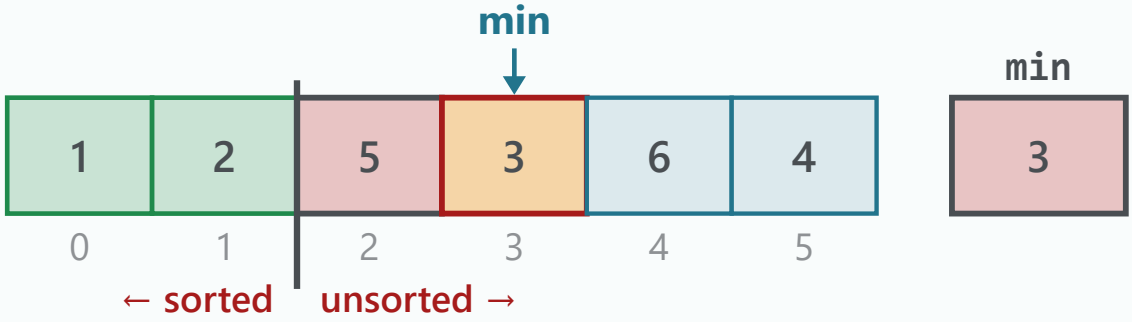
Pass 3



Pass 3: the first unsorted item is the tentative minimum: min = 5.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

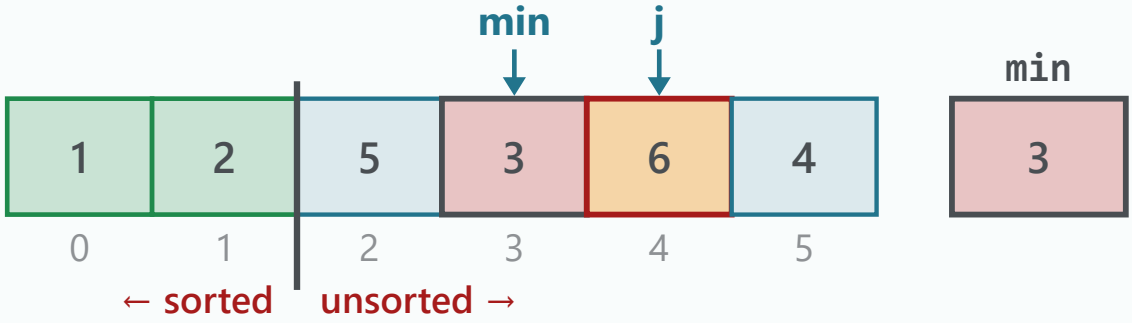
Pass 3



3 < min (5)? **Yes** → min = 3.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

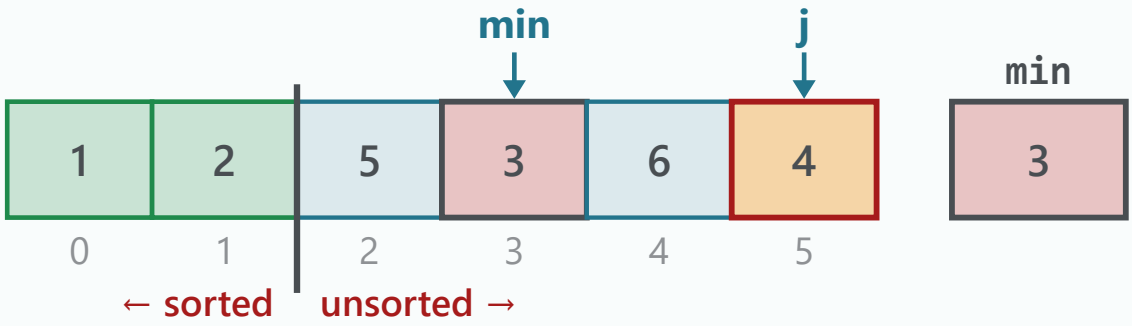
Pass 3



6 < min (3)? No.

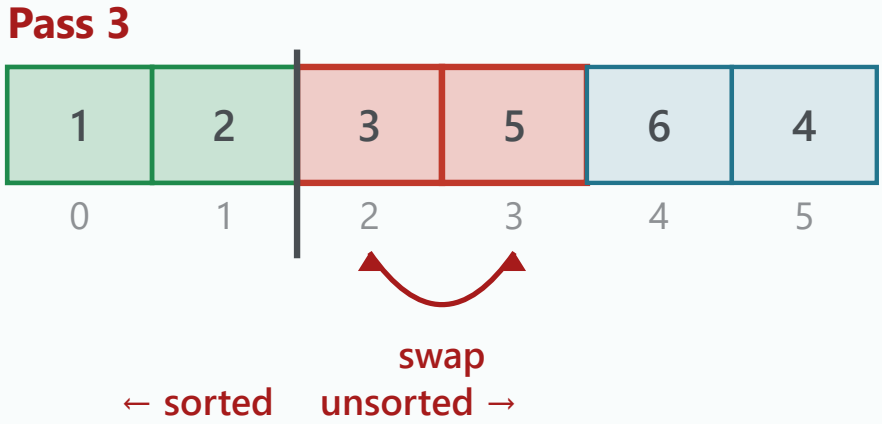
Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

Pass 3



4 < min (3)? No.

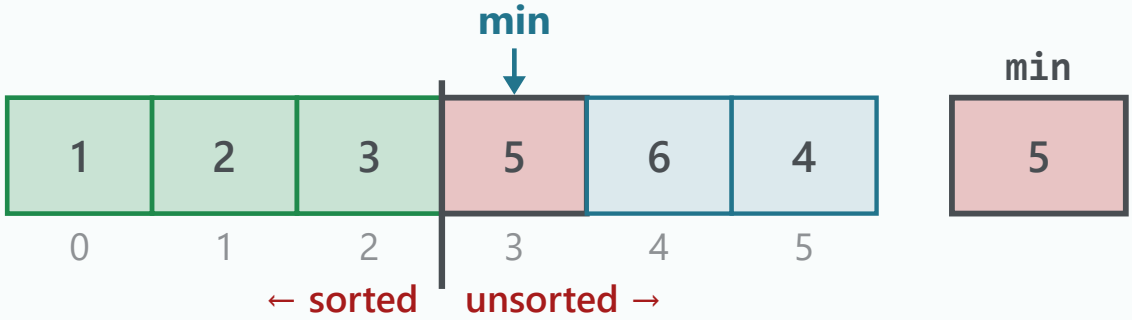
Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



The minimum (3) is swapped with the front of the unsorted part → [1, 2, 3, 5, 6, 4]; the sorted part grows by one.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

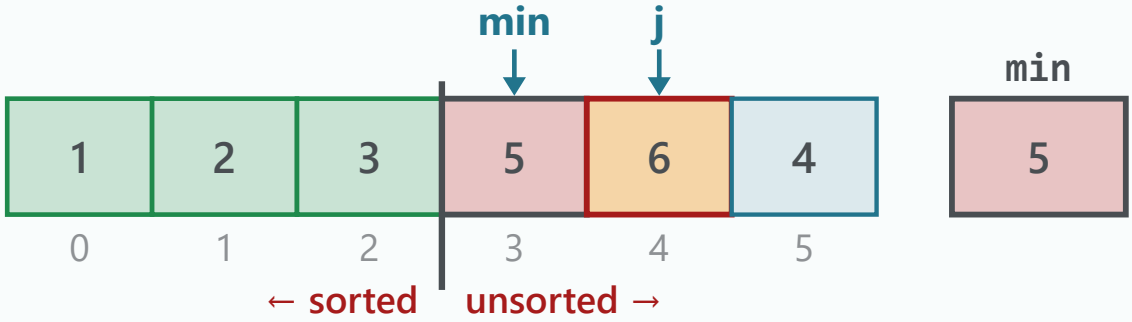
Pass 4



Pass 4: the first unsorted item is the tentative minimum: min = 5.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

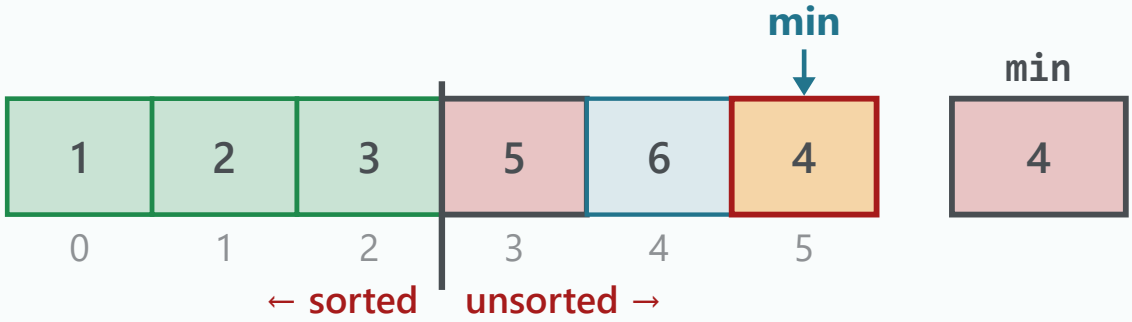
Pass 4



6 < min (5)? No.

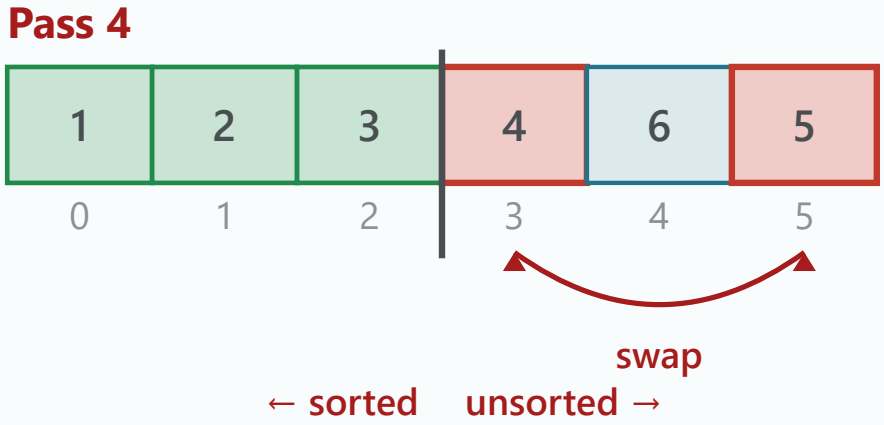
Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

Pass 4



4 < min (5)? **Yes** → min = 4.

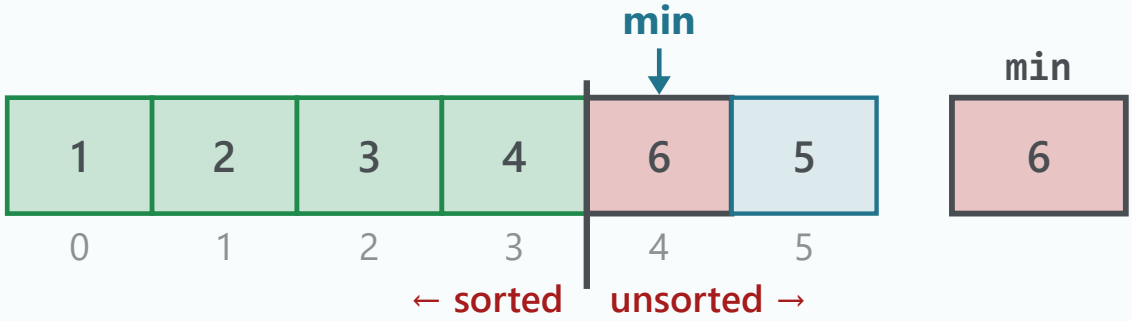
Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



The minimum (4) is swapped with the front of the unsorted part → [1, 2, 3, 4, 6, 5]; the sorted part grows by one.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

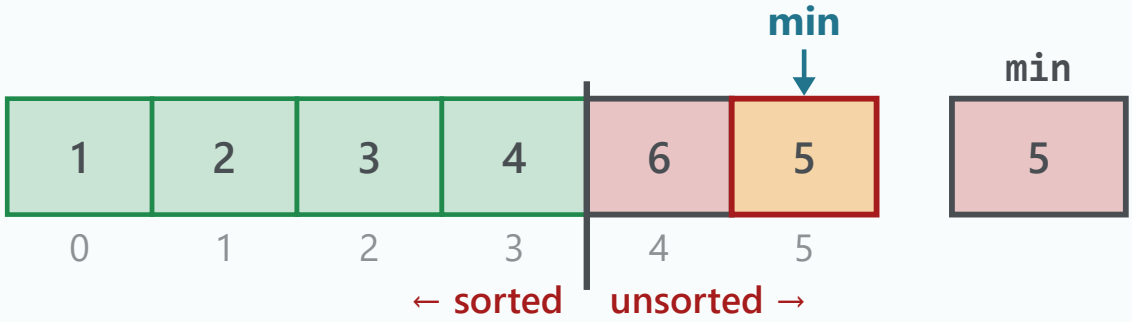
Pass 5



Pass 5: the first unsorted item is the tentative minimum: min = 6.

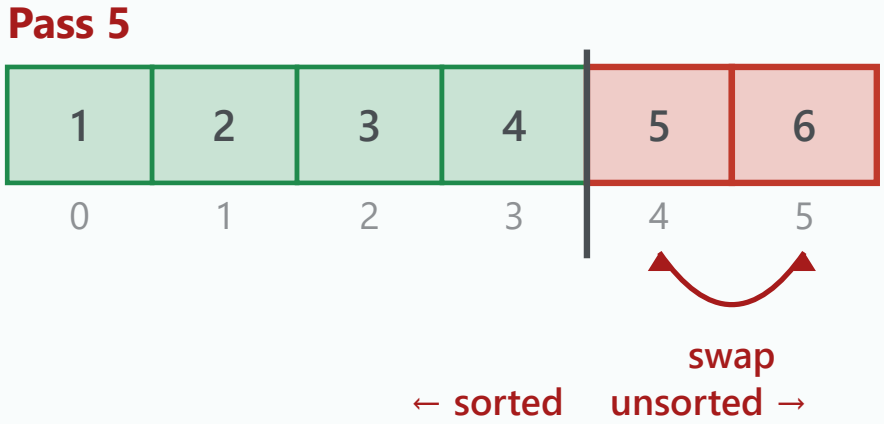
Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

Pass 5



5 < min (6)? **Yes** → min = 5.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]



The minimum (5) is swapped with the front of the unsorted part → [1, 2, 3, 4, 5, 6]; the sorted part grows by one.

Selection Sort Step by Step: [3, 1, 5, 2, 6, 4]

Result



After 5 passes the last item is in place by itself. Sorted array: [1, 2, 3, 4, 5, 6].

Selection Sort: Java Code

```java
public void sort(int[] arr) {
  int n = arr.length;
  for(int i = 0; i < n - 1; i++) {
    int minIdx = i;
    for(int j = i + 1; j < n; j++) {
      if(arr[j] < arr[minIdx]) {
        minIdx = j;
      }
    }
    int temp = arr[minIdx];
    arr[minIdx] = arr[i];
    arr[i] = temp;
  }
}
```

- `minIdx` holds the **index** of the smallest item (not its value): the swap at the end needs the index.
- The inner loop scans the whole unsorted part; the scan is no shorter even if the input is sorted.
- One swap per pass: at most **n − 1** in total.

Where writes are expensive (e.g. flash memory), the small number of swaps is selection sort's only real advantage.

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
▶    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



n = arr.length = 5.

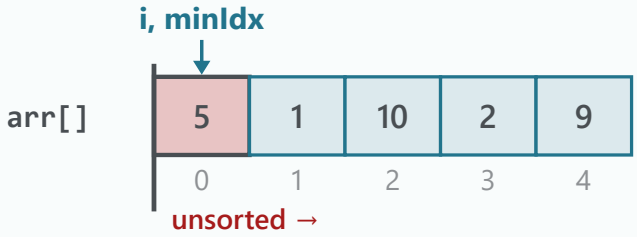
Step 1 / 24

VARIABLES

n = 5 | i = – | minIdx = – | j = –

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
▶    for(int i = 0; i < n - 1; i++) {
▶      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



i = 0: minIdx = 0 (arr[minIdx] = 5).

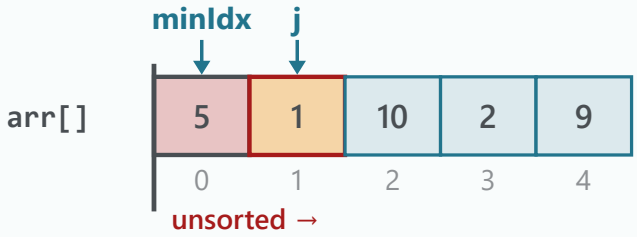
Step 2 / 24

VARIABLES

n = 5 | i = 0 | minIdx = 0 | j = –

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 1: arr[1] = 1 < arr[minIdx] = 5? **Yes**.

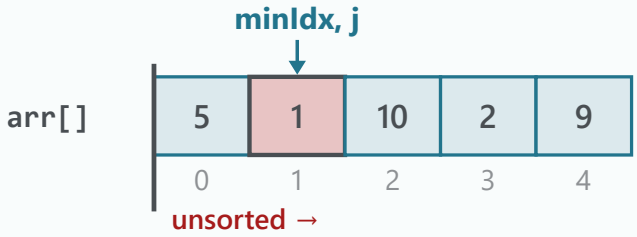
Step 3 / 24

VARIABLES

`n = 5` `i = 0` `minIdx = 0` `j = 1`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
▶          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



minIdx = 1 (new minimum: 1).

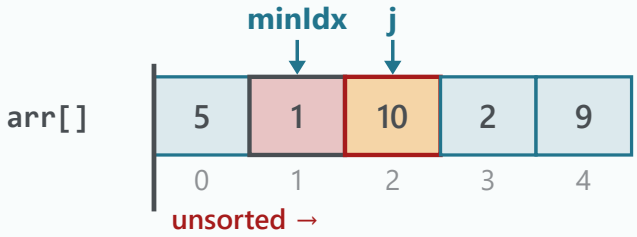
Step 4 / 24

VARIABLES

`n = 5` `i = 0` `minIdx = 1` `j = 1`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 2: arr[2] = 10 < arr[minIdx] = 1? No.

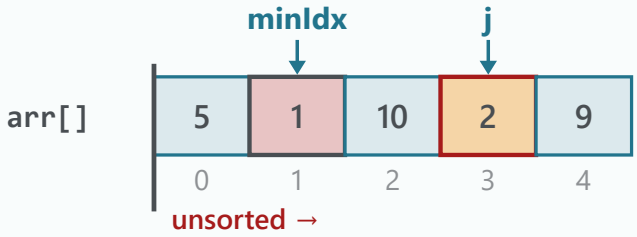
Step 5 / 24

VARIABLES

`n = 5` `i = 0` `minIdx = 1` `j = 2`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 3: arr[3] = 2 < arr[minIdx] = 1? No.

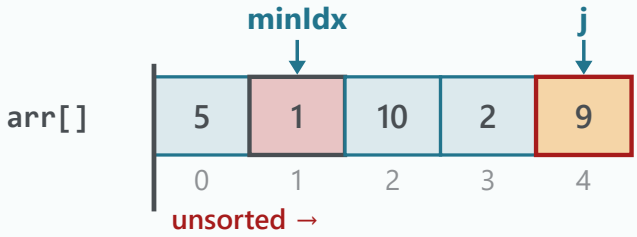
Step 6 / 24

VARIABLES

`n = 5` `i = 0` `minIdx = 1` `j = 3`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 4: arr[4] = 9 < arr[minIdx] = 1? No.

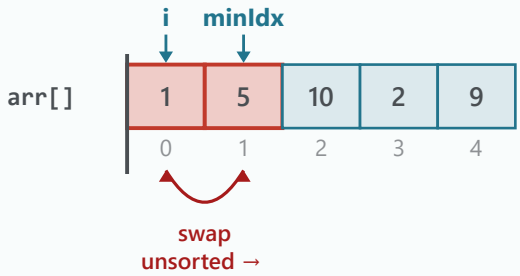
Step 7 / 24

VARIABLES

`n = 5` `i = 0` `minIdx = 1` `j = 4`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
▶      int temp = arr[minIdx];
▶      arr[minIdx] = arr[i];
▶      arr[i] = temp;
13   }
14 }
```



j = 5 → the inner loop ends. `temp` = 1; arr[1] ↔ arr[0] → [1, 5, 10, 2, 9].

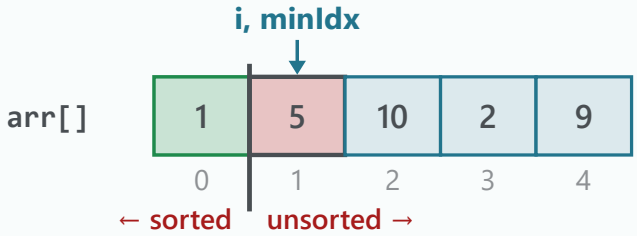
Step 8 / 24

VARIABLES

`n = 5` `i = 0` `minIdx = 1` `j = 5` `temp = 1`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
▶    for(int i = 0; i < n - 1; i++) {
▶      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



i = 1: minIdx = 1 (arr[minIdx] = 5).

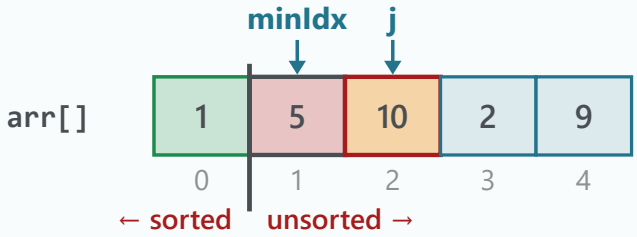
Step 9 / 24

VARIABLES

n = 5 | i = 1 | minIdx = 1 | j = –

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 2: arr[2] = 10 < arr[minIdx] = 5? No.

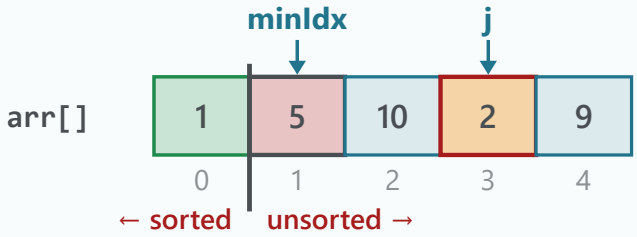
Step 10 / 24

VARIABLES

`n = 5` `i = 1` `minIdx = 1` `j = 2`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 3: arr[3] = 2 < arr[minIdx] = 5? **Yes**.

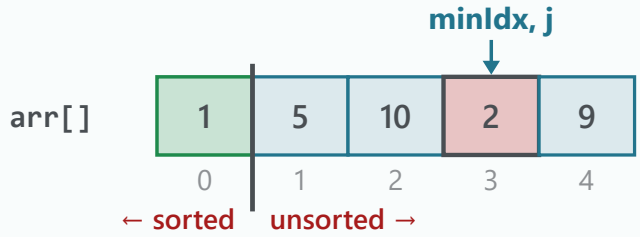
Step 11 / 24

VARIABLES

`n = 5` `i = 1` `minIdx = 1` `j = 3`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
▶          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



minIdx = 3 (new minimum: 2).

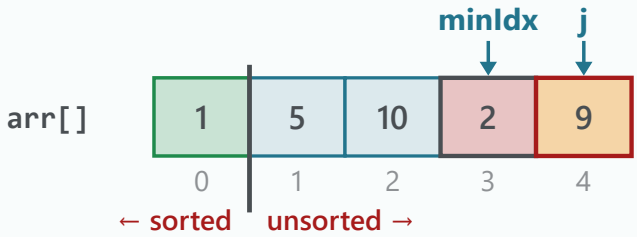
Step 12 / 24

VARIABLES

`n = 5` `i = 1` `minIdx = 3` `j = 3`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 4: arr[4] = 9 < arr[minIdx] = 2? No.

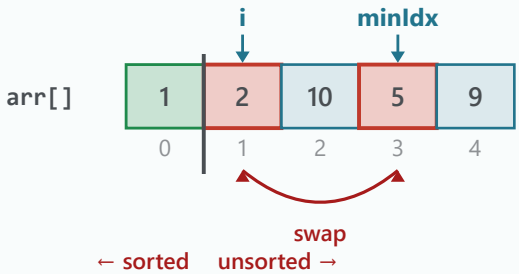
Step 13 / 24

VARIABLES

`n = 5` `i = 1` `minIdx = 3` `j = 4`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
▶      int temp = arr[minIdx];
▶      arr[minIdx] = arr[i];
▶      arr[i] = temp;
13   }
14 }
```



j = 5 → the inner loop ends. `temp` = 2; arr[3] ↔ arr[1] → [1, 2, 10, 5, 9].

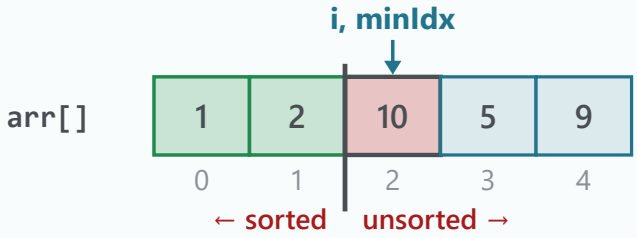
Step 14 / 24

VARIABLES

`n = 5` `i = 1` `minIdx = 3` `j = 5` `temp = 2`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
▶    for(int i = 0; i < n - 1; i++) {
▶      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



i = 2: minIdx = 2 (arr[minIdx] = 10).

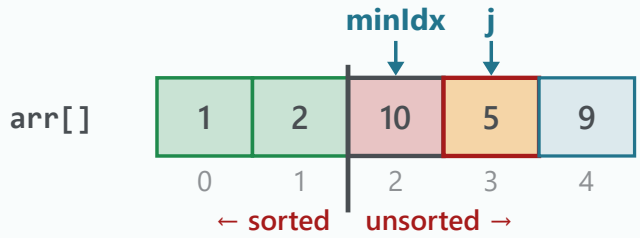
Step 15 / 24

VARIABLES

n = 5 | i = 2 | minIdx = 2 | j = –

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 3: arr[3] = 5 < arr[minIdx] = 10? **Yes**.

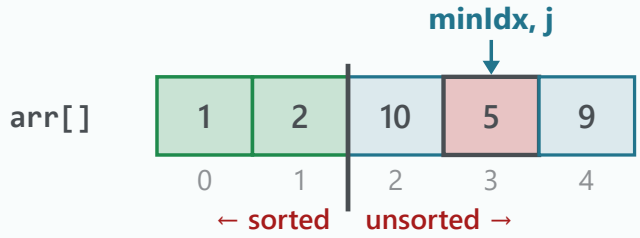
Step 16 / 24

VARIABLES

`n = 5` `i = 2` `minIdx = 2` `j = 3`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
▶          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



minIdx = 3 (new minimum: 5).

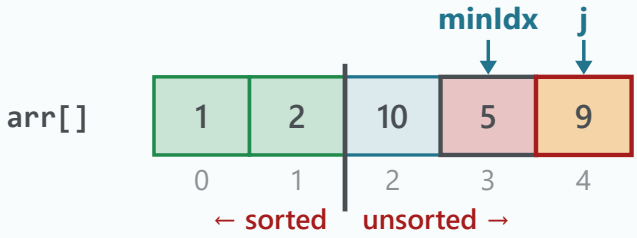
Step 17 / 24

VARIABLES

`n = 5` `i = 2` `minIdx = 3` `j = 3`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 4: arr[4] = 9 < arr[minIdx] = 5? No.

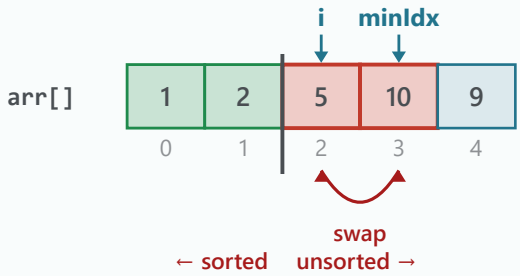
Step 18 / 24

VARIABLES

`n = 5` `i = 2` `minIdx = 3` `j = 4`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
▶      int temp = arr[minIdx];
▶      arr[minIdx] = arr[i];
▶      arr[i] = temp;
13   }
14 }
```



j = 5 → the inner loop ends. `temp` = 5; arr[3] ↔ arr[2] → [1, 2, 5, 10, 9].

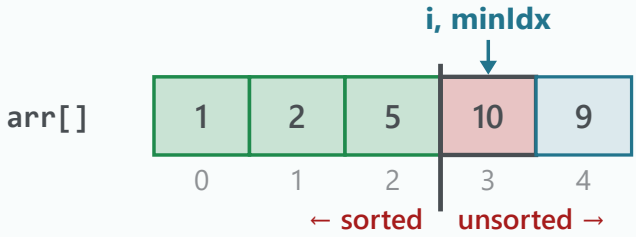
Step 19 / 24

VARIABLES

`n = 5` `i = 2` `minIdx = 3` `j = 5` `temp = 5`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
▶    for(int i = 0; i < n - 1; i++) {
▶      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



i = 3: minIdx = 3 (arr[minIdx] = 10).

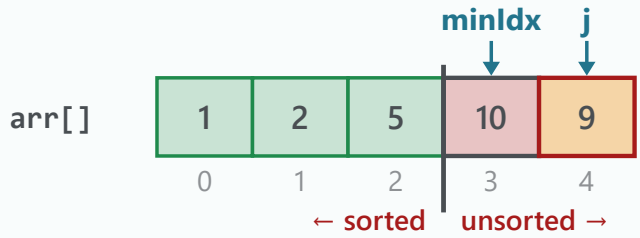
Step 20 / 24

VARIABLES

n = 5 | i = 3 | minIdx = 3 | j = –

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
▶      for(int j = i + 1; j < n; j++) {
▶        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



j = 4: arr[4] = 9 < arr[minIdx] = 10? **Yes**.

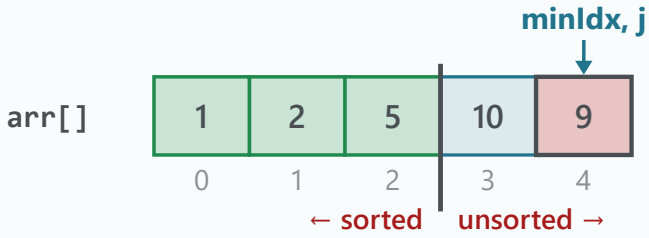
Step 21 / 24

VARIABLES

`n = 5` `i = 3` `minIdx = 3` `j = 4`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
▶          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



minIdx = 4 (new minimum: 9).

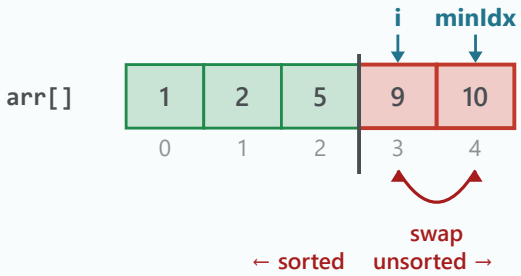
Step 22 / 24

VARIABLES

`n = 5` `i = 3` `minIdx = 4` `j = 4`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
3    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
▶      int temp = arr[minIdx];
▶      arr[minIdx] = arr[i];
▶      arr[i] = temp;
13   }
14 }
```



j = 5 → the inner loop ends. `temp` = 9; arr[4] ↔ arr[3] → [1, 2, 5, 9, 10].

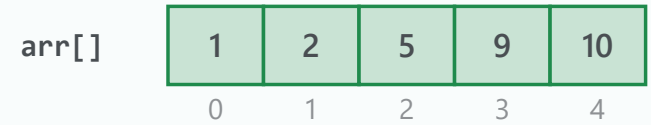
Step 23 / 24

VARIABLES

`n = 5` `i = 3` `minIdx = 4` `j = 5` `temp = 9`

Selection Sort: Code Trace, arr = [5, 1, 10, 2, 9]

```
1  public void sort(int[] arr) {
2    int n = arr.length;
▶    for(int i = 0; i < n - 1; i++) {
4      int minIdx = i;
5      for(int j = i + 1; j < n; j++) {
6        if(arr[j] < arr[minIdx]) {
7          minIdx = j;
8        }
9      }
10     int temp = arr[minIdx];
11     arr[minIdx] = arr[i];
12     arr[i] = temp;
13   }
14 }
```



i = 4 is not < n − 1 → done. Result: [1, 2, 5, 9, 10].

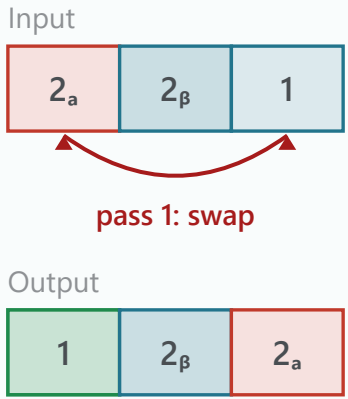
Step 24 / 24

VARIABLES

n = 5 | i = 4 | minIdx = – | j = –

Selection Sort: Analysis

Measure	Value
Comparisons	$(n - 1) + (n - 2) + \ldots + 1 = \mathbf{n(n - 1)/2}$ on *every* input
Swaps	at most $n - 1$
Time	best = average = worst = $\mathbf{\Theta(n^2)}$
Extra memory	O(1), in place
Stable?	**No** (long-distance swaps)



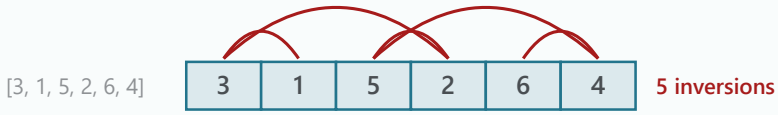
The minimum (1) is swapped with 2_a at the front: 2_a ends up **behind** 2_β.

Not adaptive: it spends $n(n - 1)/2$ comparisons even on a sorted array.

Bubble, Insertion, Selection: Same Input

Cells: **comparisons / swaps (or shifts)**, n = 6. Values were counted by running the code.

Algorithm	Move type	[3, 1, 5, 2, 6, 4]	sorted 1…6	reverse 6…1
Bubble (with flag)	swap	12 / 5	5 / 0	15 / 15
Insertion	shift	9 / 5	5 / 0	15 / 15
Selection	swap	15 / 5	15 / 0	15 / 3



- In bubble and insertion sort the number of moves = the number of **inversions** (5 for [3, 1, 5, 2, 6, 4]); a swap is 3 assignments, a shift is 1.
- Selection sort always makes n(n − 1)/2 = 15 comparisons regardless of the input, but it makes the fewest swaps.
- On sorted input, flagged bubble sort and insertion sort make only n − 1 = 5 comparisons (adaptive).

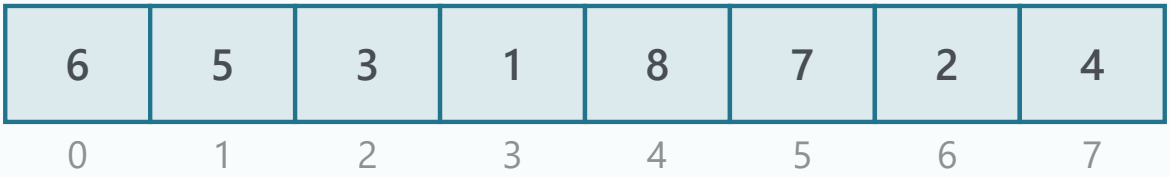
Shell Sort

- D. Shell (1959): a **generalization** of insertion sort.
- First, items a **gap** apart are insertion-sorted; small items far away jump a long distance in one move.
- The gap shrinks (n/2, n/4, …); the last pass has **gap = 1**, i.e. plain insertion sort — but by now the array is nearly sorted.
- In place, **unstable**; the time depends on the gap sequence: worst case $\Theta(n^2)$ with n/2 gaps, $O(n^{1.5})$ with Hibbard's $(2^k - 1)$.

```
void shellSort(int[] arr) {
  int n = arr.length;
  for (int gap = n / 2; gap > 0; gap /= 2)
    for (int i = gap; i < n; i++) {
      int key = arr[i], j;
      for (j = i; j >= gap && arr[j - gap] > key; j -= gap)
        arr[j] = arr[j - gap];
      arr[j] = key;
    }
}
```

Shell Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]

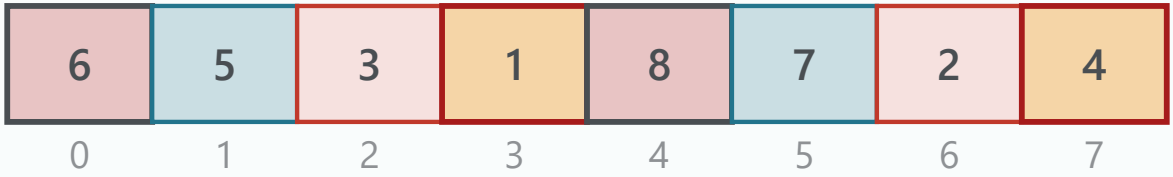
Start



Array [6, 5, 3, 1, 8, 7, 2, 4], n = 8. Gap sequence: n/2 = 4, 2, … , 1.

Shell Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]

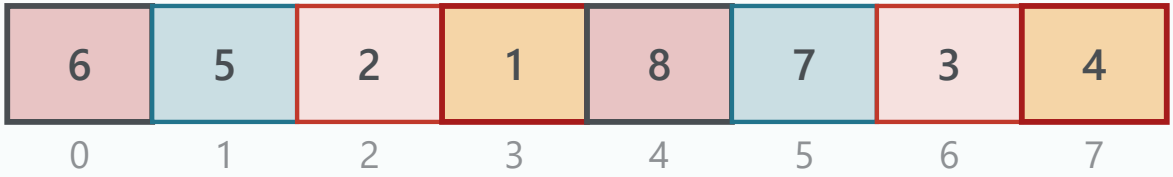
gap = 4: groups



gap = 4: items 4 indices apart form a group (same color): [6, 8], [5, 7], [3, 2], [1, 4].

Shell Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]

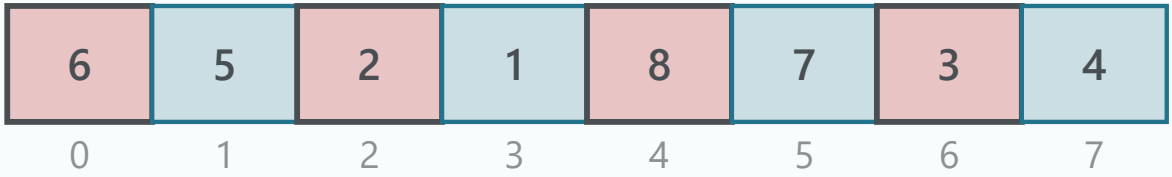
gap = 4: after



Each group is insertion-sorted (1 shift): [6, 8], [5, 7], [2, 3], [1, 4] → [6, 5, 2, 1, 8, 7, 3, 4].

Shell Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]

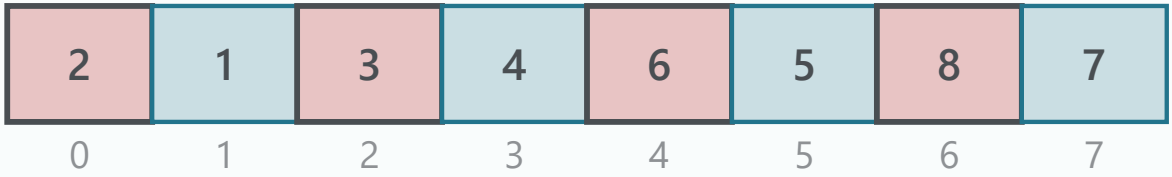
gap = 2: groups



gap = 2: items 2 indices apart form a group (same color): [6, 2, 8, 3], [5, 1, 7, 4].

Shell Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]

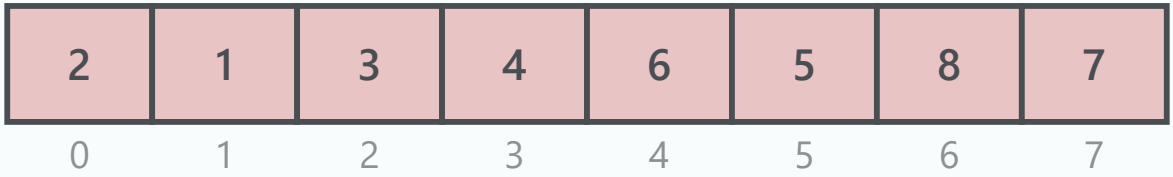
gap = 2: after



Each group is insertion-sorted (6 shifts): [2, 3, 6, 8], [1, 4, 5, 7] → [2, 1, 3, 4, 6, 5, 8, 7].

Shell Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]

gap = 1: groups



gap = 1: items 1 index apart form a group (same color): [2, 1, 3, 4, 6, 5, 8, 7].

Shell Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]

gap = 1: after



gap = 1 is plain insertion sort; since the array is nearly sorted, only 3 shifts were needed. Result: [1, 2, 3, 4, 5, 6, 7, 8].

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
▶    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
4      for (int i = gap; i < n; i++) {
5        int key = arr[i], j;
6        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
7          arr[j] = arr[j - gap];
8        arr[j] = key;
9      }
10 }
```



gap = 3: same marks form a group

arr = [5, 2, 6, 1, 4, 3], n = 6. The first gap is n / 2 = 3: items 3 apart, (0, 3), (1, 4), (2, 5), are insertion-sorted as separate groups.

Step 1 / 10

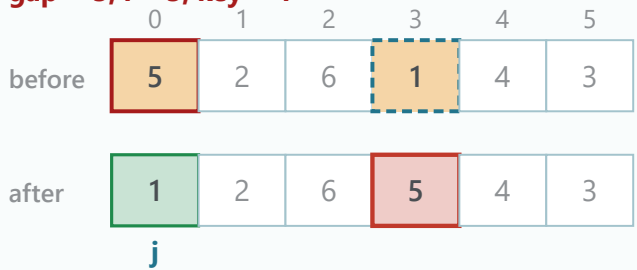
VARIABLES

`n = 6` `gap = –` `i = –` `key = –` `j = –`

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
4      for (int i = gap; i < n; i++) {
▶        int key = arr[i], j;
▶        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
▶          arr[j] = arr[j - gap];
▶        arr[j] = key;
9      }
10 }
```

gap = 3, i = 3, key = 1



i = 3: key = 1. arr[0] = 5 > 1 → moves to index 3. Stop at j = 0 and write 1 there.

Step 2 / 10

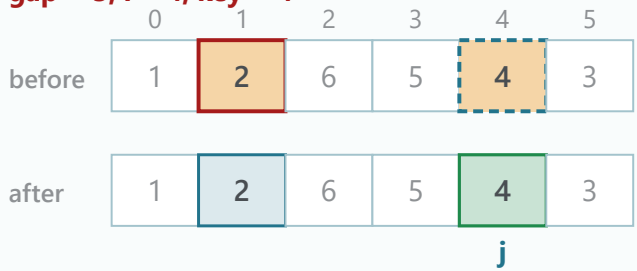
VARIABLES

`n = 6` `gap = 3` `i = 3` `key = 1` `j = 0`

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
       for (int i = gap; i < n; i++) {
         int key = arr[i], j;
         for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
7          arr[j] = arr[j - gap];
         arr[j] = key;
9      }
10 }
```

gap = 3, i = 4, key = 4



i = 4: key = 4. arr[1] = 2 ≤ 4: no shift, 4 stays.

Step 3 / 10

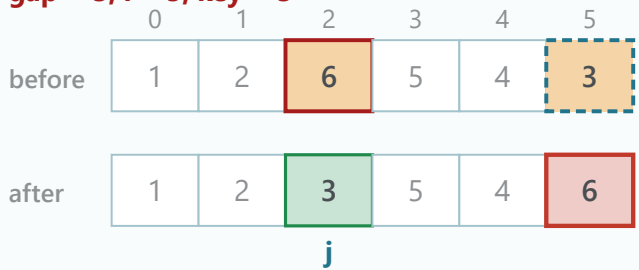
VARIABLES

n = 6 | gap = 3 | i = 4 | key = 4 | j = 4

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
4      for (int i = gap; i < n; i++) {
▶        int key = arr[i], j;
▶        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
▶          arr[j] = arr[j - gap];
▶        arr[j] = key;
9      }
10 }
```

gap = 3, i = 5, key = 3



i = 5: key = 3. arr[2] = 6 > 3 → moves to index 5. Stop at j = 2 and write 3 there.

Step 4 / 10

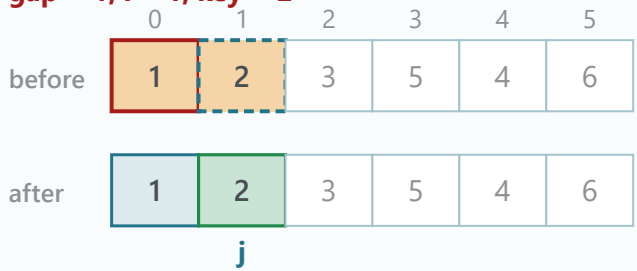
VARIABLES

`n = 6` `gap = 3` `i = 5` `key = 3` `j = 2`

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
▶      for (int i = gap; i < n; i++) {
▶        int key = arr[i], j;
▶        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
7          arr[j] = arr[j - gap];
▶        arr[j] = key;
9      }
10 }
```

gap = 1, i = 1, key = 2



gap = 3 / 2 = 1: now plain insertion sort. i = 1: key = 2. arr[0] = 1 ≤ 2: no shift, 2 stays.

Step 5 / 10

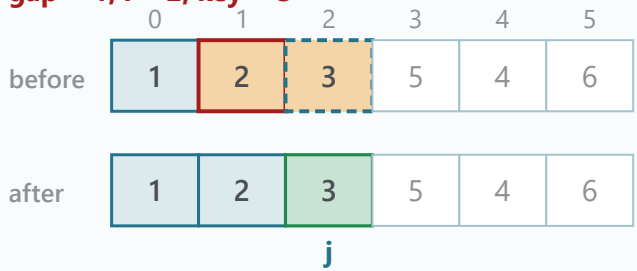
VARIABLES

`n = 6` `gap = 1` `i = 1` `key = 2` `j = 1`

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
▶      for (int i = gap; i < n; i++) {
▶        int key = arr[i], j;
▶        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
7          arr[j] = arr[j - gap];
▶        arr[j] = key;
9      }
10 }
```

gap = 1, i = 2, key = 3



i = 2: key = 3. arr[1] = 2 ≤ 3: no shift, 3 stays.

Step 6 / 10

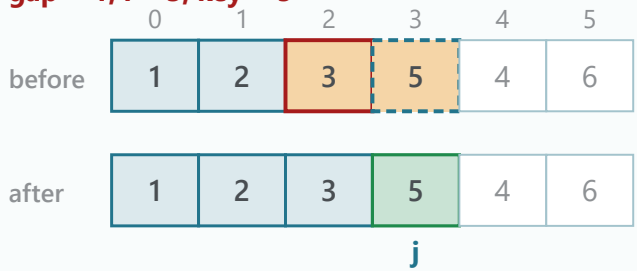
VARIABLES

`n = 6` `gap = 1` `i = 2` `key = 3` `j = 2`

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
▶      for (int i = gap; i < n; i++) {
▶        int key = arr[i], j;
▶        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
7          arr[j] = arr[j - gap];
▶        arr[j] = key;
9      }
10 }
```

gap = 1, i = 3, key = 5



i = 3: key = 5. arr[2] = 3 ≤ 5: no shift, 5 stays.

Step 7 / 10

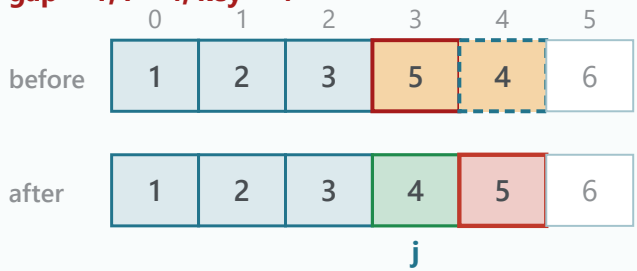
VARIABLES

`n = 6` `gap = 1` `i = 3` `key = 5` `j = 3`

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
4      for (int i = gap; i < n; i++) {
▶        int key = arr[i], j;
▶        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
▶          arr[j] = arr[j - gap];
▶        arr[j] = key;
9      }
10 }
```

gap = 1, i = 4, key = 4



i = 4: key = 4. arr[3] = 5 > 4 → moves to index 4. arr[2] = 3 ≤ 4, stop; write 4 there.

Step 8 / 10

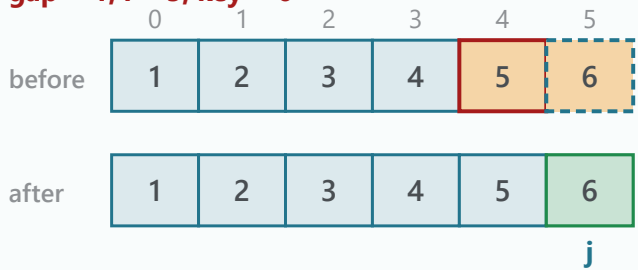
VARIABLES

`n = 6` `gap = 1` `i = 4` `key = 4` `j = 3`

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
3    for (int gap = n / 2; gap > 0; gap /= 2)
       for (int i = gap; i < n; i++) {
         int key = arr[i], j;
         for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
7          arr[j] = arr[j - gap];
         arr[j] = key;
9      }
10 }
```

gap = 1, i = 5, key = 6



i = 5: key = 6. arr[4] = 5 ≤ 6: no shift, 6 stays.

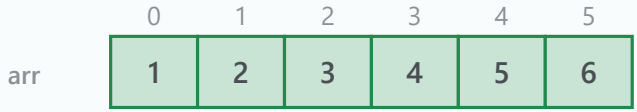
Step 9 / 10

VARIABLES

n = 6 | gap = 1 | i = 5 | key = 6 | j = 5

Shell Sort: Code Trace, arr = [5, 2, 6, 1, 4, 3]

```
1  void shellSort(int[] arr) {
2    int n = arr.length;
▶    for (int gap = n / 2; gap > 0; gap /= 2)
4      for (int i = gap; i < n; i++) {
5        int key = arr[i], j;
6        for (j = i; j >= gap && arr[j - gap] >
   key; j -= gap)
7          arr[j] = arr[j - gap];
8        arr[j] = key;
9      }
10 }
```



gap = 1 / 2 = 0: the loop ends and the array is sorted. Only **3 shifts** in total; the gap-3 pass moved the small items forward in one go.

Step 10 / 10

VARIABLES

`n = 6` `gap = 0` `i = –` `key = –` `j = –`

Merge Sort

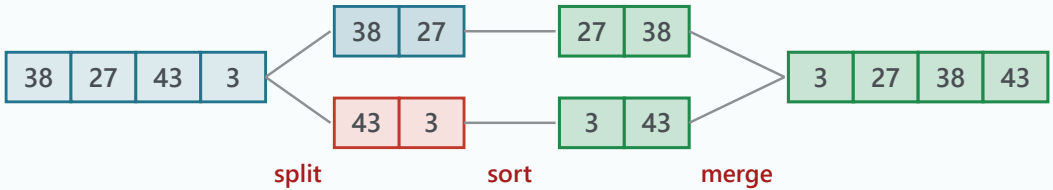
- A **divide-and-conquer** algorithm.
- Invented by **John von Neumann** in 1945.
- **O(n log n)** even in the worst case.
- **Stable**; merging needs **extra memory** (O(n)).

How does it work?

1. Divide: split the list into two sublists, and keep splitting until each has one item.

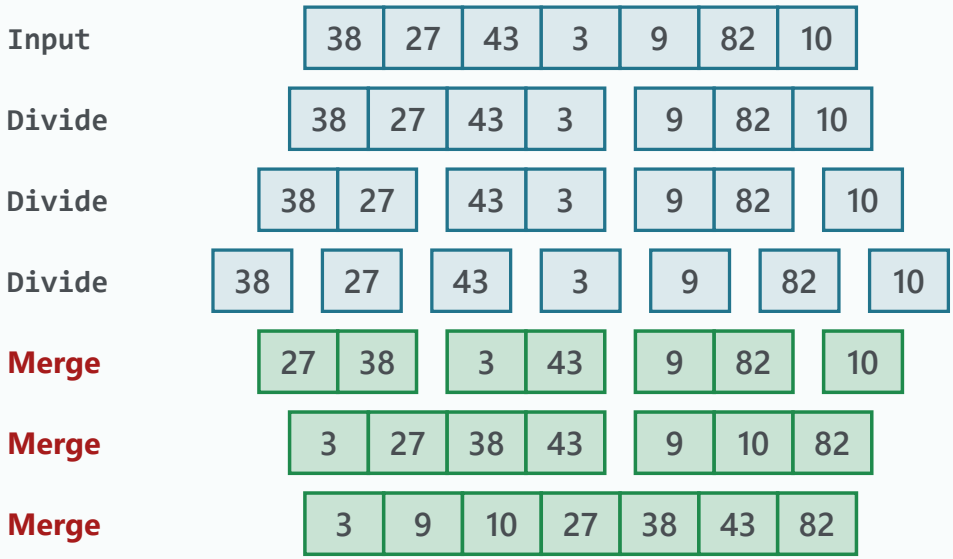
2. Conquer: sort each half the same way (recursively).

3. Combine: merge the sorted sublists until a single sorted list remains.



A one-item list is sorted by definition; this is the recursion's **base case**. The real work happens in the merge step.

Example: [38, 27, 43, 3, 9, 82, 10]



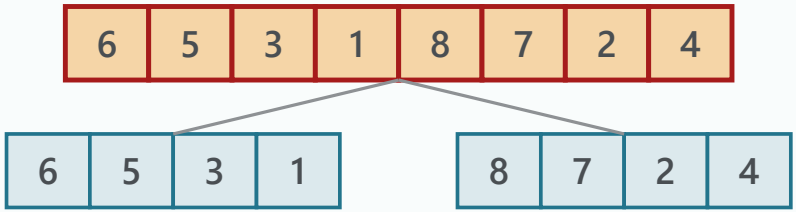
Seven items: since mid = ⌊(left + right)/2⌋, the left half gets one extra item ([38, 27, 43, 3] and [9, 82, 10]).

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



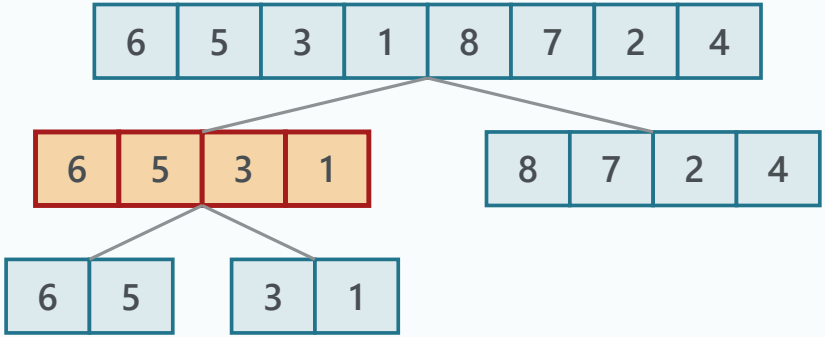
Array [6, 5, 3, 1, 8, 7, 2, 4]: first **split** down to single items, then the sorted pieces are **merged**.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



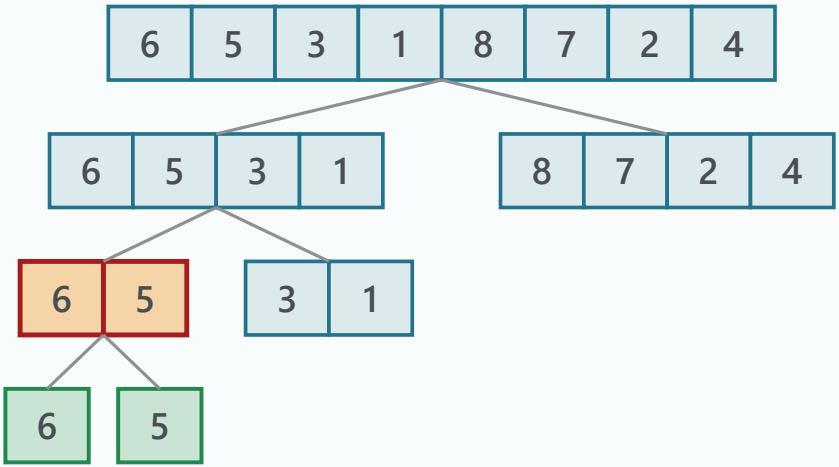
Split: [6, 5, 3, 1, 8, 7, 2, 4] → [6, 5, 3, 1] and [8, 7, 2, 4].

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



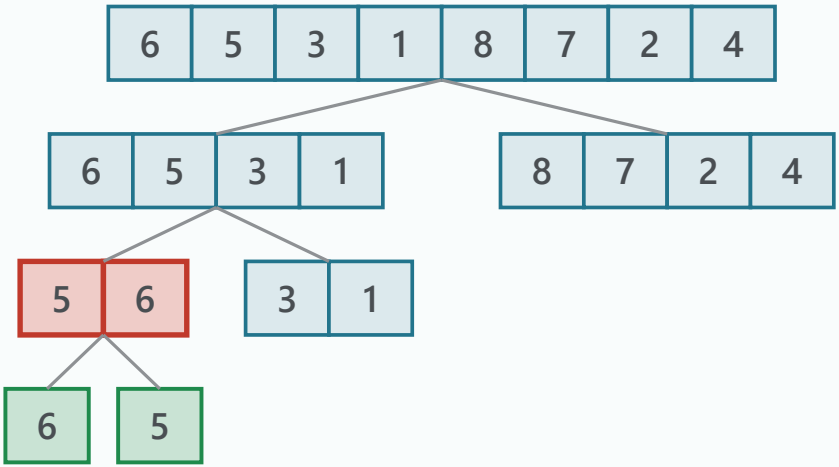
Split: [6, 5, 3, 1] → [6, 5] and [3, 1].

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



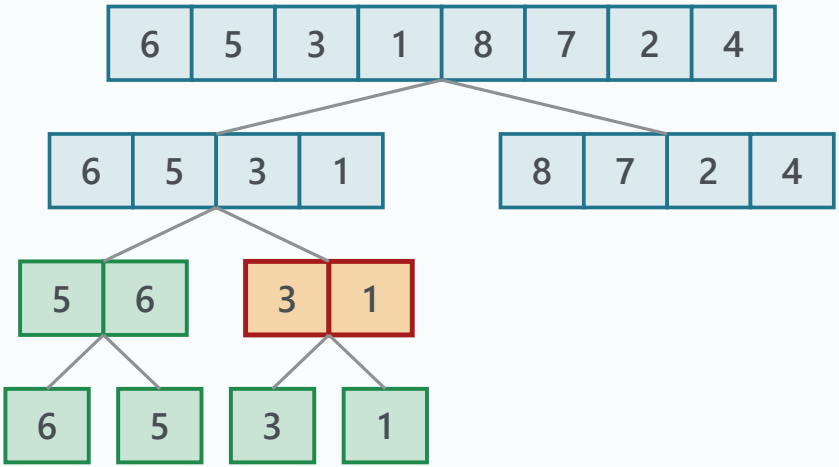
Split: [6, 5] → [6] and [5]. One-item pieces are already sorted.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



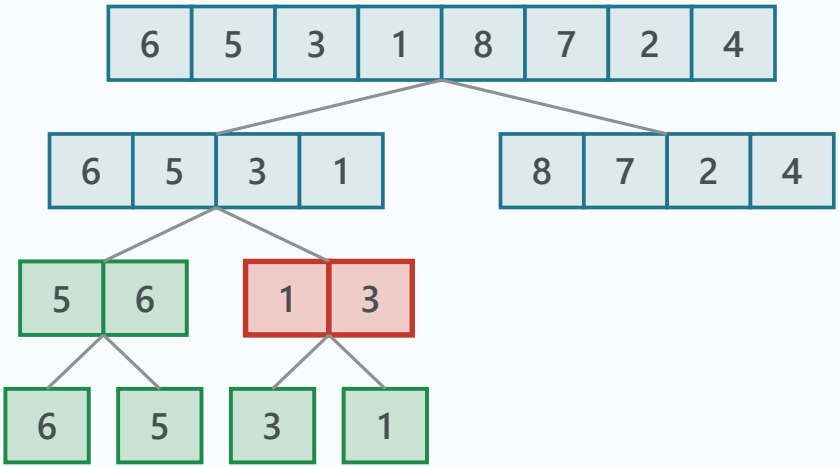
Merge: [6] + [5] → **[5, 6]**.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



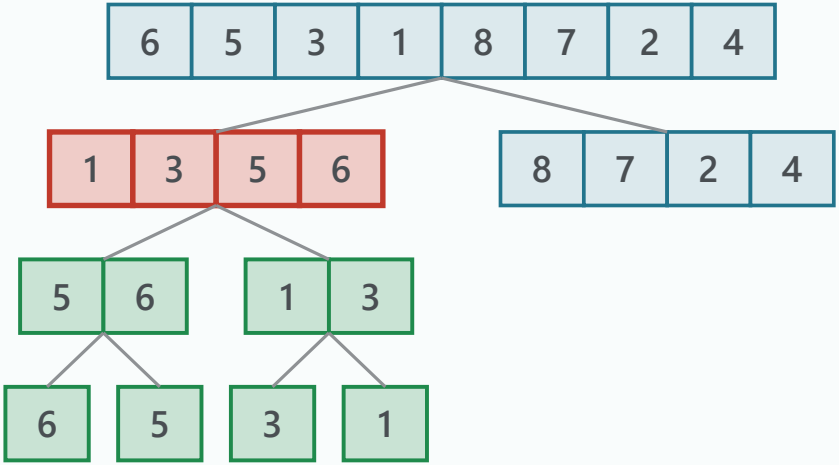
Split: [3, 1] → [3] and [1]. One-item pieces are already sorted.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



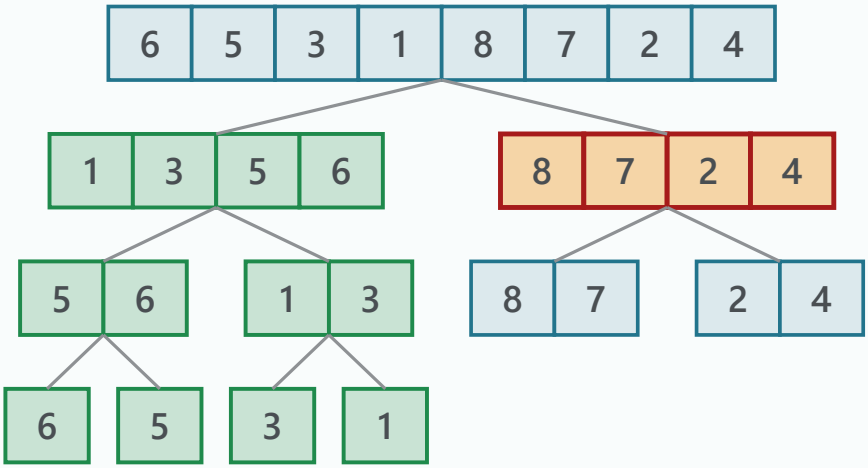
Merge: [3] + [1] → **[1, 3]**.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



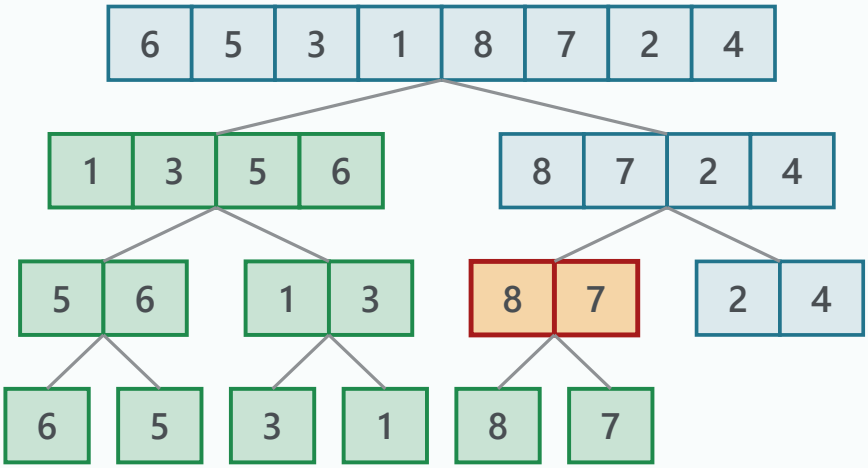
Merge: [5, 6] + [1, 3] → **[1, 3, 5, 6]**.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



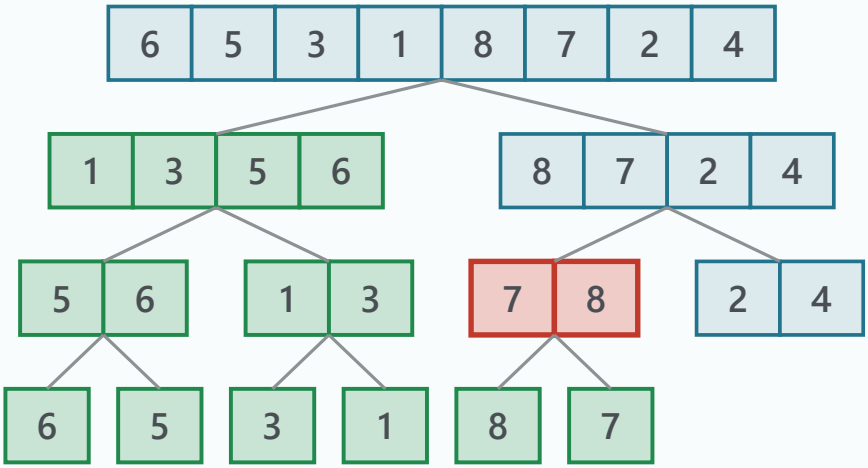
Split: [8, 7, 2, 4] → [8, 7] and [2, 4].

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



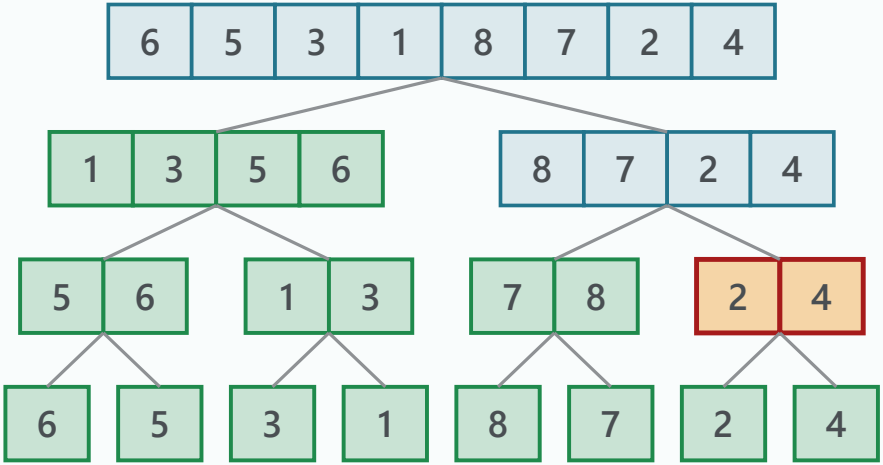
Split: [8, 7] → [8] and [7]. One-item pieces are already sorted.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



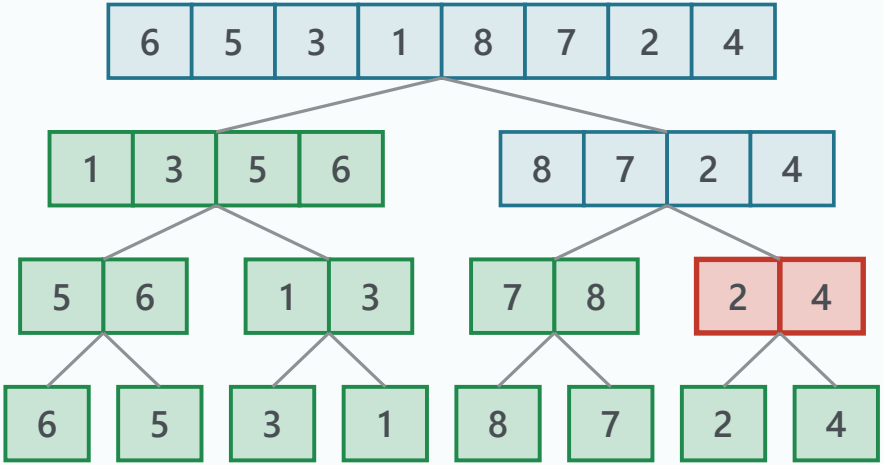
Merge: [8] + [7] → **[7, 8]**.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



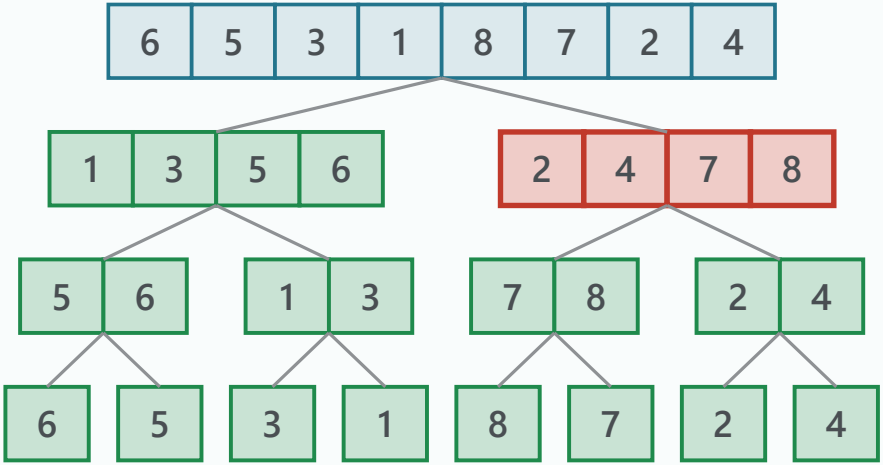
Split: [2, 4] → [2] and [4]. One-item pieces are already sorted.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



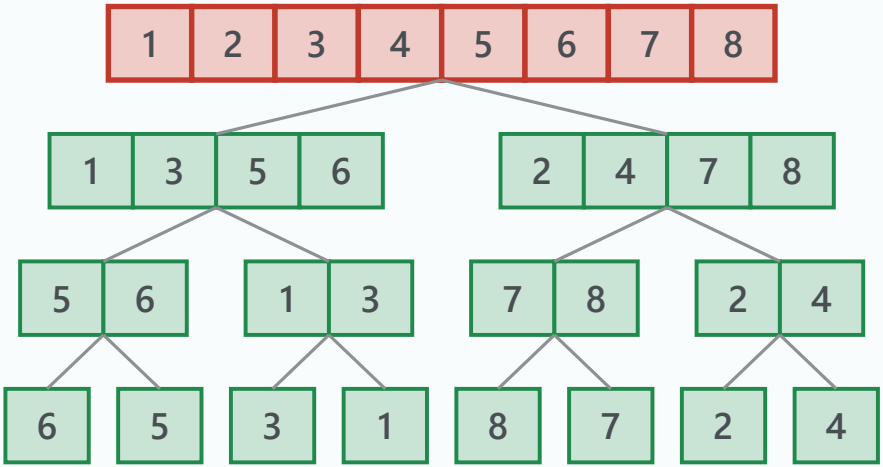
Merge: [2] + [4] → **[2, 4]**.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



Merge: [7, 8] + [2, 4] → **[2, 4, 7, 8]**.

Merge Sort Step by Step: [6, 5, 3, 1, 8, 7, 2, 4]



Merge: [1, 3, 5, 6] + [2, 4, 7, 8] → **[1, 2, 3, 4, 5, 6, 7, 8]**. The array is sorted.

Merge Sort: Java Code

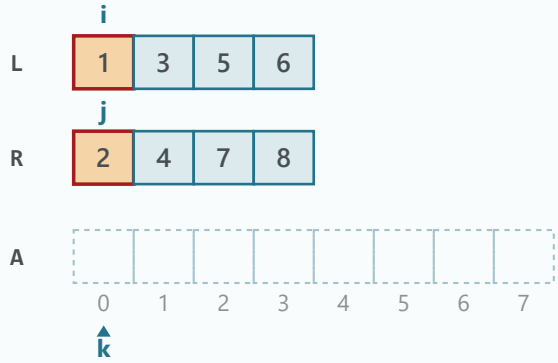
```java
void mergeSort(int[] arr, int left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
  }
}
```

```java
void merge(int[] A, int p, int q, int r) {
  int[] L = Arrays.copyOfRange(A, p, q + 1);
  int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
  int i = 0, j = 0, k = p;
  while (i < L.length && j < R.length) {
    if (L[i] <= R[j]) A[k++] = L[i++];
    else              A[k++] = R[j++];
  }
  while (i < L.length) A[k++] = L[i++];
  while (j < R.length) A[k++] = R[j++];
}
```

Writing `left + (right - left) / 2` avoids the int overflow in `(left + right) / 2`.

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
▶    int[] L = Arrays.copyOfRange(A, p, q + 1);
▶    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
▶    int i = 0, j = 0, k = p;
5    while (i < L.length && j < R.length) {
6      if (L[i] <= R[j]) A[k++] = L[i++];
7      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
10   while (j < R.length) A[k++] = R[j++];
11 }
```



Copy L = A[p..q] = [1, 3, 5, 6] and R = A[q+1..r] = [2, 4, 7, 8] (both sorted). i = j = 0, k = p = 0.

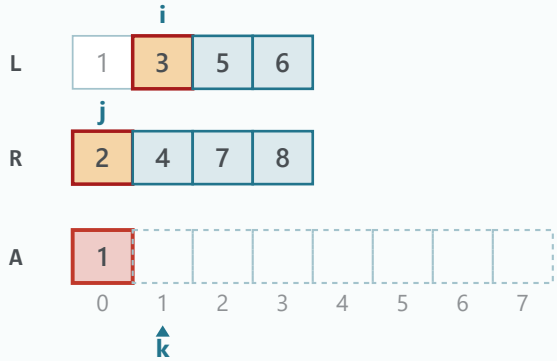
Step 1 / 9

VARIABLES

`i = 0` `j = 0` `k = 0`

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
▶    while (i < L.length && j < R.length) {
▶      if (L[i] <= R[j]) A[k++] = L[i++];
7      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
10   while (j < R.length) A[k++] = R[j++];
11 }
```



L[0] = 1 ≤ R[0] = 2 → A[0] = 1; i++, k++.

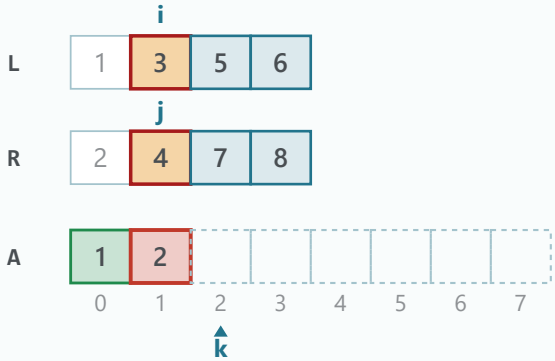
Step 2 / 9

VARIABLES

i = 1 | j = 0 | k = 1

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
▶    while (i < L.length && j < R.length) {
6      if (L[i] <= R[j]) A[k++] = L[i++];
▶      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
10   while (j < R.length) A[k++] = R[j++];
11 }
```



L[1] = 3 > R[0] = 2 → A[1] = 2; j++, k++.

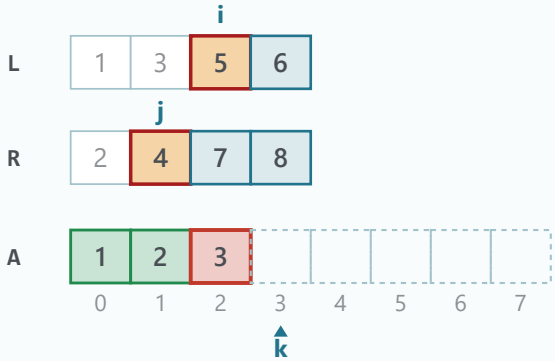
Step 3 / 9

VARIABLES

i = 1 j = 1 k = 2

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
▶    while (i < L.length && j < R.length) {
▶      if (L[i] <= R[j]) A[k++] = L[i++];
7      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
10   while (j < R.length) A[k++] = R[j++];
11 }
```



L[1] = 3 ≤ R[1] = 4 → A[2] = 3; i++, k++.

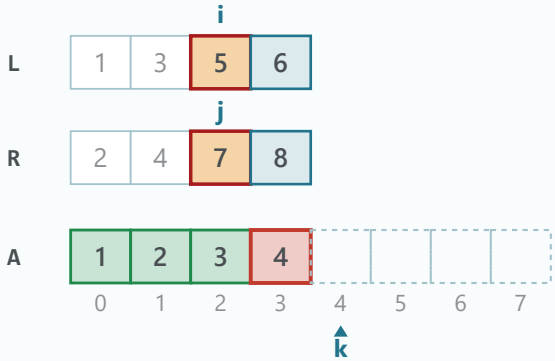
Step 4 / 9

VARIABLES

i = 2 | j = 1 | k = 3

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
▶    while (i < L.length && j < R.length) {
6      if (L[i] <= R[j]) A[k++] = L[i++];
▶      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
10   while (j < R.length) A[k++] = R[j++];
11 }
```



L[2] = 5 > R[1] = 4 → A[3] = 4; j++, k++.

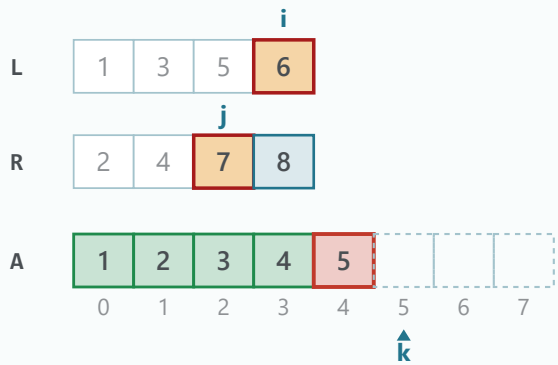
Step 5 / 9

VARIABLES

i = 2 j = 2 k = 4

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
▶    while (i < L.length && j < R.length) {
▶      if (L[i] <= R[j]) A[k++] = L[i++];
7      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
10   while (j < R.length) A[k++] = R[j++];
11 }
```



L[2] = 5 ≤ R[2] = 7 → A[4] = 5; i++, k++.

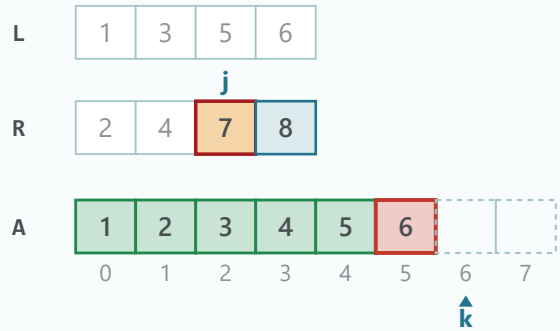
Step 6 / 9

VARIABLES

i = 3 j = 2 k = 5

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
▶    while (i < L.length && j < R.length) {
▶      if (L[i] <= R[j]) A[k++] = L[i++];
7      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
10   while (j < R.length) A[k++] = R[j++];
11 }
```



L[3] = 6 ≤ R[2] = 7 → A[5] = 6; i++, k++.

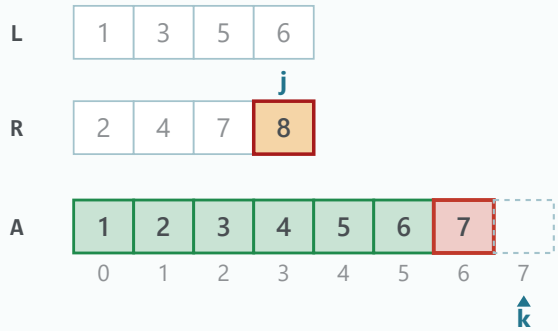
Step 7 / 9

VARIABLES

i = 4 j = 2 k = 6

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
5    while (i < L.length && j < R.length) {
6      if (L[i] <= R[j]) A[k++] = L[i++];
7      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
▶    while (j < R.length) A[k++] = R[j++];
11 }
```



L is exhausted (i = 4); the remaining R items are copied in order: A[6] = 7.

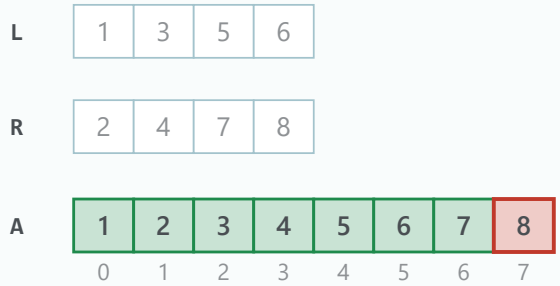
Step 8 / 9

VARIABLES

i = 4 j = 3 k = 7

merge(): Merging Two Sorted Arrays

```
1  void merge(int[] A, int p, int q, int r) {
2    int[] L = Arrays.copyOfRange(A, p, q + 1);
3    int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
4    int i = 0, j = 0, k = p;
5    while (i < L.length && j < R.length) {
6      if (L[i] <= R[j]) A[k++] = L[i++];
7      else              A[k++] = R[j++];
8    }
9    while (i < L.length) A[k++] = L[i++];
▶    while (j < R.length) A[k++] = R[j++];
11 }
```



A[7] = 8. Merge done: [1, 2, 3, 4, 5, 6, 7, 8] (8 items, at most 7 comparisons).

Step 9 / 9

VARIABLES

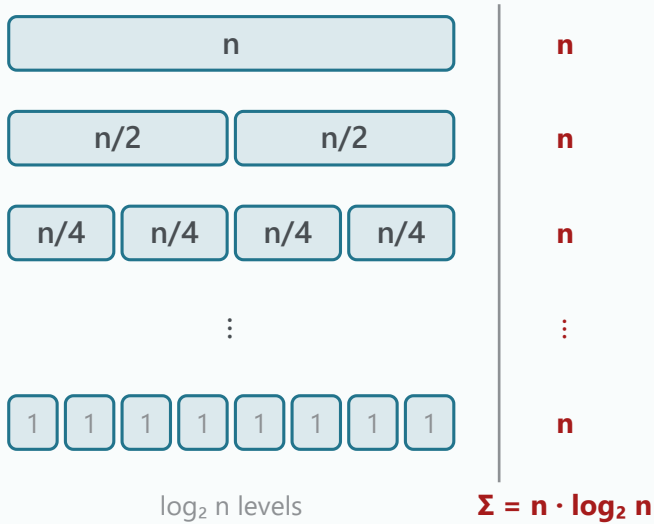
`i = 4` `j = 4` `k = 8`

Merge Sort: Analysis

Recurrence:

$$T(n) = 2\,T(n/2) + \Theta(n)$$

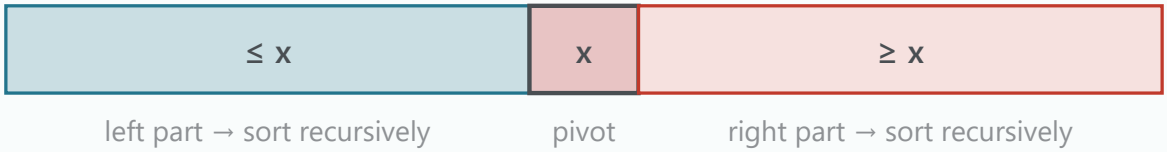
- The total merge work on each level is **$\Theta(n)$**.
- The number of levels is **$\log_2 n$** (halving n down to 1).
- Total: Θ(n log n) — **best = average = worst**.
- Extra memory Θ(n) (auxiliary arrays) + a Θ(log n) recursion stack.
- `L[i] <= R[j]` : ties are taken from the left → **stable**.



By the Master Theorem: a = 2, b = 2, f(n) = Θ(n) = Θ($n^{\log_2 2}$) → case 2 → T(n) = Θ(n log n).

Quicksort

- C. A. R. Hoare (1959). Also **divide and conquer**, but most of the work happens in the **divide** step.
- Choose a **pivot** item.
- Split the array into two parts around the pivot (**partitioning**): ≤ x on the left, ≥ x on the right.
- Sort the two parts recursively. **No** merge step is needed.



After partitioning the pivot is **in its final place**: it sits at exactly that index in the sorted array.

The Partitioning Procedure

Lomuto: pivot = last item

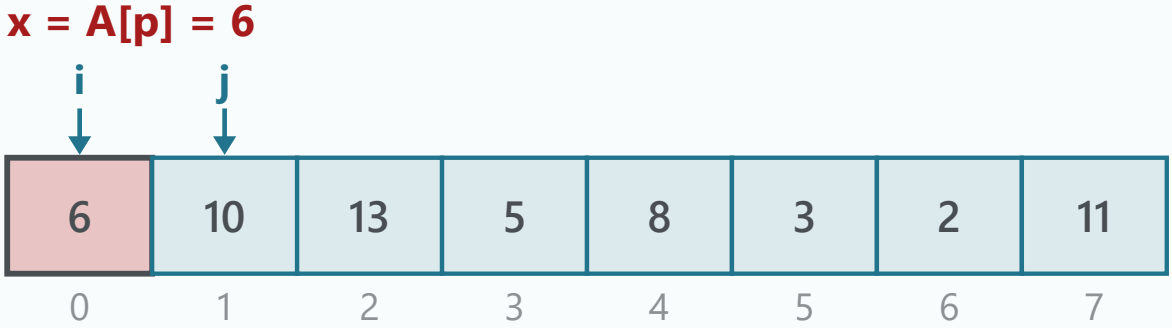
```
partition(A, p, q):
  pivot = A[q]        // choose pivot
  i = p - 1           // end of ≤ zone
  for j = p to q - 1:
    if A[j] <= pivot:
      i = i + 1
      swap(A[i], A[j])
  swap(A[i + 1], A[q])  // place the pivot
  return i + 1          // pivot index
```

Same idea: pivot = first item

```
PARTITION(A, p, q):
  x = A[p]            // pivot
  i = p
  for j = p + 1 to q:
    if A[j] <= x:
      i = i + 1
      swap(A[i], A[j])
  swap(A[p], A[i])
  return i
```

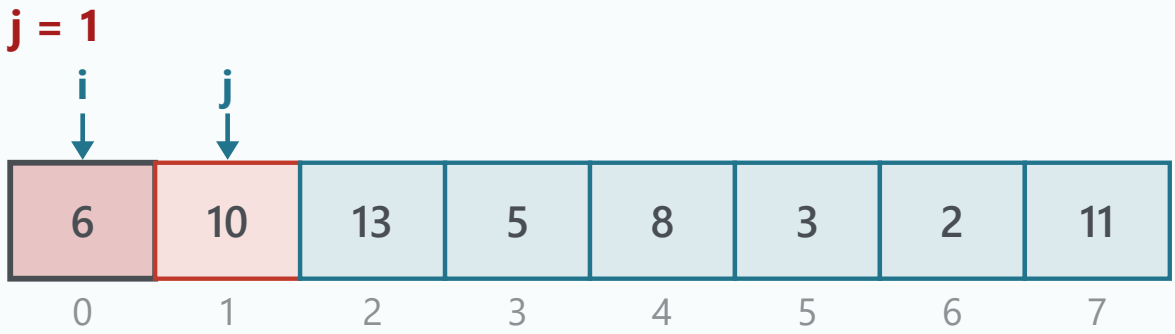
Invariant: A[p..i] ≤ pivot, A[i+1..j−1] > pivot, A[j..] not yet examined. One pass: **Θ(n)** time, O(1) extra memory.

Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]



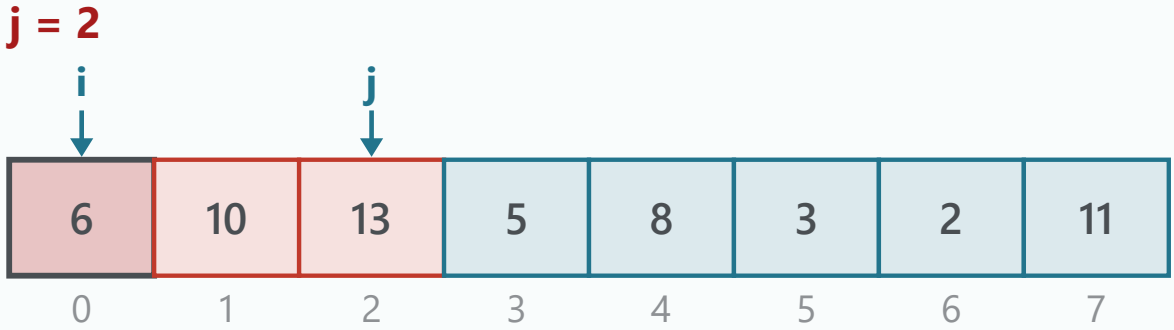
Pivot x = A[p] = 6 (yellow). i = p; j starts at p + 1. Blue: ≤ x, red: > x, white: not yet seen.

Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]



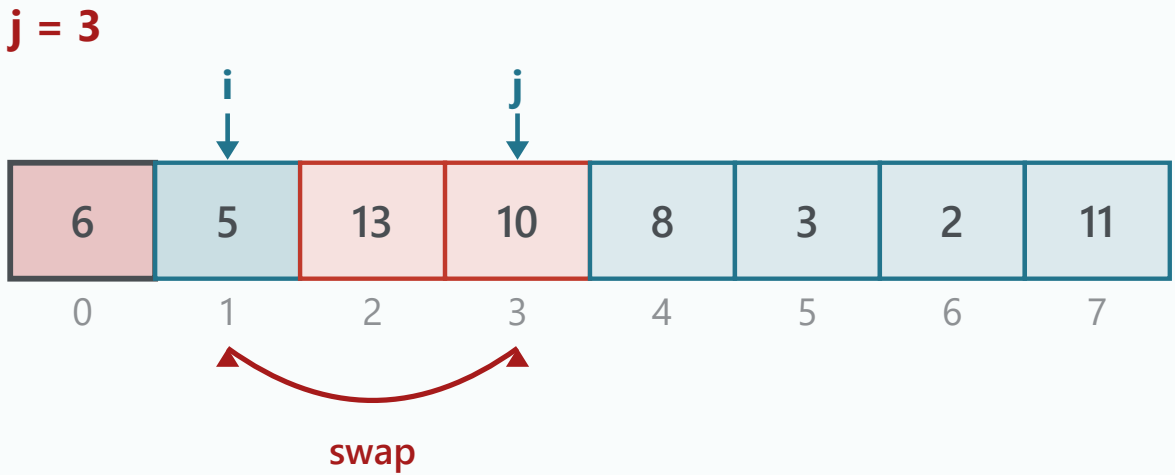
A[j] = 10 > 6 → only j advances; the red zone grows.

Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]



A[j] = 13 > 6 → only j advances; the red zone grows.

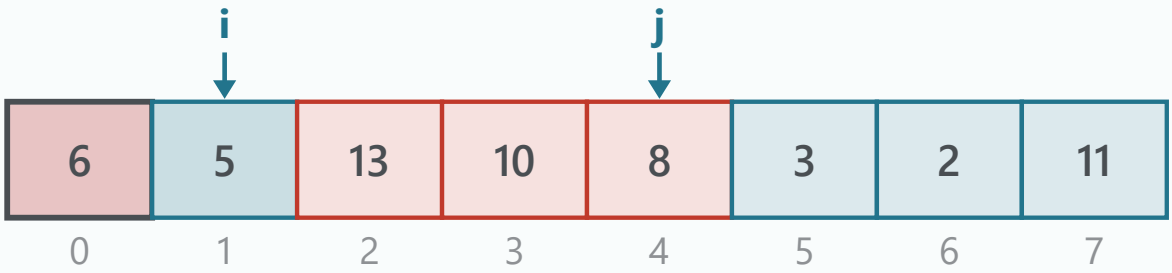
Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]



A[j] = 5 ≤ 6 → i = 1; A[i] ↔ A[j]: [6, 5, 13, 10, 8, 3, 2, 11]. The blue zone grows by one.

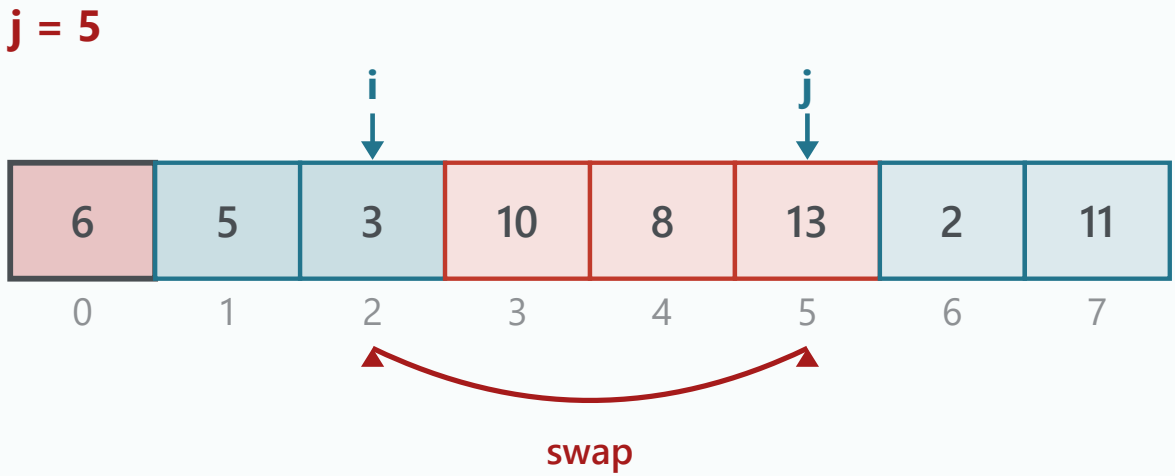
Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]

j = 4



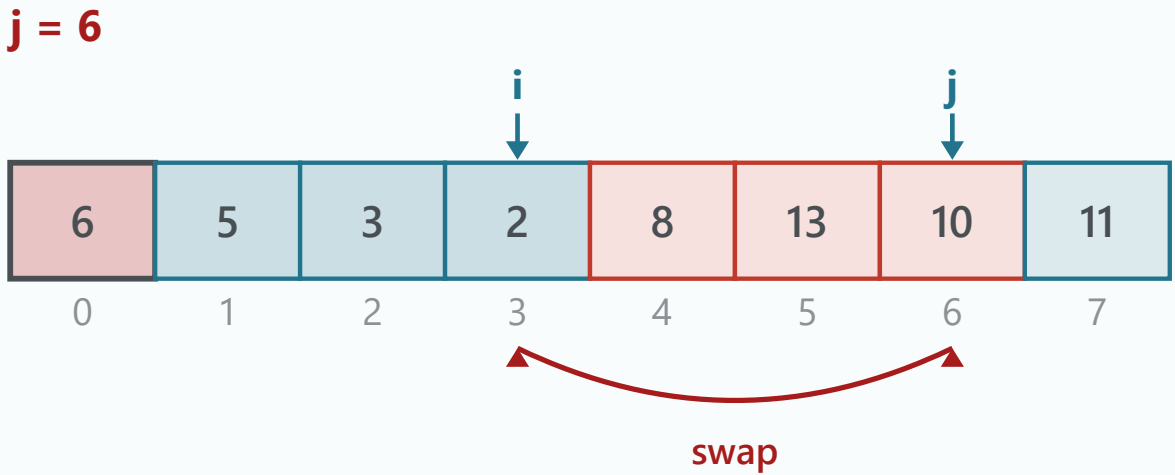
A[j] = 8 > 6 → only j advances; the red zone grows.

Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]



A[j] = 3 ≤ 6 → i = 2; A[i] ↔ A[j]: [6, 5, 3, 10, 8, 13, 2, 11]. The blue zone grows by one.

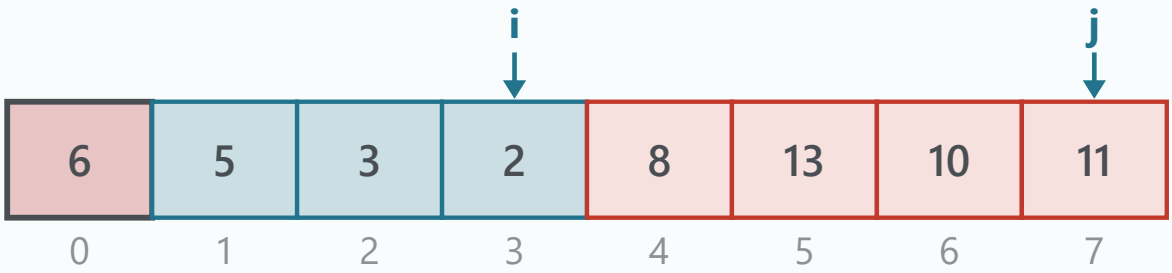
Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]



A[j] = 2 ≤ 6 → i = 3; A[i] ↔ A[j]: [6, 5, 3, 2, 8, 13, 10, 11]. The blue zone grows by one.

Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]

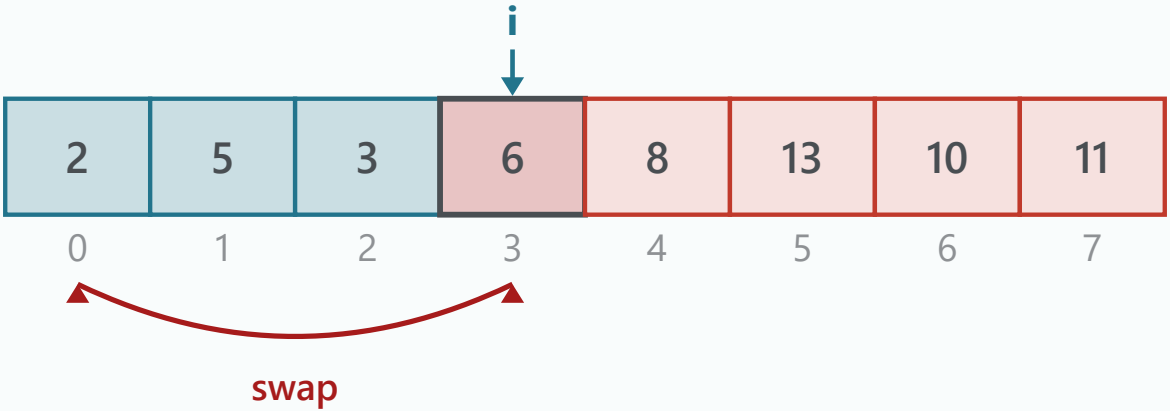
j = 7



A[j] = 11 > 6 → only j advances; the red zone grows.

Partitioning: [6, 10, 13, 5, 8, 3, 2, 11]

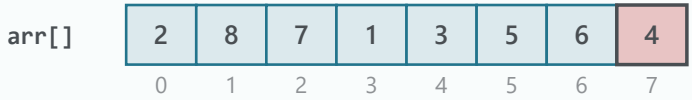
Finally: A[p] ↔ A[i]



The loop is done. The pivot is swapped with the last item of the blue zone: [2, 5, 3, 6, 8, 13, 10, 11]. Pivot 6 is now **in its final place** (index 3); everything to its left is ≤ 6, to its right ≥ 6.

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
▶    int pivot = arr[hi];
▶    int i = lo - 1;
4    for (int j = lo; j < hi; j++) {
5      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



pivot = arr[hi] = 4; i = lo − 1 = −1 (the ≤ pivot zone is empty).

Step 1 / 13

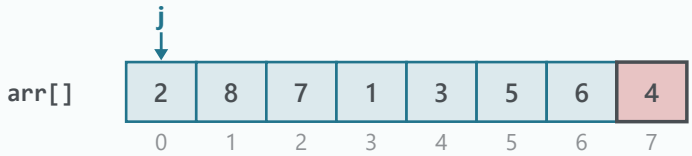
VARIABLES

`lo = 0` `hi = 7` `pivot = 4` `i = -1` `j = –`

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
▶    for (int j = lo; j < hi; j++) {
▶      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



j = 0: arr[0] = 2 ≤ 4? **Yes**.

Step 2 / 13

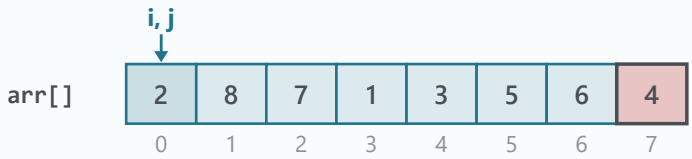
VARIABLES

`lo = 0` `hi = 7` `pivot = 4` `i = -1` `j = 0`

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
4    for (int j = lo; j < hi; j++) {
5      if (arr[j] <= pivot) {
▶        i++;
▶        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



i = 0; swap(arr, 0, 0) → [2, 8, 7, 1, 3, 5, 6, 4] (with itself).

Step 3 / 13

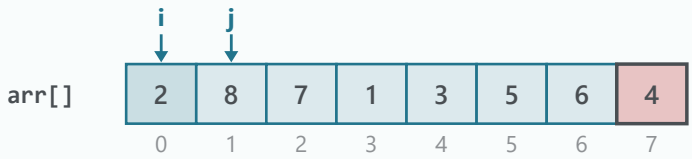
VARIABLES

lo = 0 | hi = 7 | pivot = 4 | i = 0 | j = 0

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
▶    for (int j = lo; j < hi; j++) {
▶      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



j = 1: arr[1] = 8 ≤ 4? No → j advances.

Step 4 / 13

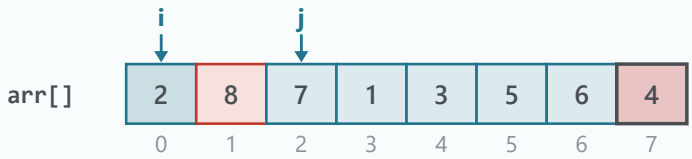
VARIABLES

lo = 0 | hi = 7 | pivot = 4 | i = 0 | j = 1

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
▶    for (int j = lo; j < hi; j++) {
▶      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



j = 2: arr[2] = 7 ≤ 4? No → j advances.

Step 5 / 13

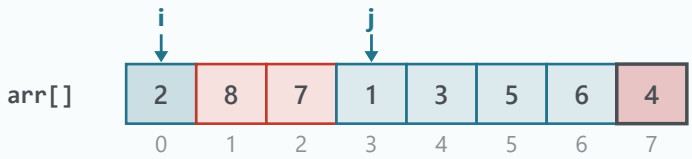
VARIABLES

lo = 0 | hi = 7 | pivot = 4 | i = 0 | j = 2

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
▶    for (int j = lo; j < hi; j++) {
▶      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



j = 3: arr[3] = 1 ≤ 4? **Yes**.

Step 6 / 13

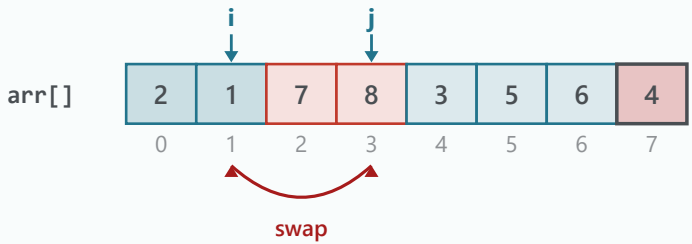
VARIABLES

`lo = 0` `hi = 7` `pivot = 4` `i = 0` `j = 3`

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
4    for (int j = lo; j < hi; j++) {
5      if (arr[j] <= pivot) {
▶        i++;
▶        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



i = 1; swap(arr, 1, 3) → [2, 1, 7, 8, 3, 5, 6, 4].

Step 7 / 13

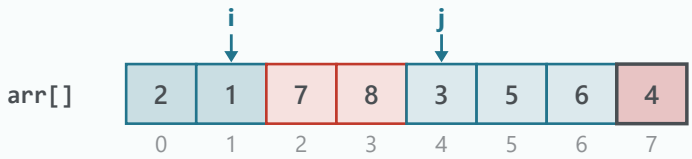
VARIABLES

`lo = 0` `hi = 7` `pivot = 4` `i = 1` `j = 3`

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
▶    for (int j = lo; j < hi; j++) {
▶      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



j = 4: arr[4] = 3 ≤ 4? **Yes**.

Step 8 / 13

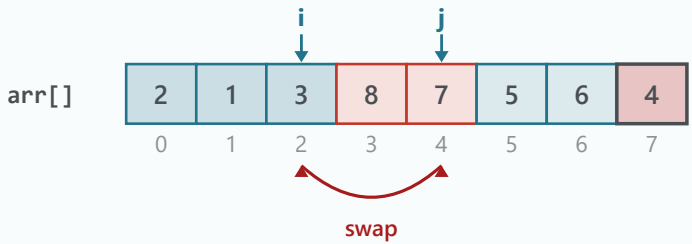
VARIABLES

`lo = 0` `hi = 7` `pivot = 4` `i = 1` `j = 4`

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
4    for (int j = lo; j < hi; j++) {
5      if (arr[j] <= pivot) {
▶        i++;
▶        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



i = 2; swap(arr, 2, 4) → [2, 1, 3, 8, 7, 5, 6, 4].

Step 9 / 13

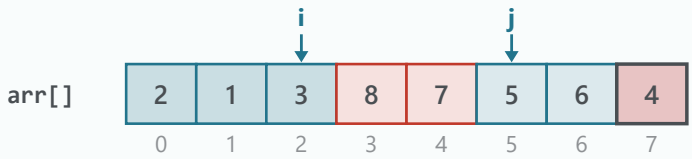
VARIABLES

`lo = 0` `hi = 7` `pivot = 4` `i = 2` `j = 4`

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
▶    for (int j = lo; j < hi; j++) {
▶      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



j = 5: arr[5] = 5 ≤ 4? No → j advances.

Step 10 / 13

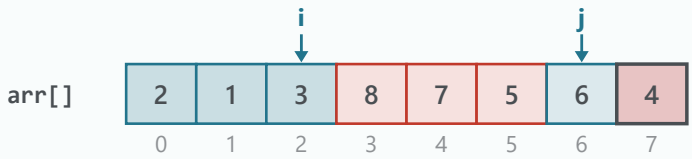
VARIABLES

lo = 0 | hi = 7 | pivot = 4 | i = 2 | j = 5

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
▶    for (int j = lo; j < hi; j++) {
▶      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



j = 6: arr[6] = 6 ≤ 4? No → j advances.

Step 11 / 13

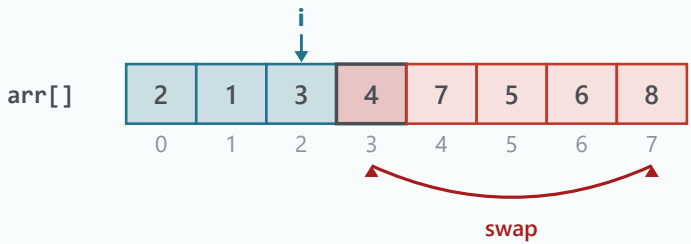
VARIABLES

lo = 0 | hi = 7 | pivot = 4 | i = 2 | j = 6

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
4    for (int j = lo; j < hi; j++) {
5      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
▶    swap(arr, i + 1, hi);
11   return i + 1;
12 }
```



The loop is done. swap(arr, i+1 = 3, hi = 7) → [2, 1, 3, 4, 7, 5, 6, 8].

Step 12 / 13

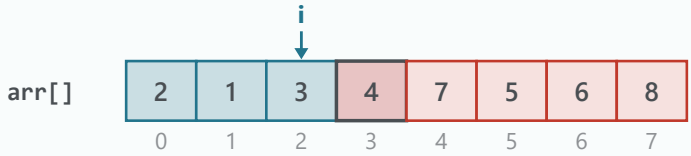
VARIABLES

lo = 0 | hi = 7 | pivot = 4 | i = 2 | j = 7

OUTPUT

partition(): Code Trace, arr = [2, 8, 7, 1, 3, 5, 6, 4]

```
1  int partition(int[] arr, int lo, int hi) {
2    int pivot = arr[hi];
3    int i = lo - 1;
4    for (int j = lo; j < hi; j++) {
5      if (arr[j] <= pivot) {
6        i++;
7        swap(arr, i, j);
8      }
9    }
10   swap(arr, i + 1, hi);
     return i + 1;
12 }
```



`return 3` : pivot 4 is in its final place; left [0..2] ≤ 4, right [4..7] > 4.

Step 13 / 13

VARIABLES

`lo = 0` `hi = 7` `pivot = 4` `i = 2` `j = 7`

OUTPUT

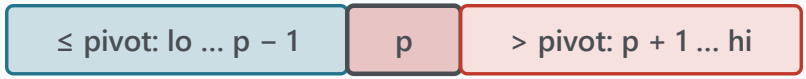
```
partition → 3
```

Quicksort: Recursive Structure

```
QUICKSORT(A, p, r)
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q – 1)
    QUICKSORT(A, q + 1, r)

Initial call: QUICKSORT(A, 1, n)
```

```java
void quickSort(int[] arr, int lo, int hi) {
  if (lo < hi) {
    int p = partition(arr, lo, hi);
    quickSort(arr, lo, p - 1);
    quickSort(arr, p + 1, hi);
  }
}
```

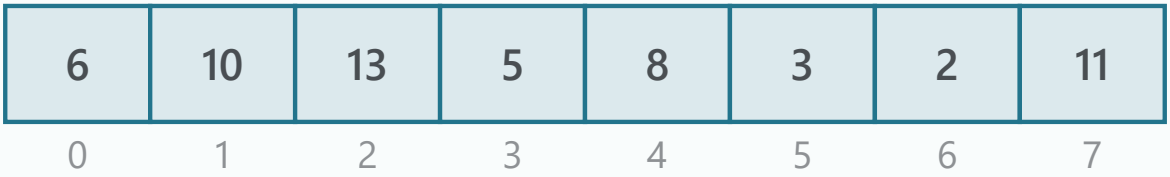


quickSort(arr, lo, p – 1) in place quickSort(arr, p + 1, hi)

- The pivot (index q) is included in **neither** recursive call: it is already in place.
- Base case: a subarray of 0 or 1 items (p ≥ r) is already sorted.
- The Java version is 0-based: the initial call is `quickSort(arr, 0, n – 1)`.

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

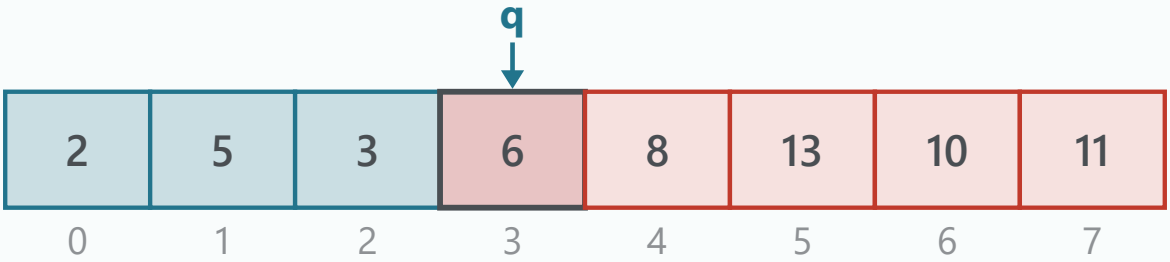
QUICKSORT(A, 0, 7)



Array [6, 10, 13, 5, 8, 3, 2, 11]. Each call partitions its subarray with pivot = first item, then sorts both sides recursively.

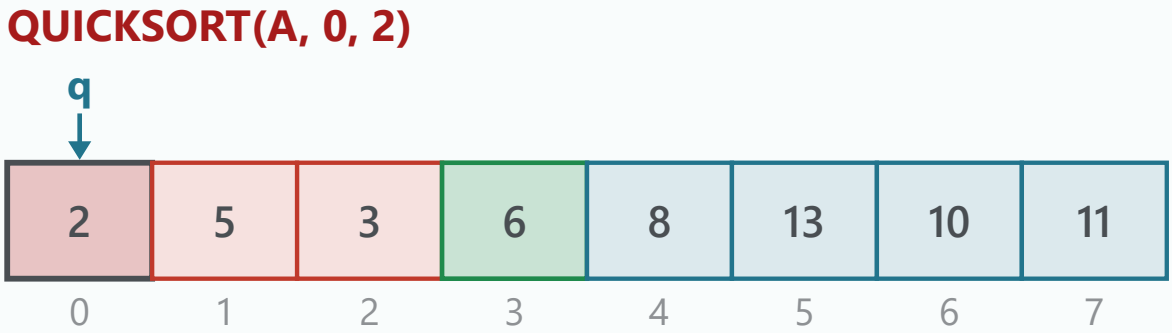
Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

QUICKSORT(A, 0, 7)



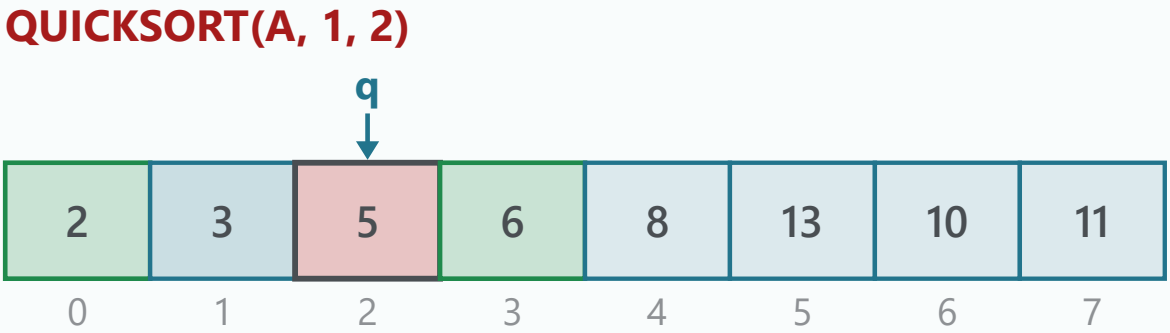
Depth 0: [6, 10, 13, 5, 8, 3, 2, 11] is partitioned around 6 → the pivot lands at index 3; then [0..2] and [4..7] are sorted.

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]



Depth 1: [2, 5, 3] is partitioned around 2 → the pivot lands at index 0; then [0..-1] and [1..2] are sorted.

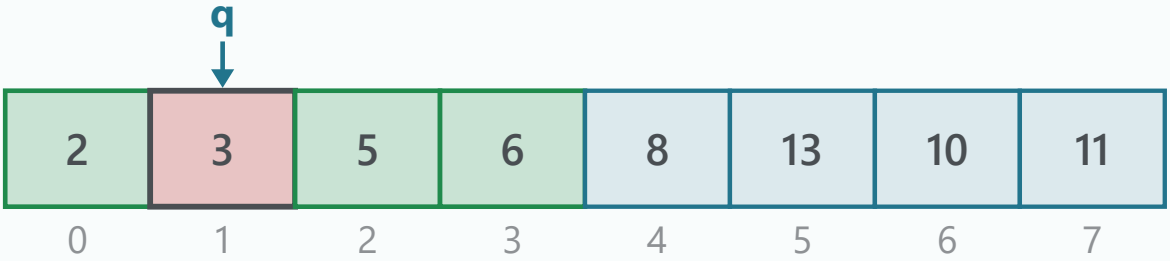
Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]



Depth 2: [5, 3] is partitioned around 5 → the pivot lands at index 2; then [1..1] and [3..2] are sorted.

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

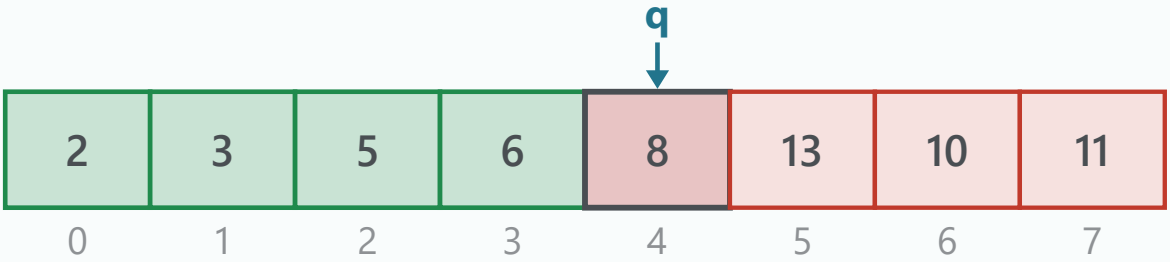
QUICKSORT(A, 1, 1)



Depth 3: [1..1] has one item (3) → already sorted (p < r fails).

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

QUICKSORT(A, 4, 7)



Depth 1: [8, 13, 10, 11] is partitioned around 8 → the pivot lands at index 4; then [4..3] and [5..7] are sorted.

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

QUICKSORT(A, 5, 7)



Depth 2: [13, 10, 11] is partitioned around 13 → the pivot lands at index 7; then [5..6] and [8..7] are sorted.

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

QUICKSORT(A, 5, 6)



Depth 3: [11, 10] is partitioned around 11 → the pivot lands at index 6; then [5..5] and [7..6] are sorted.

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

QUICKSORT(A, 5, 5)



Depth 4: [5..5] has one item (10) → already sorted (p < r fails).

Quicksort Step by Step: [6, 10, 13, 5, 8, 3, 2, 11]

Result



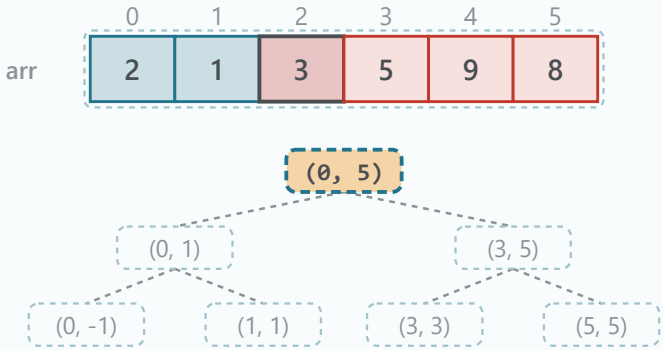
All calls are done; there is no merge step, and the array was sorted in place: [2, 3, 5, 6, 8, 10, 11, 13].

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
▶   if (lo < hi) {
▶     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
5     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

`quickSort(0, 5)`



`quickSort(0, 5)` : lo < hi. `partition` splits around pivot = 3 (last item): [2, 1] on the left, [5, 9, 8] on the right; the pivot lands at index **p = 2**. Then line 4: the left part.

Step 1 / 8

VARIABLES

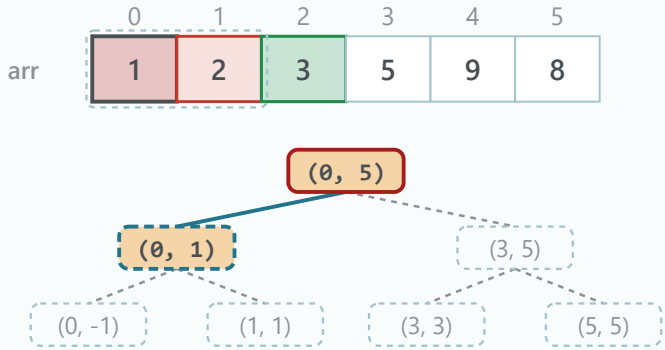
`lo = 0` `hi = 5` `p = 2`

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
▶   if (lo < hi) {
▶     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
5     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

- `quickSort(0, 1)`
- `quickSort(0, 5)`



`quickSort(0, 1)` : lo < hi. `partition` splits around pivot = 1 (last item): [] on the left, [2] on the right; the pivot lands at index **p = 0**.

Step 2 / 8

VARIABLES

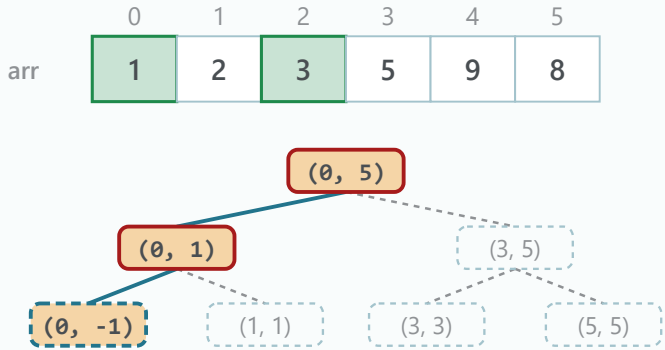
`lo = 0` `hi = 1` `p = 0`

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
▶   if (lo < hi) {
3     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
5     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

- **quickSort(0, -1)**
- quickSort(0, 1)
- quickSort(0, 5)



`quickSort(0, -1)` : lo > hi, empty part → returns at once (base case).

Step 3 / 8

VARIABLES

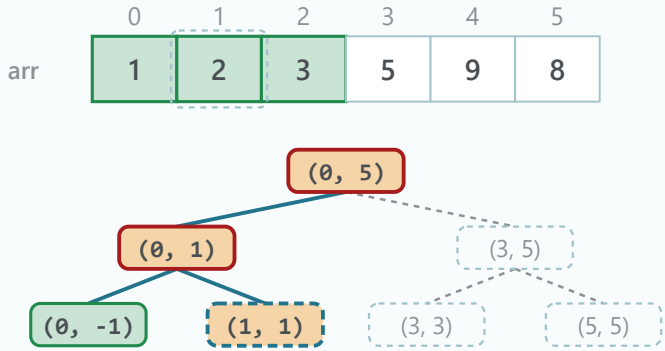
`lo = 0` `hi = -1` `p = –`

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
▶   if (lo < hi) {
3     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
5     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

- **quickSort(1, 1)**
- quickSort(0, 1)
- quickSort(0, 5)



`quickSort(1, 1)` : the one-item part [2] is already sorted → returns. Call (0, 1) is done; line 5: (3, 5).

Step 4 / 8

VARIABLES

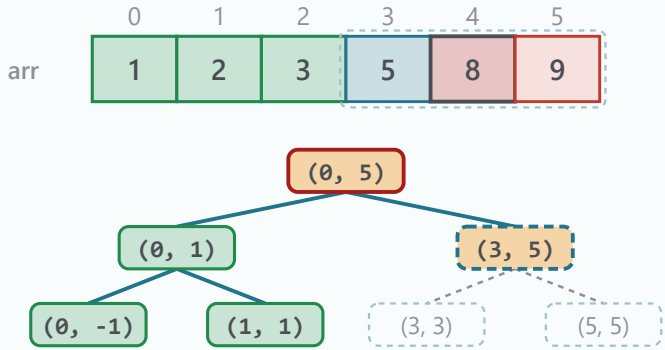
`lo = 1` `hi = 1` `p = –`

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
▶   if (lo < hi) {
▶     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
5     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

- **quickSort(3, 5)**
- quickSort(0, 5)



`quickSort(3, 5)` : lo < hi. `partition` splits around pivot = 8 (last item): [5] on the left, [9] on the right; the pivot lands at index **p = 4**.

Step 5 / 8

VARIABLES

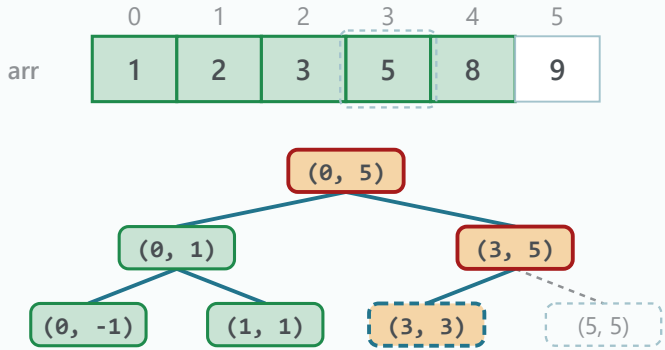
`lo = 3` `hi = 5` `p = 4`

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
▶   if (lo < hi) {
3     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
5     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

- **quickSort(3, 3)**
- quickSort(3, 5)
- quickSort(0, 5)



`quickSort(3, 3)` : the one-item part [5] is already sorted → returns.

Step 6 / 8

VARIABLES

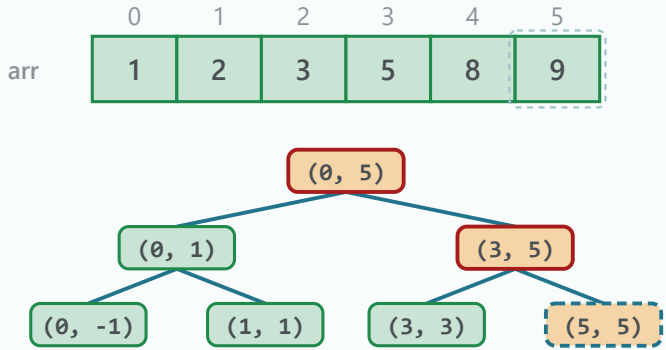
`lo = 3` `hi = 3` `p = –`

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
▶   if (lo < hi) {
3     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
5     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

- **quickSort(5, 5)**
- quickSort(3, 5)
- quickSort(0, 5)



`quickSort(5, 5)` : the one-item part [9] is already sorted → returns.

Step 7 / 8

VARIABLES

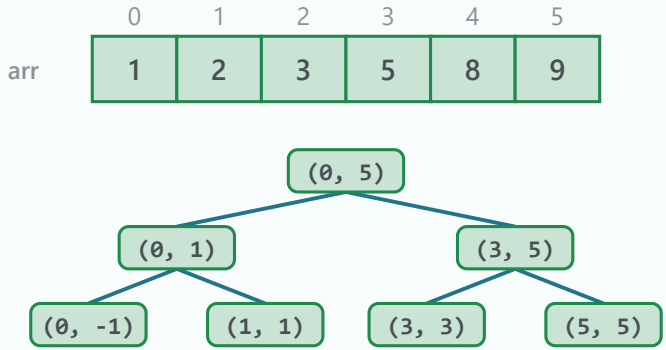
`lo = 5` `hi = 5` `p = –`

quickSort: Tracing the Calls, arr = [5, 2, 8, 1, 9, 3]

```
1 void quickSort(int[] arr, int lo, int hi) {
2   if (lo < hi) {
3     int p = partition(arr, lo, hi);
4     quickSort(arr, lo, p - 1);
▶     quickSort(arr, p + 1, hi);
6   }
7 }
```

CALL STACK

`quickSort(0, 5)`



All calls have returned: arr = [1, 2, 3, 5, 8, 9]. 3 `partition` calls and 4 base-case calls; no pivot entered a subcall, each landed in its own place.

Step 8 / 8

VARIABLES

`lo = 0` `hi = 5` `p = 2`

Quicksort: Analysis and Pivot Choice

Case	When?	Recurrence	Time
Best	pivot is always the median	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \log n)$
Average	random input / random pivot	expected $\approx 1.39\, n \log_2 n$ comparisons	$\Theta(n \log n)$
Worst	pivot always the min/max (e.g. a *sorted* array + last-item pivot)	$T(n) = T(n-1) + \Theta(n)$	$\Theta(n^2)$

Random pivot

No fixed input can force the bad case every time; $\Theta(n^2)$ happens only with tiny probability, and the expected time is $\Theta(n \log n)$.

Median of three

The pivot is the median of the first, middle and last items; this avoids the bad case on sorted and reverse-sorted input, though crafted inputs can still cause $\Theta(n^2)$.

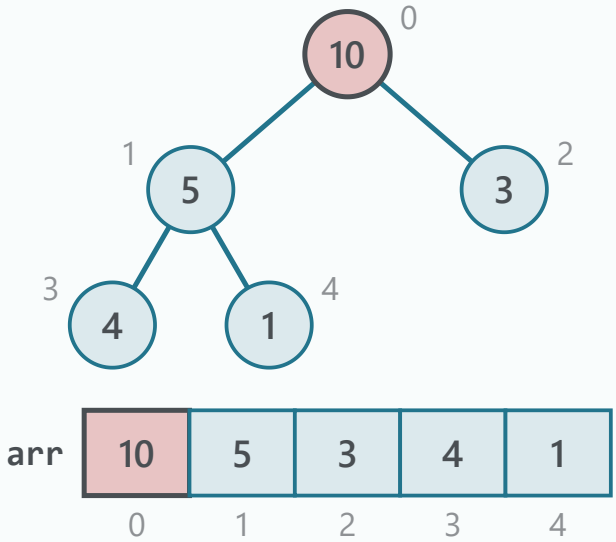
Small parts

Switching to insertion sort when $n \lesssim 16$ and recursing into the smaller side first (stack $O(\log n)$) speeds things up in practice.

In place ($O(\log n)$ stack on average), **unstable**. Being cache-friendly, it is usually the fastest comparison sort in practice (Java `Arrays.sort(int[])` : dual-pivot quicksort).

Heapsort

- Sorts using the **heap** data structure; **O(n log n)** on average and in the worst case.
- **Max-heap**: every node is greater than or equal to its children → the largest item is at the **root**.
- The tree is not a separate structure; **it is the array itself**: the children of arr[i] are `2i + 1` and `2i + 2`, its parent `(i − 1) / 2`.

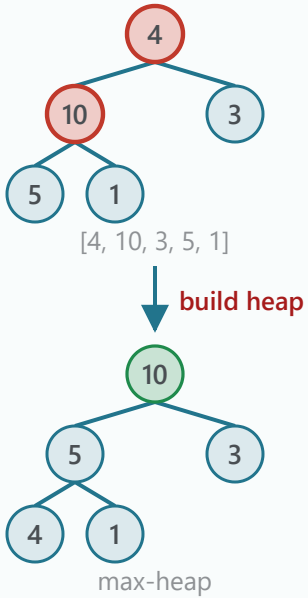


How it works

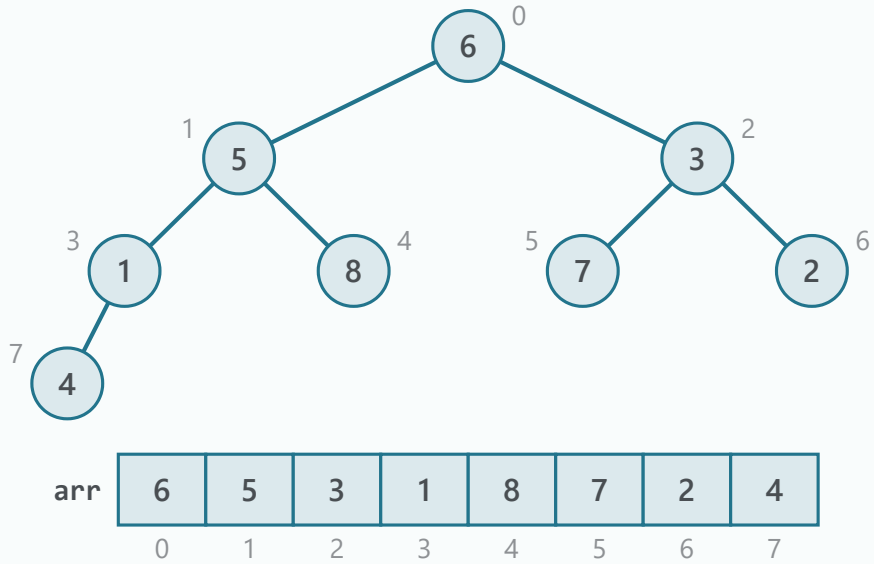
1. Turn the array into a max-heap.
2. Move the largest item (the root) to the end of the array.
3. Restore the max-heap on the rest (sift the root down).
4. Repeat until all items are sorted.

Example: [4, 10, 3, 5, 1]

Step	Array (heap | sorted)	Explanation
Array	`[4, 10, 3, 5, 1]`	
Build max-heap	`[10, 5, 3, 4, 1]`	root = largest
Root ↔ arr[4]	`[1, 5, 3, 4] | [10]`	10 to the sorted part
Restore max-heap	`[5, 4, 3, 1] | [10]`	root sifted down
Root ↔ arr[3]	`[1, 4, 3] | [5, 10]`	5 to the sorted part
Restore max-heap	`[4, 1, 3] | [5, 10]`	root sifted down
Root ↔ arr[2]	`[3, 1] | [4, 5, 10]`	4 to the sorted part
Restore max-heap	`[3, 1] | [4, 5, 10]`	root sifted down
Root ↔ arr[1]	`[1] | [3, 4, 5, 10]`	3 to the sorted part
Result	`[1, 3, 4, 5, 10]`	

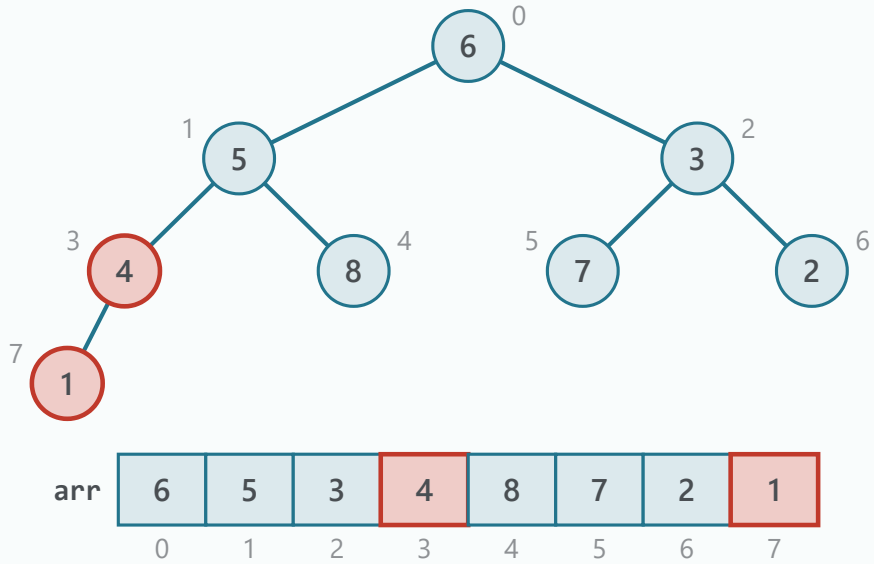


Phase 1: Building the Max-Heap — [6, 5, 3, 1, 8, 7, 2, 4]



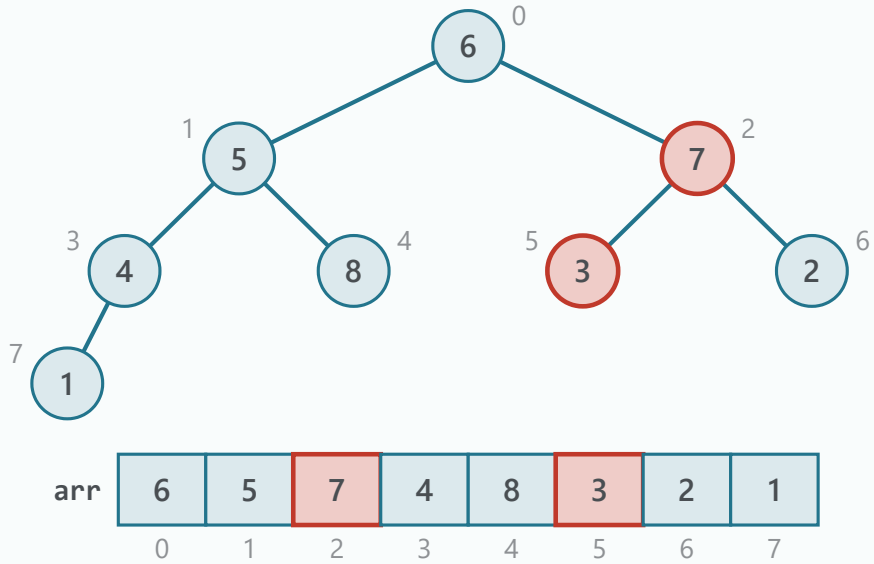
View the array [6, 5, 3, 1, 8, 7, 2, 4] as a complete binary tree: the children of arr[i] are 2i+1 and 2i+2. Not a heap yet.

Phase 1: Building the Max-Heap — [6, 5, 3, 1, 8, 7, 2, 4]



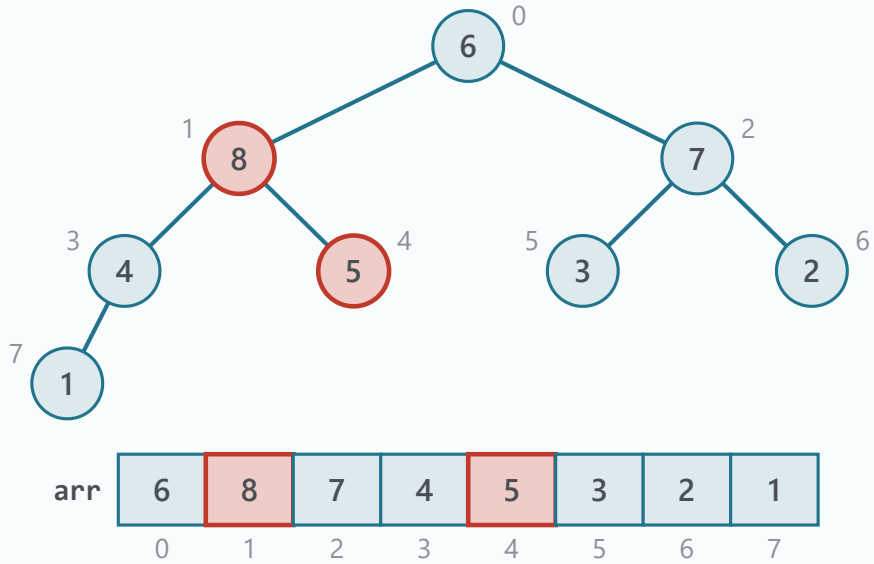
heapify(i = 3): 1 is smaller than its child → swap (1 ↔ 4). Array: [6, 5, 3, 4, 8, 7, 2, 1].

Phase 1: Building the Max-Heap — [6, 5, 3, 1, 8, 7, 2, 4]



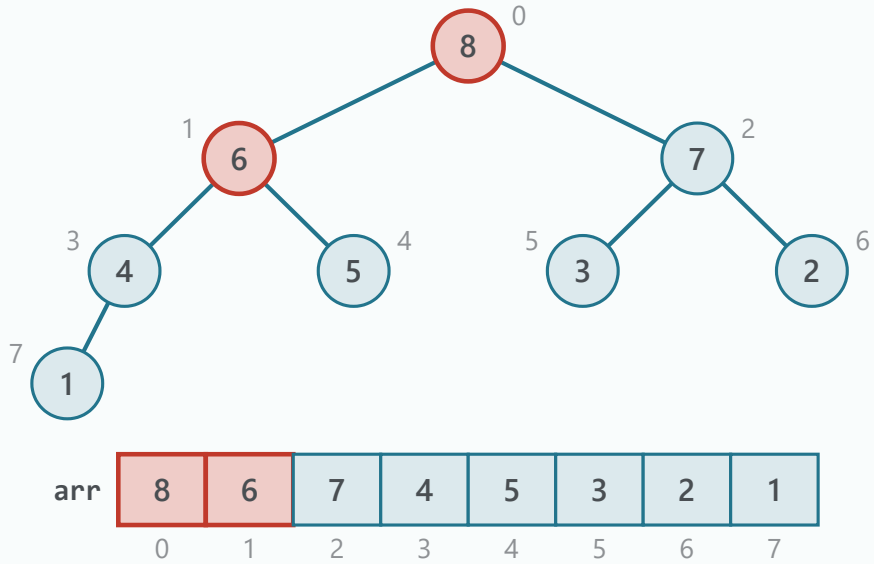
heapify(i = 2): 3 is smaller than its larger child → swap (3 ↔ 7). Array: [6, 5, 7, 4, 8, 3, 2, 1].

Phase 1: Building the Max-Heap — [6, 5, 3, 1, 8, 7, 2, 4]



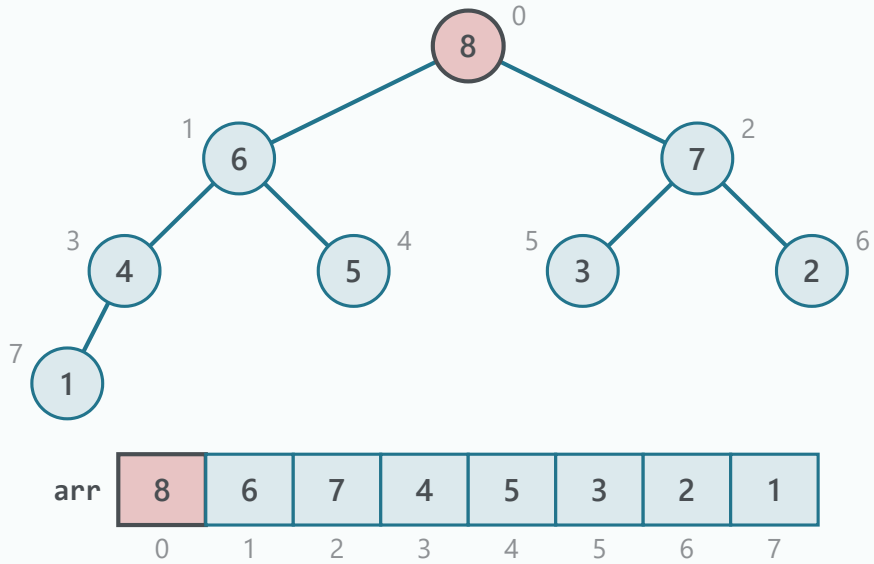
heapify(i = 1): 5 is smaller than its larger child → swap (5 ↔ 8). Array: [6, 8, 7, 4, 5, 3, 2, 1].

Phase 1: Building the Max-Heap — [6, 5, 3, 1, 8, 7, 2, 4]



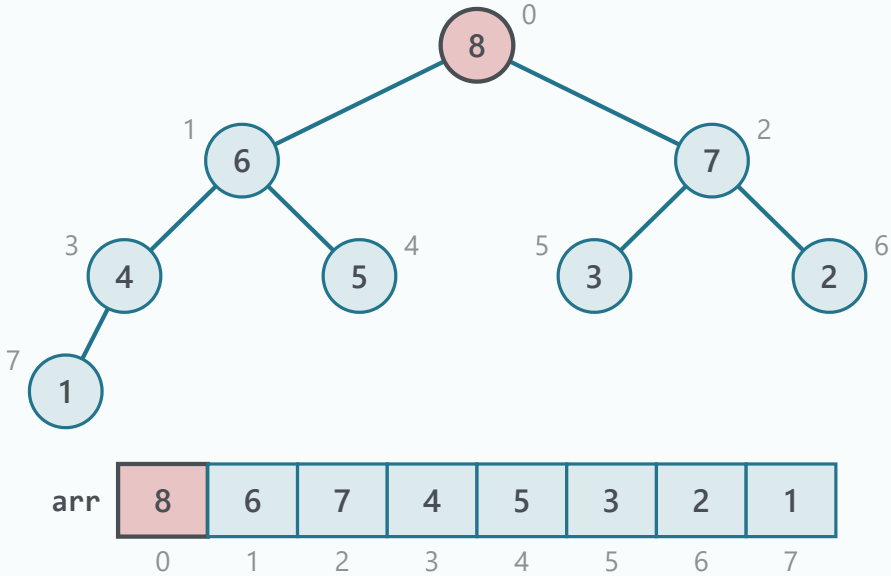
heapify(i = 0): 6 is smaller than its children → swap (6 ↔ 8). Array: [8, 6, 7, 4, 5, 3, 2, 1].

Phase 1: Building the Max-Heap — [6, 5, 3, 1, 8, 7, 2, 4]



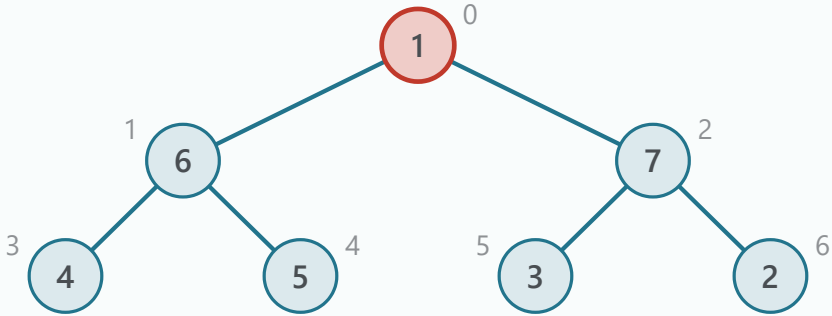
All internal nodes (i = 3 … 0) are done: **max-heap** [8, 6, 7, 4, 5, 3, 2, 1]. The largest item (8) is at the root.

Phase 2: Move the Root to the End, Sift Down



Max-heap: [8, 6, 7, 4, 5, 3, 2, 1]. Each round swaps the root (the largest) with the heap's last item, and the heap shrinks by one.

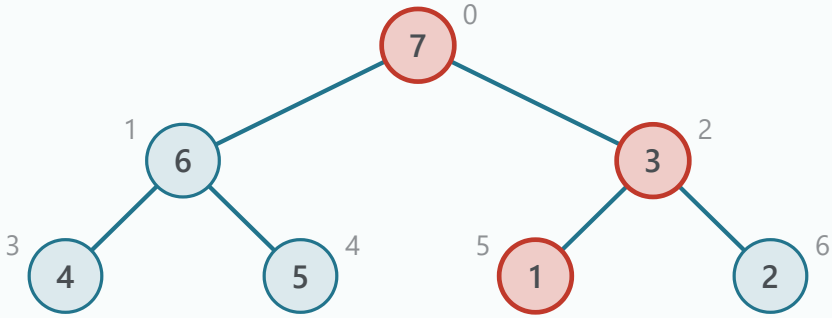
Phase 2: Move the Root to the End, Sift Down





Root 8 ↔ arr[7] = 1: **8** joins the sorted part (green). Heap size 7.

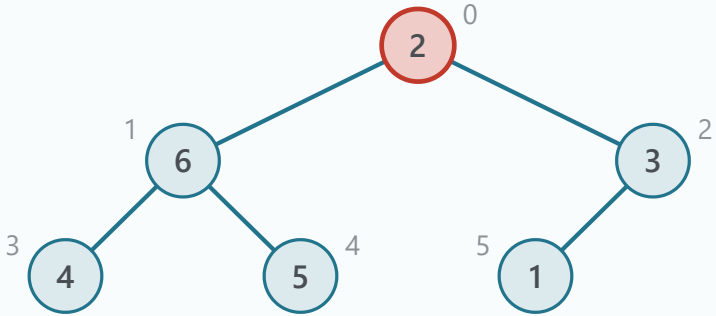
Phase 2: Move the Root to the End, Sift Down





heapify(0) sifts the root down (1 ↔ 7; 1 ↔ 3) → heap: [7, 6, 3, 4, 5, 1, 2].

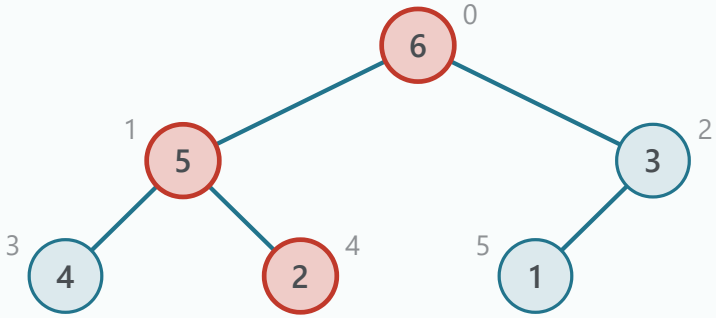
Phase 2: Move the Root to the End, Sift Down

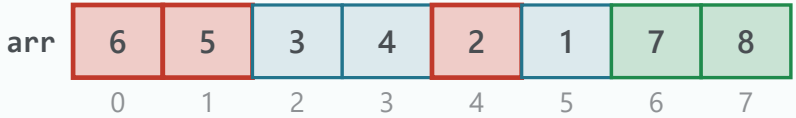




Root 7 ↔ arr[6] = 2: **7** joins the sorted part (green). Heap size 6.

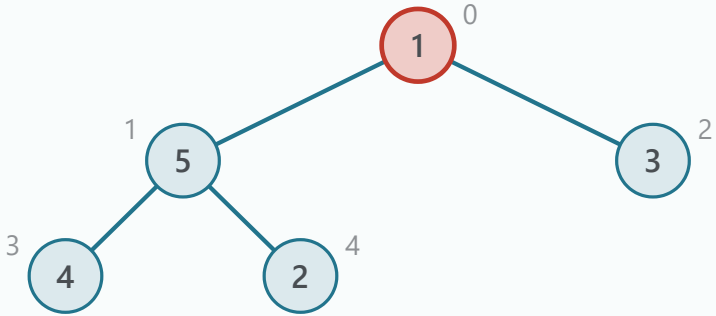
Phase 2: Move the Root to the End, Sift Down





heapify(0) sifts the root down (2 ↔ 6; 2 ↔ 5) → heap: [6, 5, 3, 4, 2, 1].

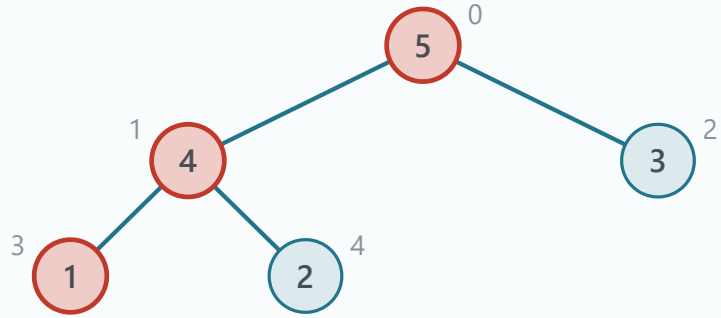
Phase 2: Move the Root to the End, Sift Down

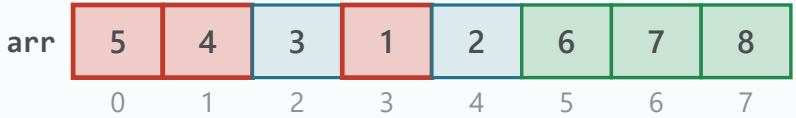




Root 6 ↔ arr[5] = 1: **6** joins the sorted part (green). Heap size 5.

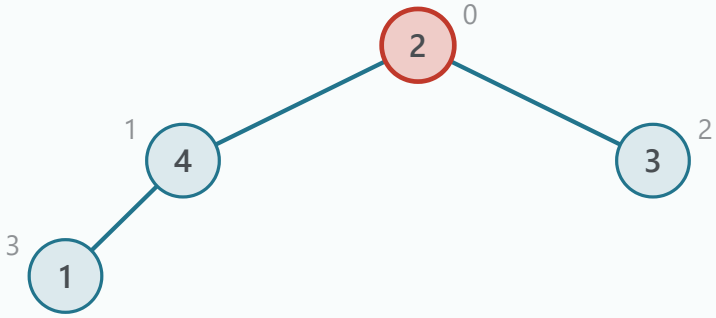
Phase 2: Move the Root to the End, Sift Down

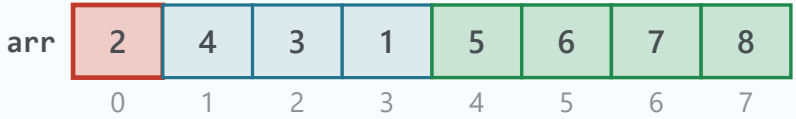




heapify(0) sifts the root down (1 ↔ 5; 1 ↔ 4) → heap: [5, 4, 3, 1, 2].

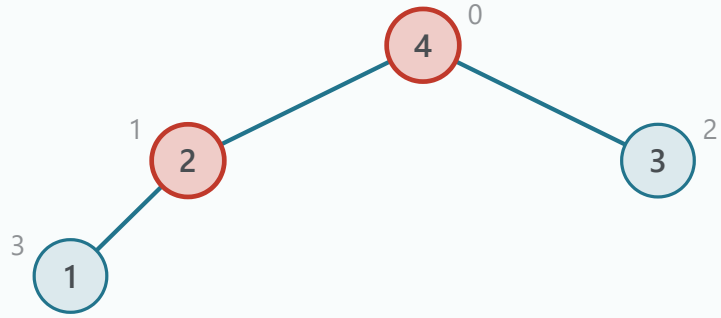
Phase 2: Move the Root to the End, Sift Down

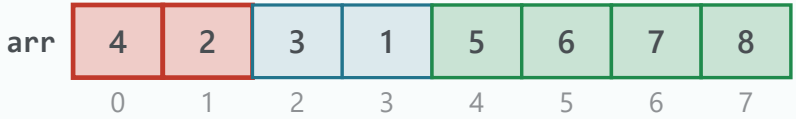




Root 5 ↔ arr[4] = 2: **5** joins the sorted part (green). Heap size 4.

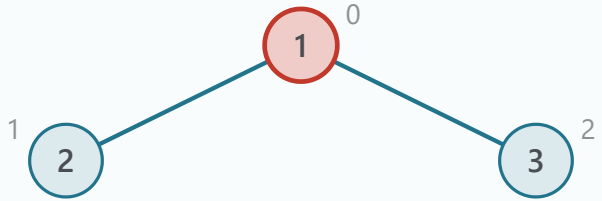
Phase 2: Move the Root to the End, Sift Down





heapify(0) sifts the root down (2 ↔ 4) → heap: [4, 2, 3, 1].

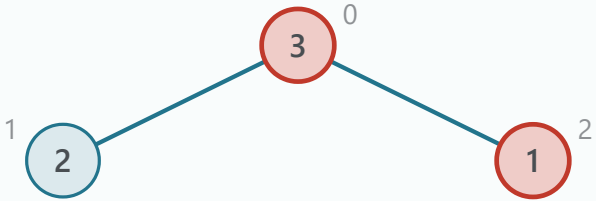
Phase 2: Move the Root to the End, Sift Down





Root 4 ↔ arr[3] = 1: **4** joins the sorted part (green). Heap size 3.

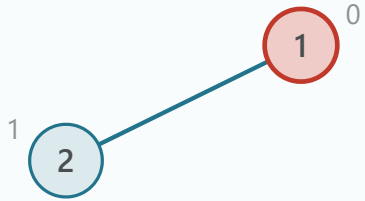
Phase 2: Move the Root to the End, Sift Down

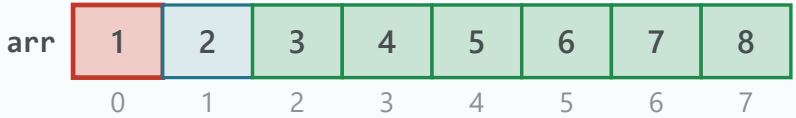




heapify(0) sifts the root down (1 ↔ 3) → heap: [3, 2, 1].

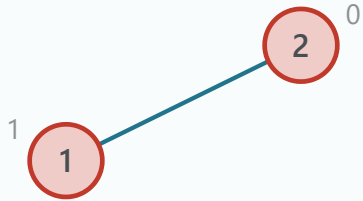
Phase 2: Move the Root to the End, Sift Down





Root 3 ↔ arr[2] = 1: **3** joins the sorted part (green). Heap size 2.

Phase 2: Move the Root to the End, Sift Down





heapify(0) sifts the root down (1 ↔ 2) → heap: [2, 1].

Phase 2: Move the Root to the End, Sift Down





Root 2 ↔ arr[1] = 1: **2** joins the sorted part (green). Heap size 1.

Phase 2: Move the Root to the End, Sift Down





One item is left in the heap, and it is in place. Sorted array: [1, 2, 3, 4, 5, 6, 7, 8].

Heapsort: Java Code and Analysis

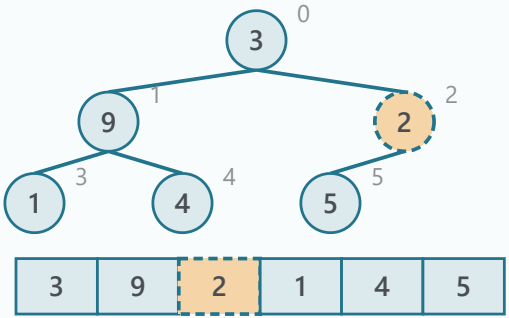
```java
void heapSort(int[] arr) {
  int n = arr.length;
  for (int i = n / 2 - 1; i >= 0; i--)   // 1) build heap
    heapify(arr, n, i);
  for (int i = n - 1; i > 0; i--) {      // 2) sort
    swap(arr, 0, i);                    // max to the end
    heapify(arr, i, 0);                 // sift root down
  }
}
void heapify(int[] arr, int n, int i) {
  while (true) {
    int largest = i, left = 2 * i + 1, right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;
    if (largest == i) return;           // in place
    swap(arr, i, largest);
    i = largest;                        // keep sifting
  }
}
```

- **Building the heap**: the leaves ($i \geq n/2$) are already one-item heaps; sift down from $i = n/2 - 1$ toward the root. Total **O(n)** (most nodes are near the bottom).
- **Sorting**: $n - 1$ times, move the root to the end and sift down in $O(\log n)$ → **O(n log n)**.
- $O(n \log n)$ even in the worst case, with O(1) extra memory (in place).
- **Unstable**: the root–last swap can reorder equal keys.

A heap can also be built by inserting items one by one (sift up); that too gives a valid max-heap, but takes $O(n \log n)$. The bottom-up construction used here (Floyd's method) is O(n).

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
▶ for (int i = n / 2 - 1; i >= 0; i--)
▶   heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
6      int largest = i, left = 2 * i + 1, right = 2
   * i + 2;
7      if (left < n && arr[left] > arr[largest])
   largest = left;
8      if (right < n && arr[right] > arr[largest])
   largest = right;
9      if (largest == i) return;          // in
   place
10     swap(arr, i, largest);
11     i = largest;                       // keep
   sifting
12   }
13 }
```



arr = [3, 9, 2, 1, 4, 5]. The leaves (indices 3, 4, 5) are already one-item heaps. Building runs from i = n / 2 − 1 = **2** toward the root.

Step 1 / 13

VARIABLES

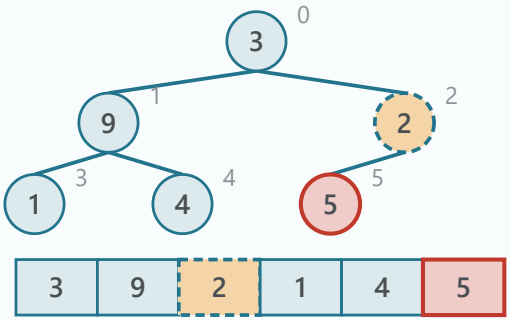
`n = 6` `i = 2` `left = –` `right = –` `largest = –`

CALL STACK

`heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
▶      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
▶      if (left < n && arr[left] > arr[largest]) largest = left;
▶      if (right < n && arr[right] > arr[largest]) largest = right;
9      if (largest == i) return;          // in place
10     swap(arr, i, largest);
11     i = largest;                       // keep sifting
12   }
13 }
```



i = 2 (2): left = 5 (5) > 2 → largest = left; no right child.

Step 2 / 13

VARIABLES

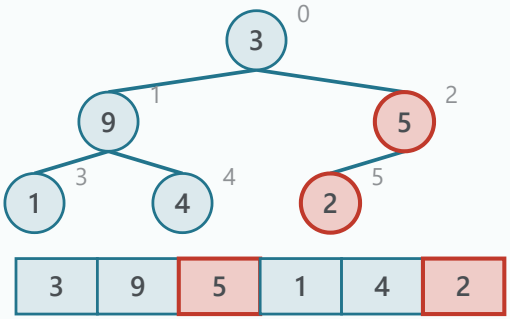
`n = 6` `i = 2` `left = 5` `right = 6` `largest = 5`

CALL STACK

- `heapify(2)`
- `heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
6      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
7      if (left < n && arr[left] > arr[largest]) largest = left;
8      if (right < n && arr[right] > arr[largest]) largest = right;
▶      if (largest == i) return;          // in place
▶      swap(arr, i, largest);
▶      i = largest;                       // keep sifting
12   }
13 }
```



largest ≠ i: `swap(arr, 2, 5)` → 5 moves up, 2 moves down; the loop continues with `i = 5` (line 11).

Step 3 / 13

VARIABLES

`n = 6` `i = 2` `left = 5` `right = 6` `largest = 5`

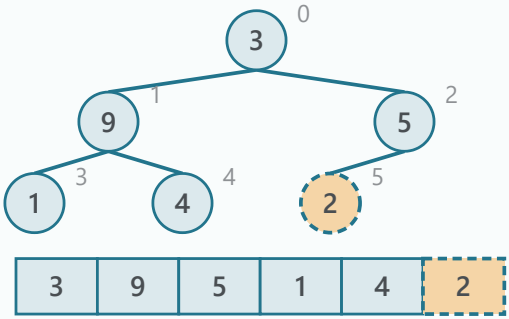
CALL STACK

`heapify(2)`

`heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
▶      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
▶      if (left < n && arr[left] > arr[largest]) largest = left;
▶      if (right < n && arr[right] > arr[largest]) largest = right;
▶      if (largest == i) return;          // in place
10     swap(arr, i, largest);
11     i = largest;                       // keep sifting
12   }
13 }
```



i = 5 (2): no children (left = 11 ≥ n). largest == i → `return`, sifting ends.

Step 4 / 13

VARIABLES

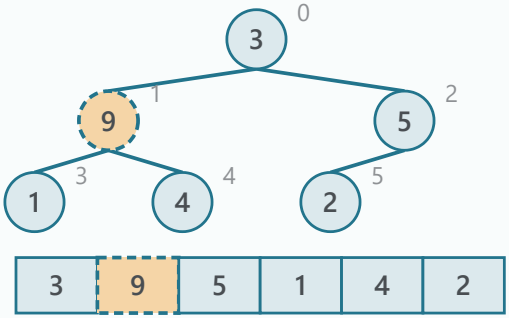
- n = 6
- i = 5
- left = 11
- right = 12
- largest = 5

CALL STACK

- heapify(2)
- heapSort

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
▶ for (int i = n / 2 - 1; i >= 0; i--)
▶   heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
6      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
7      if (left < n && arr[left] > arr[largest]) largest = left;
8      if (right < n && arr[right] > arr[largest]) largest = right;
9      if (largest == i) return;          // in place
10     swap(arr, i, largest);
11     i = largest;                       // keep sifting
12   }
13 }
```



Loop i = 1: `heapify(arr, 6, 1)`.

Step 5 / 13

VARIABLES

`n = 6` `i = 1`

`left = –`

`right = –`

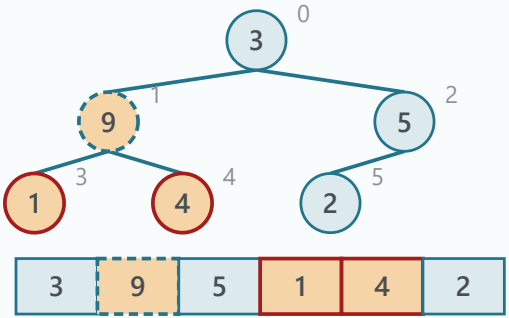
`largest = –`

CALL STACK

`heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
▶      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
▶      if (left < n && arr[left] > arr[largest]) largest = left;
▶      if (right < n && arr[right] > arr[largest]) largest = right;
▶      if (largest == i) return;          // in place
10     swap(arr, i, largest);
11     i = largest;                       // keep sifting
12   }
13 }
```



i = 1 (9): left = 3 (1) is not > 9; right = 4 (4) is not > 9. No change: it is already larger than its children.

Step 6 / 13

VARIABLES

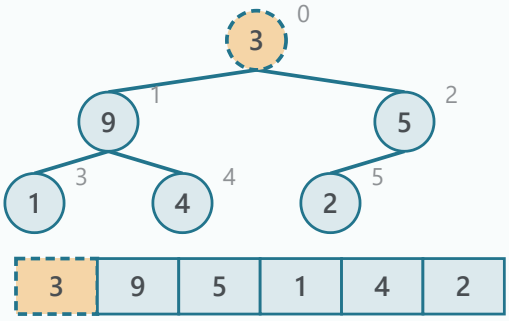
`n = 6` `i = 1` `left = 3` `right = 4` `largest = 1`

CALL STACK

- **heapify(1)**
- heapSort

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
▶ for (int i = n / 2 - 1; i >= 0; i--)
▶   heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
6      int largest = i, left = 2 * i + 1, right = 2
   * i + 2;
7      if (left < n && arr[left] > arr[largest])
   largest = left;
8      if (right < n && arr[right] > arr[largest])
   largest = right;
9      if (largest == i) return;          // in
   place
10     swap(arr, i, largest);
11     i = largest;                       // keep
   sifting
12   }
13 }
```



Loop i = 0: `heapify(arr, 6, 0)`.

Step 7 / 13

VARIABLES

`n = 6` `i = 0`

`left = –`

`right = –`

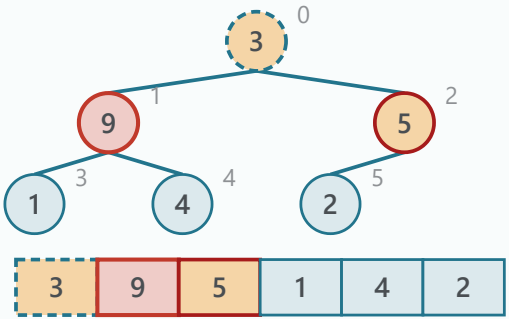
`largest = –`

CALL STACK

`heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
▶      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
▶      if (left < n && arr[left] > arr[largest]) largest = left;
▶      if (right < n && arr[right] > arr[largest]) largest = right;
9      if (largest == i) return;          // in place
10     swap(arr, i, largest);
11     i = largest;                       // keep sifting
12   }
13 }
```



i = 0 (3): left = 1 (9) > 3 → largest = left; right = 2 (5) is not > 9.

Step 8 / 13

VARIABLES

`n = 6` `i = 0` `left = 1` `right = 2` `largest = 1`

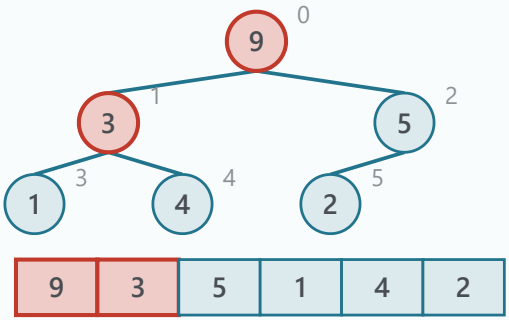
CALL STACK

`heapify(0)`

`heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
6      int largest = i, left = 2 * i + 1, right = 2
   * i + 2;
7      if (left < n && arr[left] > arr[largest])
   largest = left;
8      if (right < n && arr[right] > arr[largest])
   largest = right;
▶      if (largest == i) return;          // in
   place
▶      swap(arr, i, largest);
▶      i = largest;                       // keep
   sifting
12   }
13 }
```



largest ≠ i: `swap(arr, 0, 1)` → 9 moves up, 3 moves down; the loop continues with `i = 1` (line 11).

Step 9 / 13

VARIABLES

`n = 6` `i = 0` `left = 1` `right = 2` `largest = 1`

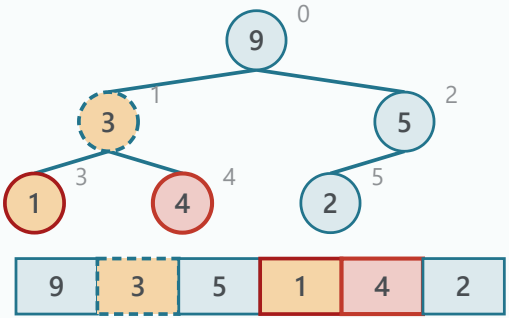
CALL STACK

`heapify(0)`

`heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
▶      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
▶      if (left < n && arr[left] > arr[largest]) largest = left;
▶      if (right < n && arr[right] > arr[largest]) largest = right;
9      if (largest == i) return;          // in place
10     swap(arr, i, largest);
11     i = largest;                       // keep sifting
12   }
13 }
```



i = 1 (3): left = 3 (1) is not > 3; right = 4 (4) > 3 → largest = right.

Step 10 / 13

VARIABLES

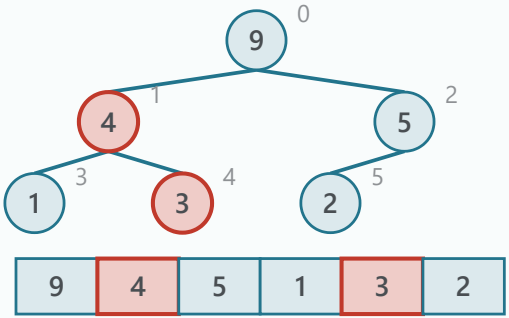
- n = 6
- i = 1
- left = 3
- right = 4
- largest = 4

CALL STACK

- heapify(0)
- heapSort

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
6      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
7      if (left < n && arr[left] > arr[largest]) largest = left;
8      if (right < n && arr[right] > arr[largest]) largest = right;
▶      if (largest == i) return;          // in place
▶      swap(arr, i, largest);
▶      i = largest;                       // keep sifting
12   }
13 }
```



largest ≠ i: `swap(arr, 1, 4)` → 4 moves up, 3 moves down; the loop continues with `i = 4` (line 11).

Step 11 / 13

VARIABLES

`n = 6` `i = 1` `left = 3` `right = 4` `largest = 4`

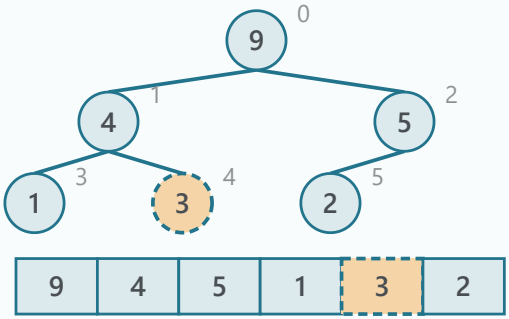
CALL STACK

`heapify(0)`

`heapSort`

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
1  for (int i = n / 2 - 1; i >= 0; i--)
2    heapify(arr, n, i);
3
4  void heapify(int[] arr, int n, int i) {
5    while (true) {
▶      int largest = i, left = 2 * i + 1, right = 2 * i + 2;
▶      if (left < n && arr[left] > arr[largest]) largest = left;
▶      if (right < n && arr[right] > arr[largest]) largest = right;
▶      if (largest == i) return;          // in place
10     swap(arr, i, largest);
11     i = largest;                       // keep sifting
12   }
13 }
```



i = 4 (3): no children (left = 9 ≥ n). largest == i → `return`, sifting ends.

Step 12 / 13

VARIABLES

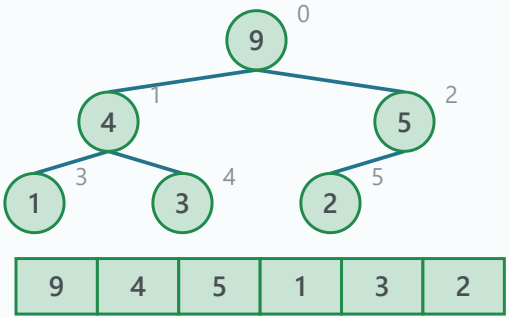
- n = 6
- i = 4
- left = 9
- right = 10
- largest = 4

CALL STACK

- heapify(0)
- heapSort

Building a Max-Heap: Code Trace, arr = [3, 9, 2, 1, 4, 5]

```
▶ for (int i = n / 2 - 1; i >= 0; i--)
2   heapify(arr, n, i);
3
4 void heapify(int[] arr, int n, int i) {
5   while (true) {
6     int largest = i, left = 2 * i + 1, right = 2 * i + 2;
7     if (left < n && arr[left] > arr[largest]) largest = left;
8     if (right < n && arr[right] > arr[largest]) largest = right;
9     if (largest == i) return;          // in place
10    swap(arr, i, largest);
11    i = largest;                       // keep sifting
12  }
13 }
```



i = −1: the loop ends. **[9, 4, 5, 1, 3, 2]** is a max-heap: every parent is larger than its children, and the largest item, 9, is at the root.

Step 13 / 13

VARIABLES

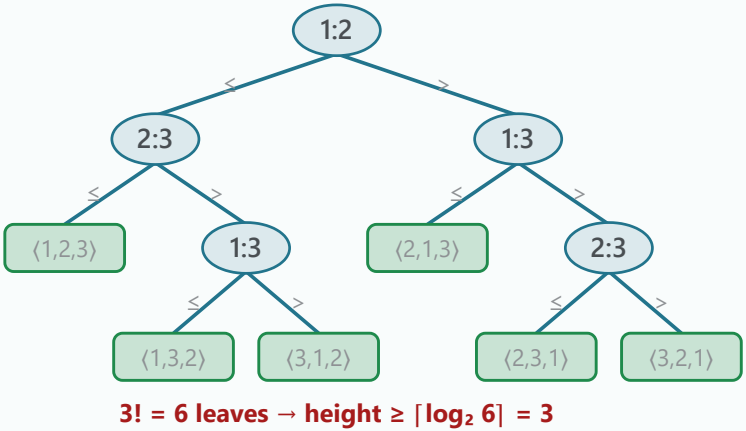
n = **6** i = **−1** left = **–** right = **–** largest = **–**

CALL STACK

heapSort

A Lower Bound for Comparison Sorting

- Any algorithm that only asks "$a_i \le a_j$?" can be drawn as a **decision tree**.
- Each leaf is an ordering (a permutation); to be correct the tree needs **n!** leaves.
- A binary tree of height h has at most 2^h leaves: $2^h \ge n! \Rightarrow h \ge \log_2(n!) = \mathbf{\Omega(n \log n)}$.
- Consequence: merge sort and heapsort are **asymptotically optimal**.
- Going faster needs information beyond comparisons: small integer keys → **counting, radix**.



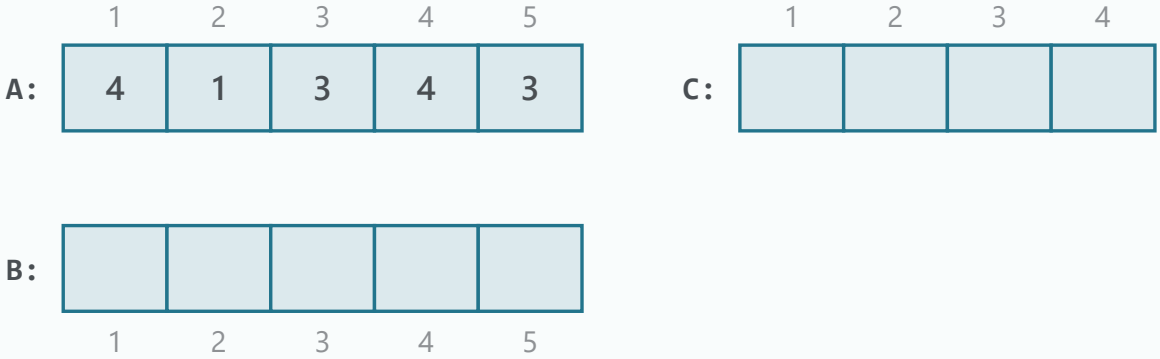
Counting Sort

- Items are **never compared** with each other.
- **Input**: A[1 … n], A[j] ∈ {1, 2, …, k}.
- **Output**: B[1 … n], sorted.
- **Temporary**: counters C[1 … k].

Idea: if we know how many items have key ≤ i, the last item with key i goes exactly at that position.

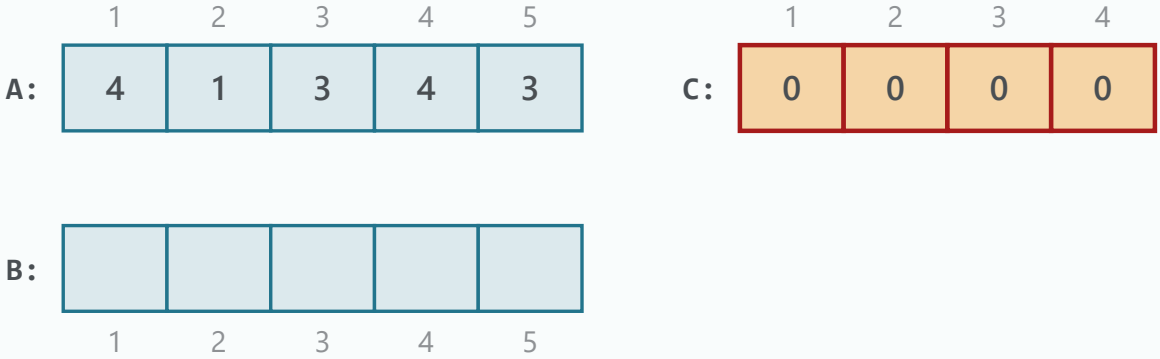
```
for i = 1 to k
  C[i] = 0
for j = 1 to n
  C[A[j]] = C[A[j]] + 1    // C[i] = |{key = i}|
for i = 2 to k
  C[i] = C[i] + C[i-1]     // C[i] = |{key ≤ i}|
for j = n downto 1
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



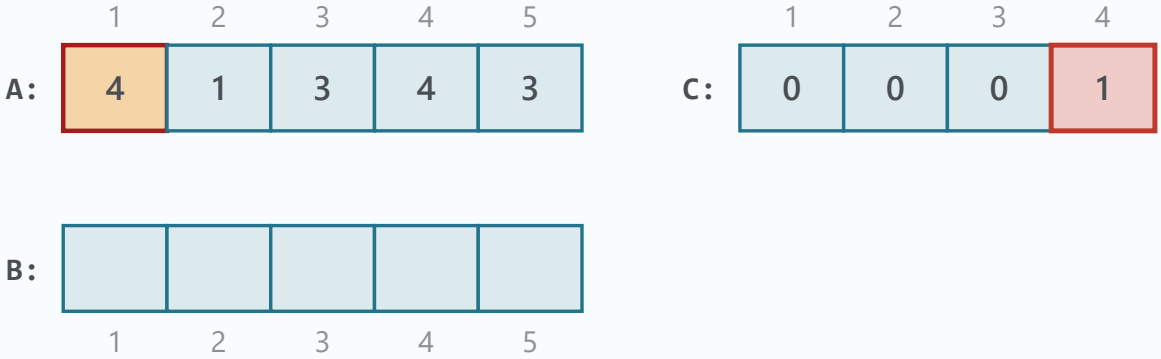
Input A = [4, 1, 3, 4, 3], n = 5, keys {1, …, 4}. C: counter array (k = 4), B: output.

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



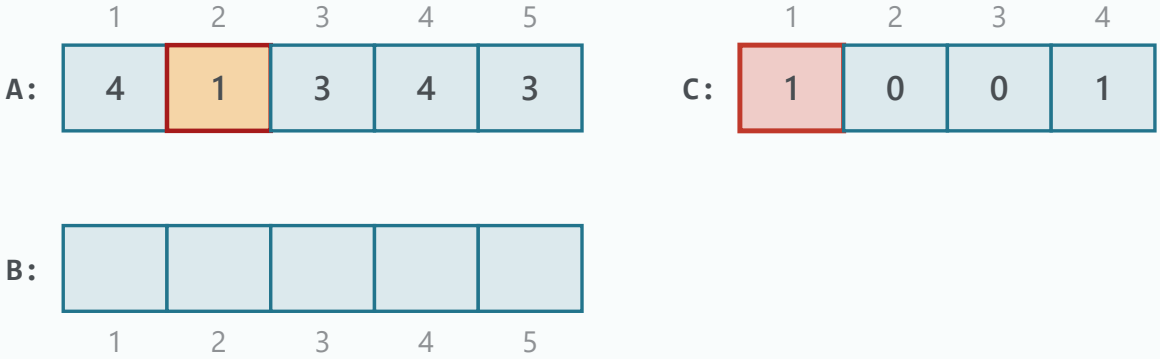
Loop 1: `for i = 1 to k: C[i] = 0` — all counters are reset. Θ(k).

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



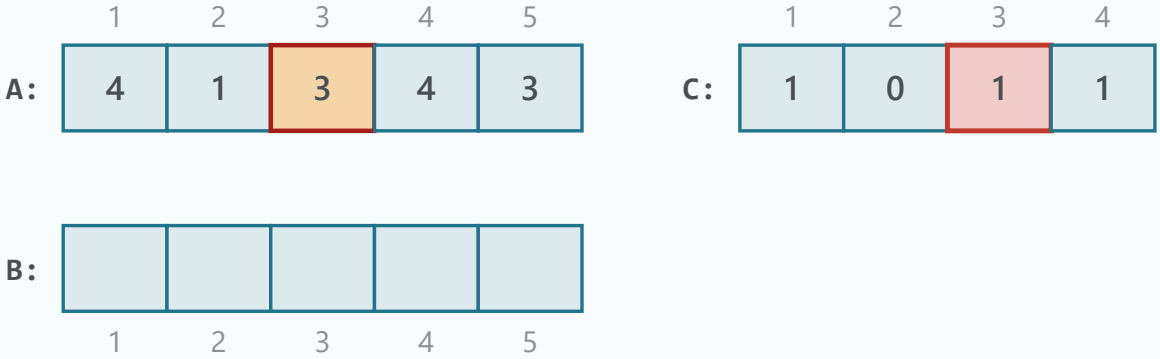
Loop 2, j = 1: A[1] = 4 → C[4]++ = 1. (C[i] = count of key i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



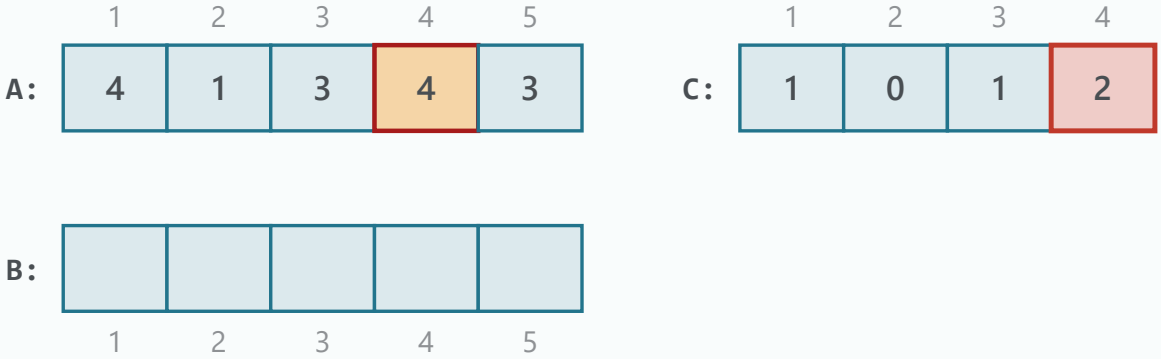
Loop 2, j = 2: A[2] = 1 → C[1]++ = 1. (C[i] = count of key i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



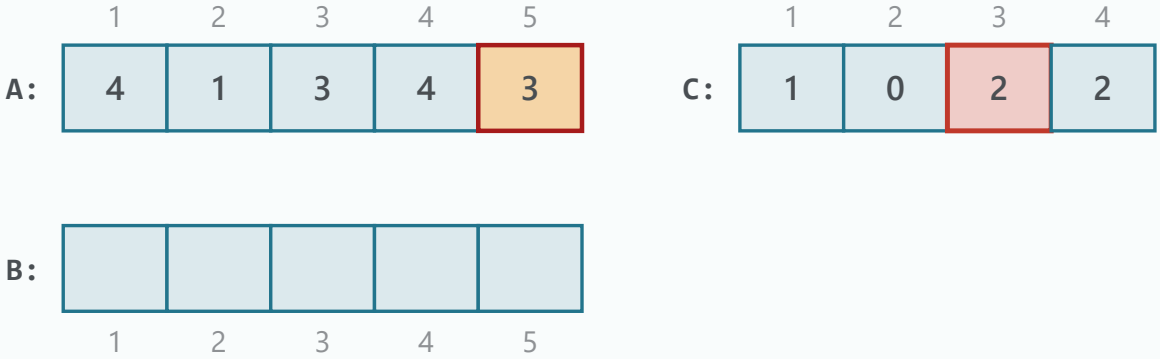
Loop 2, j = 3: A[3] = 3 → C[3]++ = 1. (C[i] = count of key i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



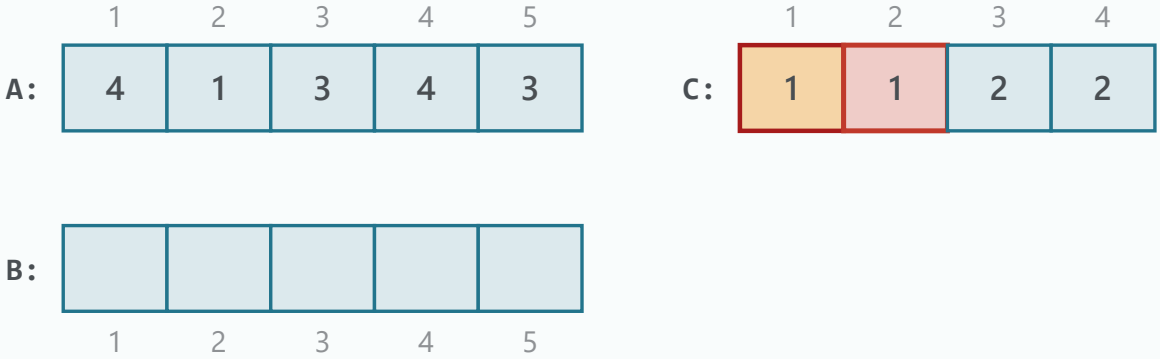
Loop 2, j = 4: A[4] = 4 → C[4]++ = 2. (C[i] = count of key i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



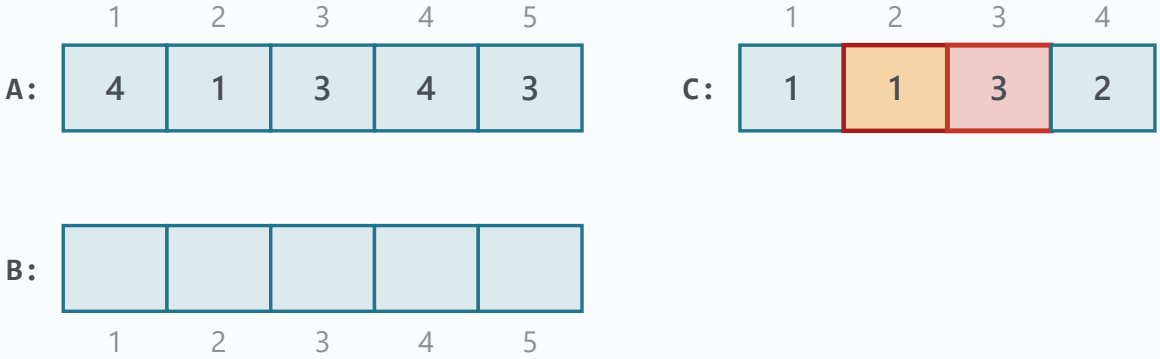
Loop 2, j = 5: A[5] = 3 → C[3]++ = 2. (C[i] = count of key i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



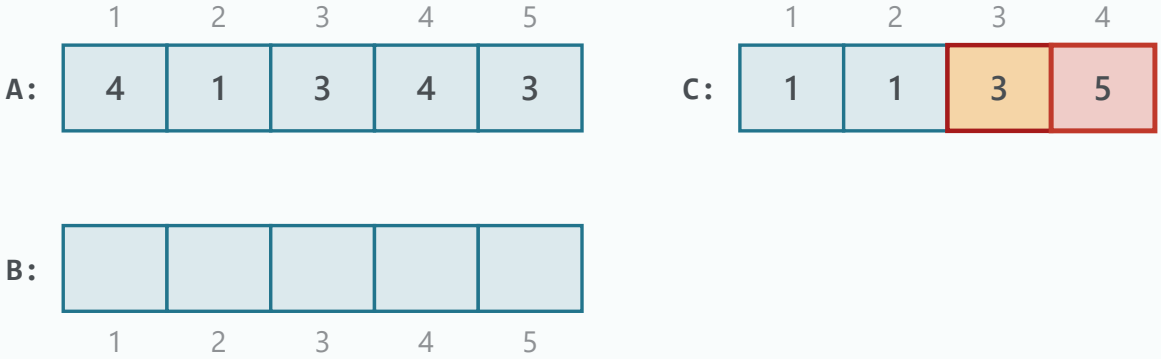
Loop 3, i = 2: C[2] = C[2] + C[1] = 1. (C[i] = count of keys ≤ i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



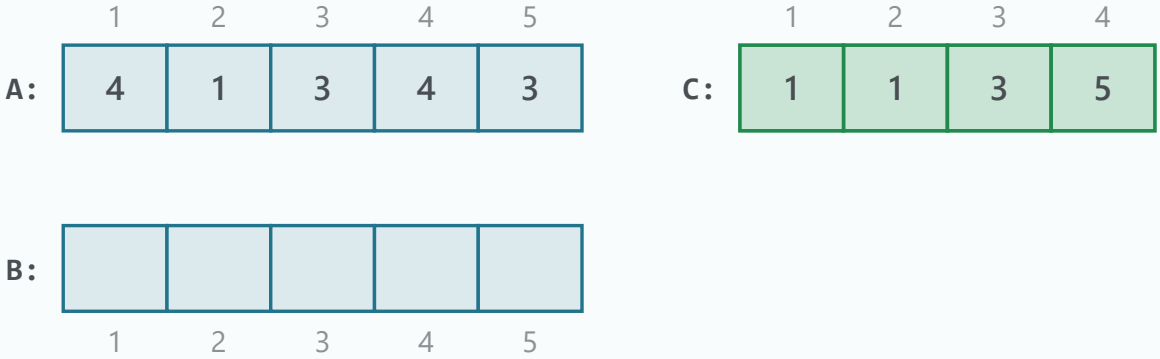
Loop 3, i = 3: C[3] = C[3] + C[2] = 3. (C[i] = count of keys ≤ i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



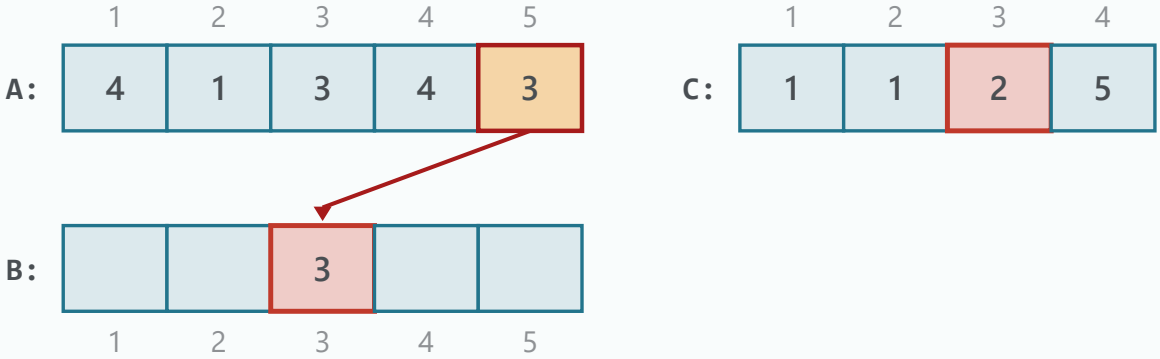
Loop 3, i = 4: C[4] = C[4] + C[3] = 5. (C[i] = count of keys ≤ i)

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



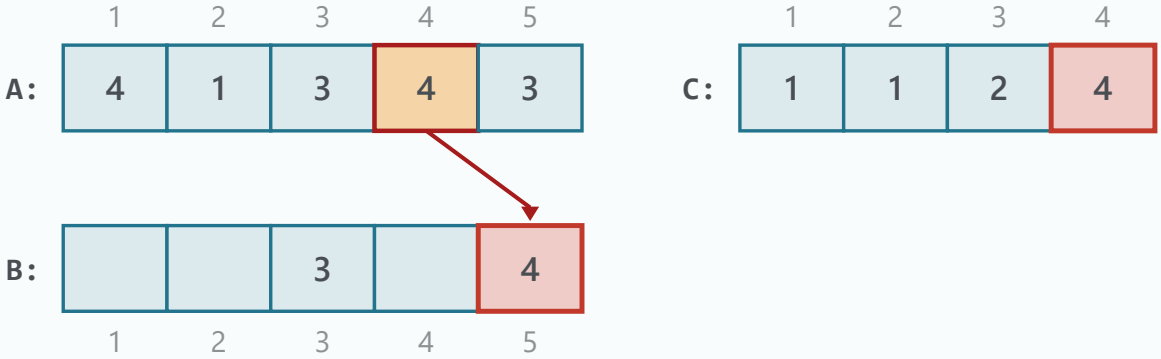
The prefix sums are ready: C = [1, 1, 3, 5]. C[i] is the position in B of the last item with key i.

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



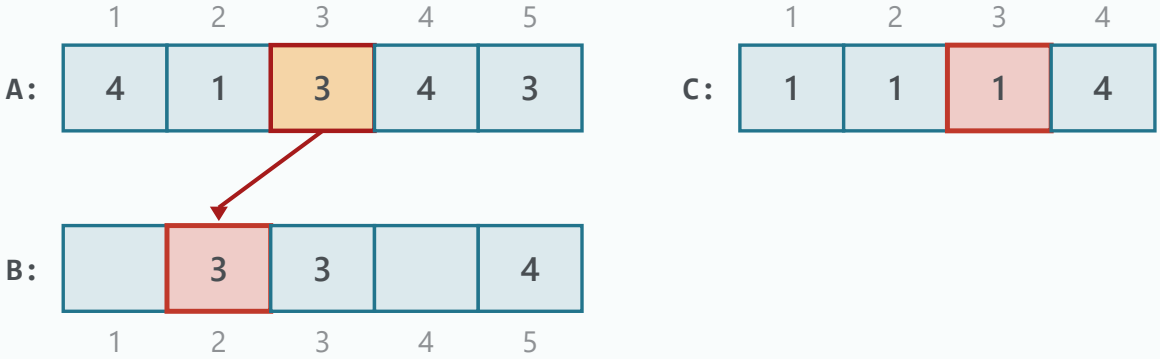
Loop 4, j = 5: A[5] = 3 → B[C[3] = 3] = 3; C[3]−− = 2.

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



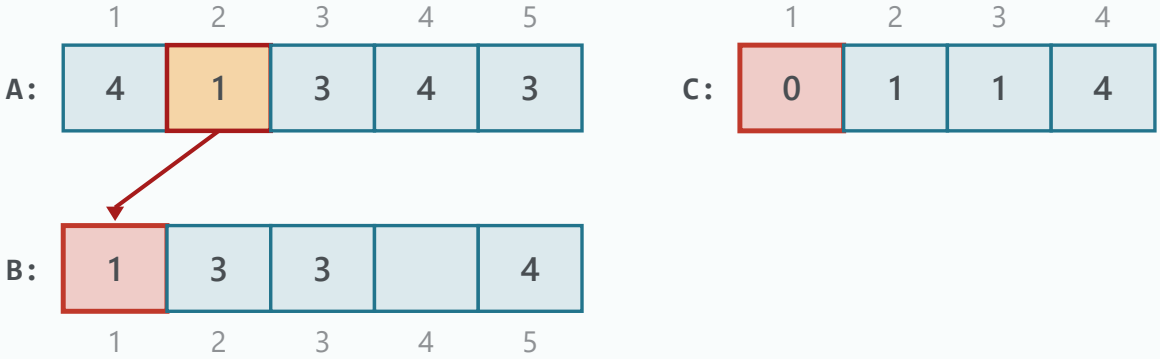
Loop 4, j = 4: A[4] = 4 → B[C[4] = 5] = 4; C[4]−− = 4.

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



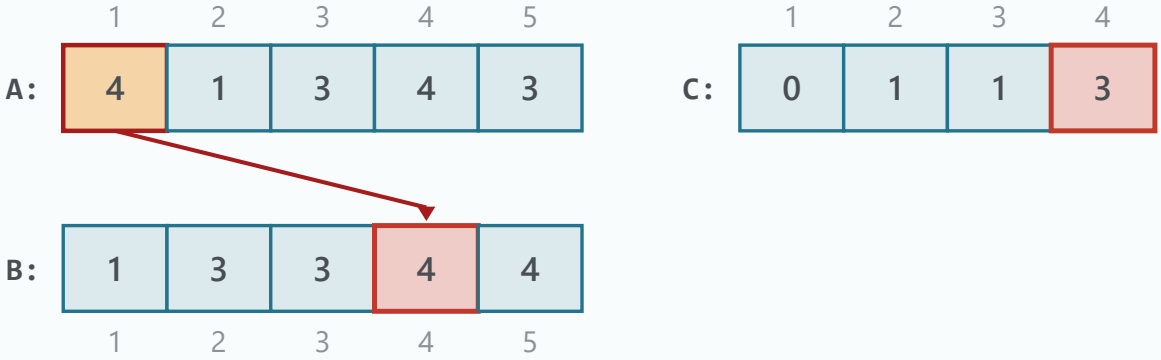
Loop 4, j = 3: A[3] = 3 → B[C[3] = 2] = 3; C[3]−− = 1.

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



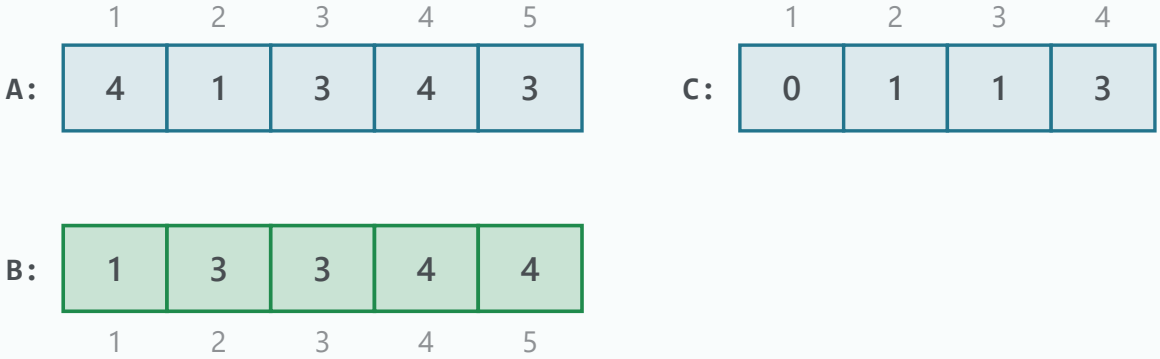
Loop 4, j = 2: A[2] = 1 → B[C[1] = 1] = 1; C[1]−− = 0.

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



Loop 4, j = 1: A[1] = 4 → B[C[4] = 4] = 4; C[4]−− = 3.

Counting Sort Step by Step: A = [4, 1, 3, 4, 3]



Result B = [1, 3, 3, 4, 4]. No comparisons were made; scanning from back to front keeps equal keys in order (**stable**).

Counting Sort: Complexity and Java

Loop	Work	Time
1	C[1..k] = 0	Θ(k)
2	counting	Θ(n)
3	prefix sums	Θ(k)
4	place into B	Θ(n)
Total		**Θ(n + k)**

```java
int[] countingSort(int[] arr) { // elements ≥ 0
  if (arr.length == 0) return arr;
  int max = Arrays.stream(arr).max().getAsInt();
  int[] count = new int[max + 1];
  for (int x : arr) count[x]++;
  for (int i = 1; i <= max; i++) count[i] += count[i - 1];
  int[] output = new int[arr.length];
  for (int i = arr.length - 1; i >= 0; i--) {
    output[count[arr[i]] - 1] = arr[i];
    count[arr[i]]--;
  }
  return output;
}
```

- If k = O(n) it is **Θ(n)**: it beats the Ω(n log n) bound because it makes no comparisons.
- Extra memory Θ(n + k); impractical if k is huge (e.g. 32-bit integers). The Java version handles only non-negative integers and uses the range 0 … max.
- Scanning the last loop **from back to front** makes it stable → the basis of radix sort.

Radix Sort

- Sorts non-negative integers **digit by digit**, starting from the least significant digit (the ones): **LSD**.
- Each pass does a **stable** sort (counting sort, k = 10) on that digit.
- Stability is essential: sorting by the tens digit must preserve the order of the ones.
- d digits, base b: **Θ(d · (n + b))**. Θ(n) if d is constant.
- Extra memory Θ(n + b); not in place.

```
int[] radixSort(int[] arr) { // elements ≥ 0
  if (arr.length == 0) return arr;
  int max = Arrays.stream(arr).max().getAsInt();
  for (long exp = 1; max / exp > 0; exp *= 10)
    arr = countByDigit(arr, (int) exp);
  return arr;
}
int[] countByDigit(int[] arr, int exp) {
  int n = arr.length;
  int[] output = new int[n], count = new int[10];
  for (int x : arr) count[(x / exp) % 10]++;
  for (int i = 1; i < 10; i++) count[i] += count[i - 1];
  for (int i = n - 1; i >= 0; i--) {
    int b = (arr[i] / exp) % 10;
    output[--count[b]] = arr[i];
  }
  return output;
}
```

Radix Sort Step by Step

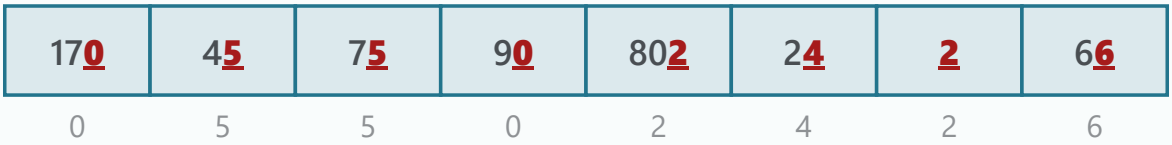
Input

170	45	75	90	802	24	2	66

Input [170, 45, 75, 90, 802, 24, 2, 66]. The largest number is 802: 3 digits → 3 passes (starting from the least significant digit, LSD).

Radix Sort Step by Step

exp = 1: ones digit



Pass 1: each number's ones digit (shown below) is the key; a missing digit counts as 0.

Radix Sort Step by Step

sorted by the ones digit

17**0**	9**0**	80**2**	**2**	2**4**	4**5**	7**5**	6**6**
0	0	2	2	4	5	5	6

After the stable counting sort: [170, 90, 802, 2, 24, 45, 75, 66]. Numbers with equal digits kept their previous order.

Radix Sort Step by Step

exp = 10: tens digit

170	90	802	2	24	45	75	66
7	9	0	0	2	4	7	6

Pass 2: each number's tens digit (shown below) is the key; a missing digit counts as 0.

Radix Sort Step by Step

sorted by the tens digit

802	2	24	45	66	170	75	90
0	0	2	4	6	7	7	9

After the stable counting sort: [802, 2, 24, 45, 66, 170, 75, 90]. Numbers with equal digits kept their previous order.

Radix Sort Step by Step

exp = 100: hundreds digit

802	2	24	45	66	170	75	90
8	0	0	0	0	1	0	0

Pass 3: each number's hundreds digit (shown below) is the key; a missing digit counts as 0.

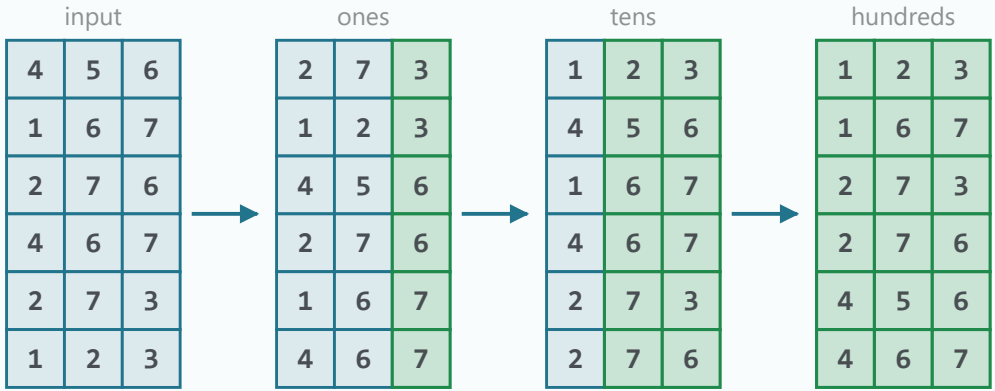
Radix Sort Step by Step

sorted by the hundreds digit

2	24	45	66	75	90	170	802
0	0	0	0	0	0	1	8

After the stable counting sort: [2, 24, 45, 66, 75, 90, 170, 802]. All digits processed → **sorted**.

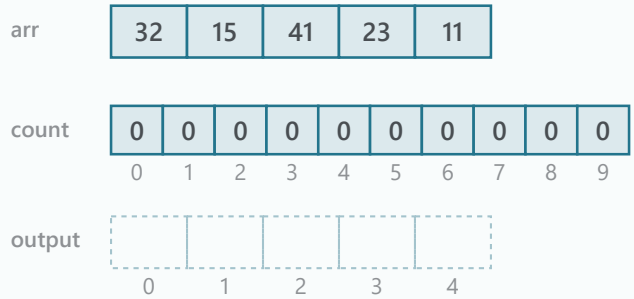
Radix Sort: Why Stability Is Needed



- The green columns are the **suffixes sorted so far**: after k passes the array is sorted by its last k digits.
- In pass 2, 167 and 467 have the same tens digit (6); the order from pass 1 (7 = 7, input order) is preserved.
- If the inner sort were **unstable**, 273 and 276, say, could swap in the last pass and ruin the result.

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
▶    int n = arr.length;
▶    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



exp = 1: sorting by the **ones** digit.

`count[0..9]` is zero, `output` is empty.

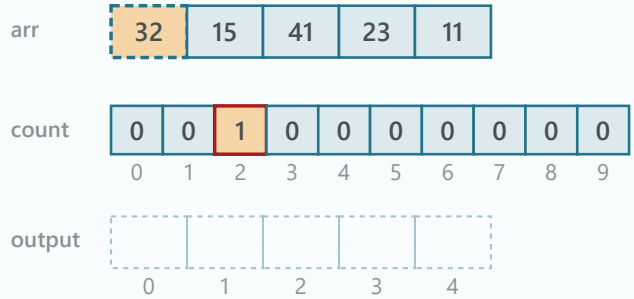
Step 1 / 13

VARIABLES

`exp = 1` `n = 5` `x = –` `b = –`

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
▶    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



x = 32: digit (32 / 1) % 10 = **2** → `count[2]++` = 1.

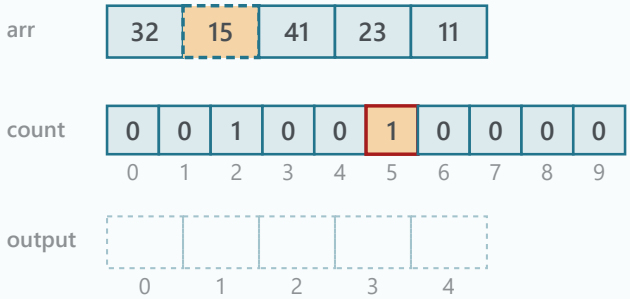
Step 2 / 13

VARIABLES

exp = **1** | n = **5** | x = **32** | b = **2**

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
▶    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



x = 15: digit (15 / 1) % 10 = **5** → `count[5]++` = 1.

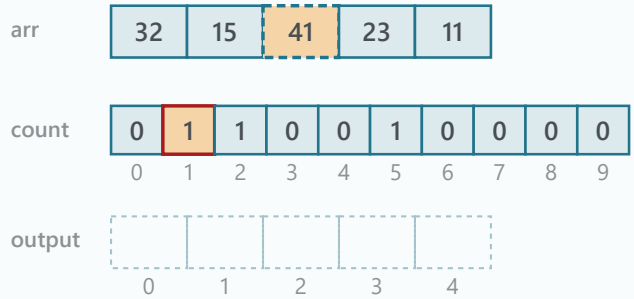
Step 3 / 13

VARIABLES

exp = **1** | n = **5** | x = **15** | b = **5**

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
▶    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



x = 41: digit (41 / 1) % 10 = **1** → `count[1]++` = 1.

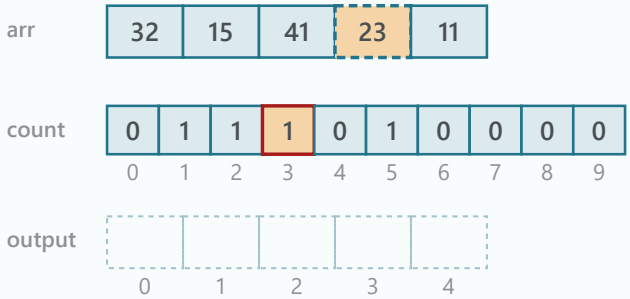
Step 4 / 13

VARIABLES

exp = **1** n = **5** x = **41** b = **1**

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
▶    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



x = 23: digit (23 / 1) % 10 = **3** → `count[3]++` = 1.

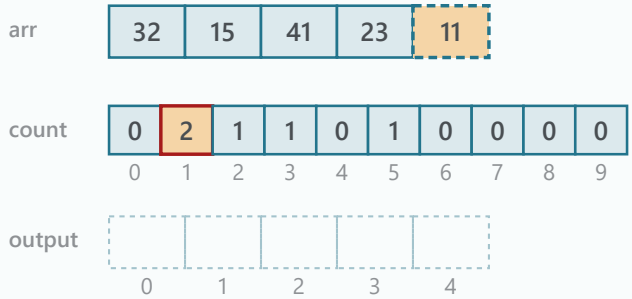
Step 5 / 13

VARIABLES

exp = **1** | n = **5** | x = **23** | b = **3**

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
▶    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



x = 11: digit (11 / 1) % 10 = **1** → `count[1]++` = 2.

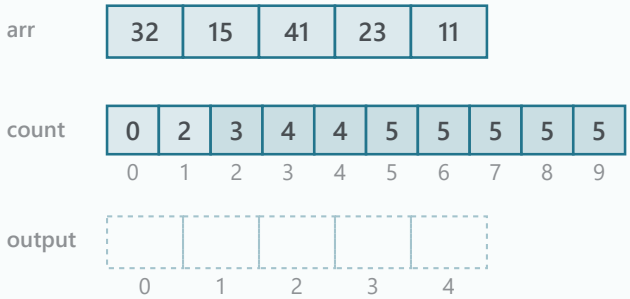
Step 6 / 13

VARIABLES

exp = **1** | n = **5** | x = **11** | b = **1**

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
▶    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



Prefix sums: `count[b]` is now the number of items with digit ≤ b; e.g. count[3] = 4. This is one more than the position in `output` of the last item with that digit.

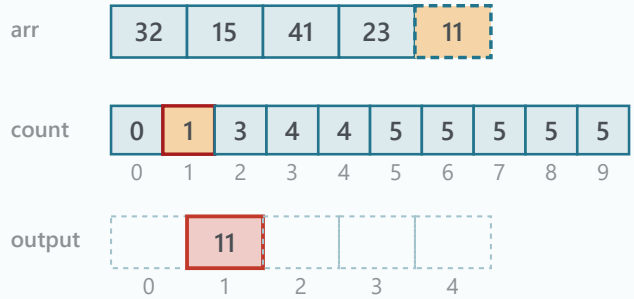
Step 7 / 13

VARIABLES

exp = 1 | n = 5 | x = – | b = –

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
▶    for (int i = n - 1; i >= 0; i--) {
▶      int b = (arr[i] / exp) % 10;
▶      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



i = 4: x = 11, b = 1. `--count[1]` = 1 → `output[1] = 11`.

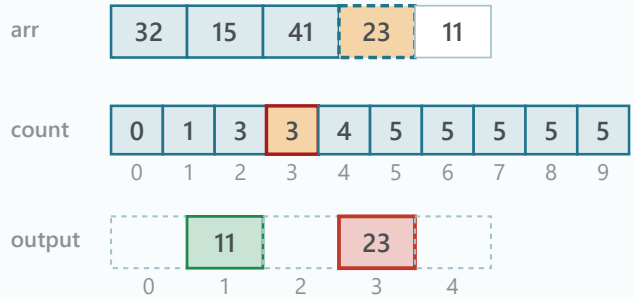
Step 8 / 13

VARIABLES

`exp = 1` `n = 5` `i = 4` `x = 11` `b = 1`

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
▶    for (int i = n - 1; i >= 0; i--) {
▶      int b = (arr[i] / exp) % 10;
▶      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



i = 3: x = 23, b = 3. `--count[3]` = 3 → `output[3] = 23`.

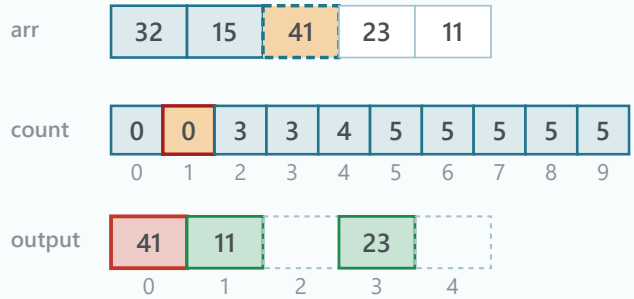
Step 9 / 13

VARIABLES

`exp = 1` `n = 5` `i = 3` `x = 23` `b = 3`

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
▶    for (int i = n - 1; i >= 0; i--) {
▶      int b = (arr[i] / exp) % 10;
▶      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



i = 2: x = 41, b = 1. `--count[1]` = 0 → `output[0] = 41`. 41 and 11 share a digit; scanning right to left puts 41 (earlier in the input) to the left of 11: **stable**.

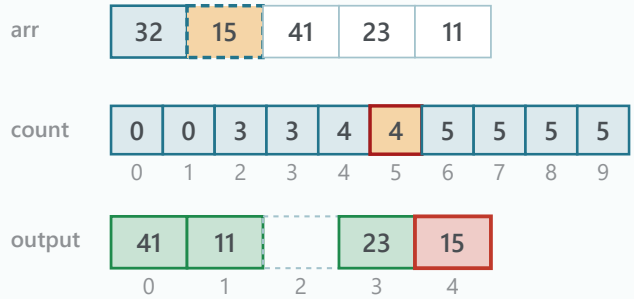
Step 10 / 13

VARIABLES

`exp = 1` `n = 5` `i = 2` `x = 41` `b = 1`

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
▶    for (int i = n - 1; i >= 0; i--) {
▶      int b = (arr[i] / exp) % 10;
▶      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



i = 1: x = 15, b = 5. `--count[5]` = 4 → `output[4] = 15`.

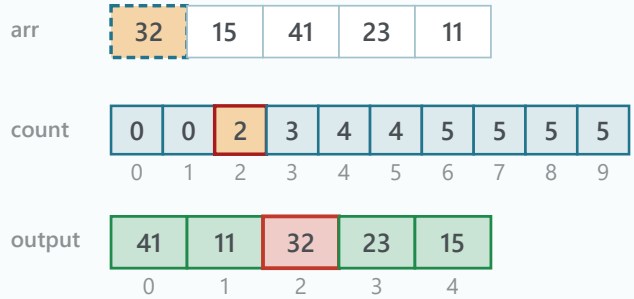
Step 11 / 13

VARIABLES

`exp = 1` `n = 5` `i = 1` `x = 15` `b = 5`

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
▶    for (int i = n - 1; i >= 0; i--) {
▶      int b = (arr[i] / exp) % 10;
▶      output[--count[b]] = arr[i];
9    }
10   return output;
11 }
```



i = 0: x = 32, b = 2. `--count[2]` = 2 → `output[2] = 32`.

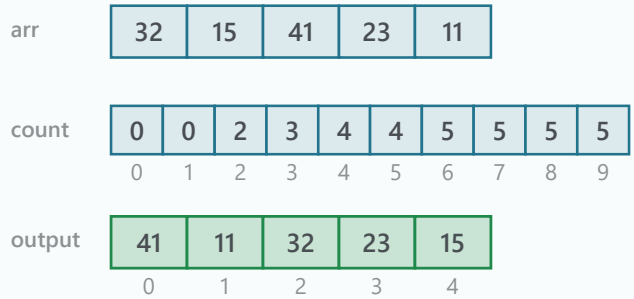
Step 12 / 13

VARIABLES

`exp = 1` `n = 5` `i = 0` `x = 32` `b = 2`

countByDigit(arr, 1): arr = [32, 15, 41, 23, 11]

```
1  int[] countByDigit(int[] arr, int exp) {
2    int n = arr.length;
3    int[] output = new int[n], count = new
   int[10];
4    for (int x : arr) count[(x / exp) % 10]++;
5    for (int i = 1; i < 10; i++) count[i] +=
   count[i - 1];
6    for (int i = n - 1; i >= 0; i--) {
7      int b = (arr[i] / exp) % 10;
8      output[--count[b]] = arr[i];
9    }
▶    return output;
11 }
```



`output` = [41, 11, 32, 23, 15]: sorted by the ones digit. The next pass, exp = 10, looks at the tens digit; stability preserves the order of the ones.

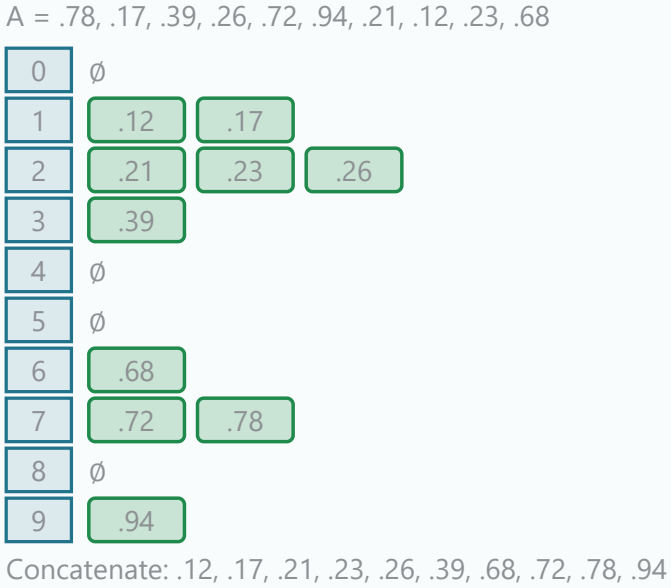
Step 13 / 13

VARIABLES

exp = 1 | n = 5 | x = – | b = –

Bucket Sort

- Suppose the input consists of numbers **uniformly distributed** over [0, 1).
- Create n buckets; a value x goes into bucket $\lfloor n \cdot x \rfloor$.
- Sort each bucket (with insertion sort) and concatenate the buckets in order.
- Expected time **Θ(n)**; worst case $\Theta(n^2)$ if every item lands in one bucket.
- The tenth algorithm in the simulator; it makes an **assumption** about the distribution.



Timsort: A Hybrid Sort

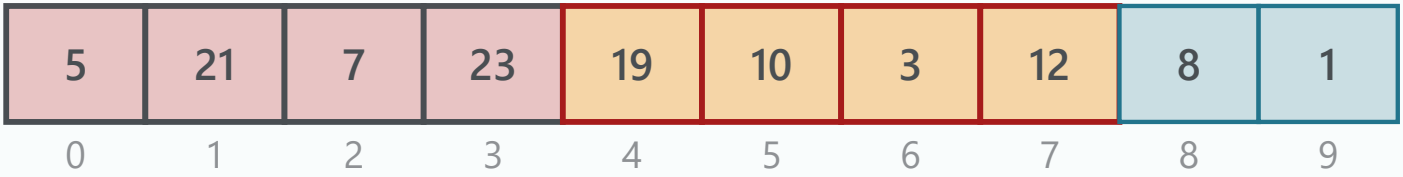
- Tim Peters (2002, Python). Java uses Timsort for object arrays (`Arrays.sort(Object[])` , `List.sort`) and Python for `sorted` .
- It exploits the **sorted stretches** (runs) common in real data: short runs are extended with **insertion** sort, and runs are combined with **merge** sort.
- Stable; O(n log n) worst case, **Θ(n)** on sorted input; O(n) extra memory.

```
static final int MIN_RUN = 4;
void timSort(int[] arr) {
  int n = arr.length;
  for (int i = 0; i < n; i += MIN_RUN)            // 1) runs
    insertionSort(arr, i, Math.min(i + MIN_RUN - 1, n - 1));
  for (int size = MIN_RUN; size < n; size *= 2)  // 2) merge
    for (int left = 0; left < n; left += 2 * size) {
      int mid = left + size - 1;
      int right = Math.min(left + 2 * size - 1, n - 1);
      if (mid < right) merge(arr, left, mid, right);
    }
}
```

This version is simplified with fixed-length runs (real Timsort uses minrun 32–64, natural run detection and "galloping").

Timsort Step by Step (MIN_RUN = 4)

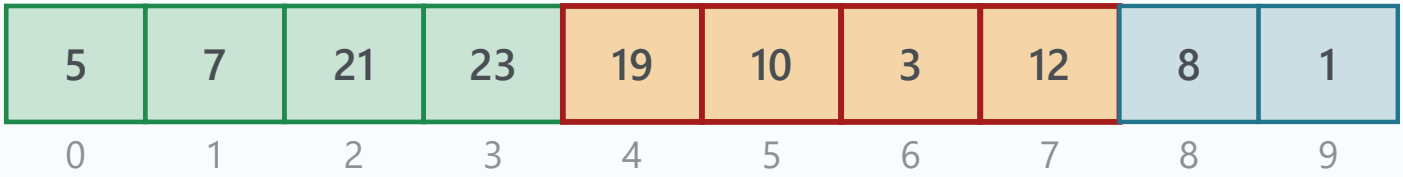
Input, MIN_RUN = 4



Array [5, 21, 7, 23, 19, 10, 3, 12, 8, 1], n = 10. The array is split into runs of 4 (colors).

Timsort Step by Step (MIN_RUN = 4)

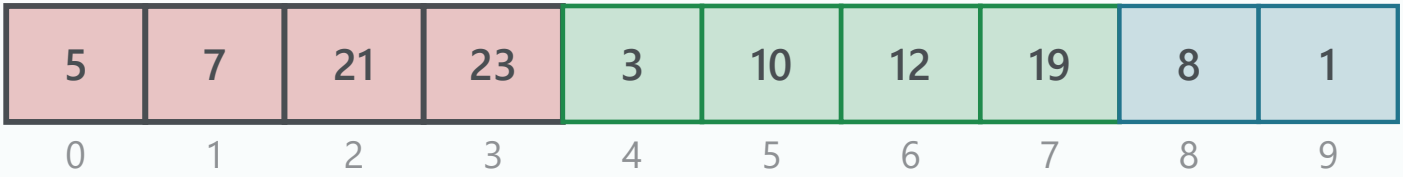
insertionSort(0, 3)



Run [0..3] sorted with insertion sort: [5, 7, 21, 23].

Timsort Step by Step (MIN_RUN = 4)

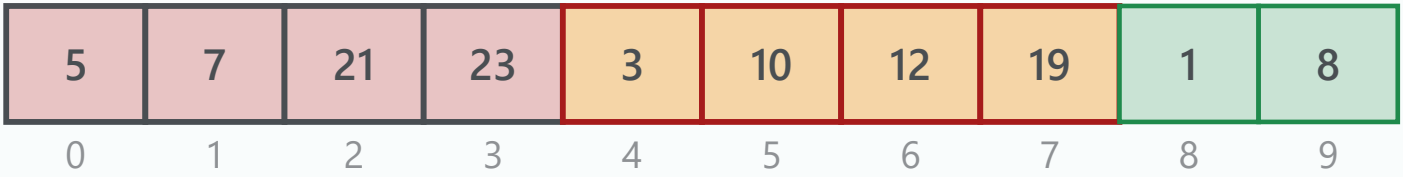
insertionSort(4, 7)



Run [4..7] sorted with insertion sort: [3, 10, 12, 19].

Timsort Step by Step (MIN_RUN = 4)

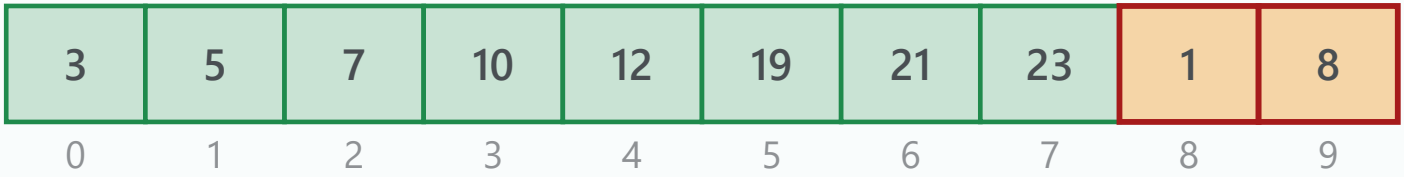
insertionSort(8, 9)



Run [8..9] sorted with insertion sort: [1, 8].

Timsort Step by Step (MIN_RUN = 4)

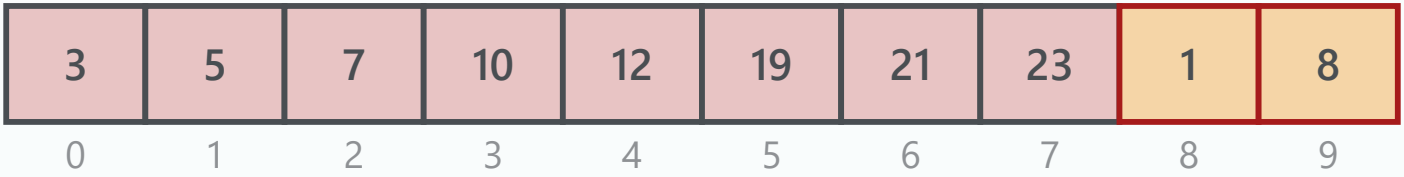
size = 4: merge(0, 3, 7)



size = 4: [5, 7, 21, 23] + [3, 10, 12, 19] merged → [3, 5, 7, 10, 12, 19, 21, 23].

Timsort Step by Step (MIN_RUN = 4)

size = 4: [8..9] alone



size = 4: run [8..9] has no partner to merge with (mid = 11 ≥ right = 9) → stays as is.

Timsort Step by Step (MIN_RUN = 4)

size = 8: merge(0, 7, 9)



size = 8: [3, 5, 7, 10, 12, 19, 21, 23] + [1, 8] merged → [1, 3, 5, 7, 8, 10, 12, 19, 21, 23]. The array is sorted.

COMPARISONS

Comparing the Sorting Algorithms

Algorithm	Best	Average	Worst	Extra memory	Stable	In place
Bubble (with flag)	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Yes	Yes
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Yes	Yes
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	No	Yes
Shell (n/2 gaps)	$\Theta(n \log n)$	depends on gaps	$\Theta(n^2)$	$O(1)$	No	Yes
Merge	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	Yes	No
Quick	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$O(\log n)$*	No	Yes
Heapsort	$O(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(1)$	No	Yes
Counting	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$	Yes	No
Radix (LSD)	$\Theta(d(n + b))$	$\Theta(d(n + b))$	$\Theta(d(n + b))$	$\Theta(n + b)$	Yes	No
Bucket	$\Theta(n)$	$\Theta(n)$†	$\Theta(n^2)$	$\Theta(n)$	Yes	No
Timsort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$	Yes	No

* recursion stack (if the smaller part is processed first). † expected, under a uniform distribution. k: key range, d: number of digits, b: base. Timsort's $\Theta(n)$ best case is for the real version with natural run detection.

How Many Comparisons? n = 1000

Algorithm	Random	Sorted	Reverse sorted
Bubble (with flag)	497,960	999	499,500
Insertion	252,205	999	499,500
Selection	499,500	499,500	499,500
Shell (n/2)	14,723	8,006	11,716
Merge	8,693	5,044	4,932
Quick (last-item pivot)	11,384	499,500	499,500
Heapsort	16,791	17,583	15,965

- Element comparisons, counted by running this chapter's code for n = 1000 (the random array was generated with a fixed seed).
- For reference: $n^2 / 2 \approx 500{,}000$, $n \log_2 n \approx 9{,}966$. On sorted input, quicksort with a last-item pivot hits its worst case.

COMPARISONS

Which Algorithm When?

Small or nearly sorted

Insertion sort: low overhead, adaptive, stable.

General purpose, speed

Quicksort (random / median-of-three pivot); cache-friendly, in place.

Stability required

Merge sort or Timsort (libraries' object sort).

Worst-case guarantee, little memory

Heapsort: $O(n \log n)$ and $O(1)$ extra memory (introsort's fallback).

Small integer keys

Counting or radix sort: $\Theta(n + k)$, $\Theta(d(n + b))$.

Linked lists, external memory

Merge sort: needs no random access; large files on disk are merged piece by piece.

Bubble and selection sort are for teaching; selection sort only makes sense when writes are very expensive.

Summary

Topic	Key points
Criteria	time (best/average/worst), extra memory, stability, in place, adaptivity
Bubble	adjacent swaps; largest at the end after each pass; best $\Theta(n)$ with a flag, otherwise $\Theta(n^2)$; stable
Insertion	save the key, shift larger items, insert; $\Theta(n + \text{inversions})$; best on small or nearly sorted data
Selection	find the minimum, move it to the front; always $n(n-1)/2$ comparisons, $\le n - 1$ swaps; unstable
Shell	insertion sort with shrinking gaps; in place, unstable
Merge	split–sort–merge; $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$; $\Theta(n)$ extra memory; stable
Quick	partition around a pivot; $\Theta(n \log n)$ on average, $\Theta(n^2)$ worst case; in place, unstable
Heapsort	build a max-heap in $O(n)$, $n - 1$ times move the root to the end and sift down; $O(n \log n)$, $O(1)$ extra memory
Linear	comparison lower bound $\Omega(n \log n)$; counting $\Theta(n + k)$, radix $\Theta(d(n + b))$, bucket $\Theta(n)$ expected

SUMMARY

Common Mistakes

Running off the array

In bubble sort the inner loop must be `j < n - 1 - i`; writing `j < n` overflows at `arr[j+1]`.

Not saving the key

In insertion sort, if `key` is not saved first, the first shift overwrites it.

Tracking the minimum by value

In selection sort, without the index the swap cannot be done.

Breaking stability

Using `<` instead of `<=` in merge, or scanning front to back in counting sort, reorders equal keys.

Including the pivot in the recursion

Writing `quickSort(arr, lo, p)` processes the pivot again; with two items left it can recurse forever.

"Quicksort is always n log n"

On sorted input a fixed pivot gives $\Theta(n^2)$ and recursion n deep (StackOverflowError).

SUMMARY

Sample Problems

1. Apply bubble sort with a flag to [5, 2, 4, 6, 1, 3]. Write the array after each pass.
2. How many comparisons and shifts does insertion sort need for the same array? How many swaps does selection sort make?
3. What are the array and the return value after `partition` (Lomuto, pivot = last item) runs on [2, 8, 7, 1, 3, 5, 6, 4]?
4. What max-heap does bottom-up construction build from [4, 10, 3, 5, 1]?
5. For A = [2, 5, 3, 0, 2, 3, 0, 3] (keys 0 … 5), what is the array C after the third loop of counting sort?
6. At most how many comparisons does merge sort need to sort 8 items?

▶ **Show answers**

SUMMARY

Exercises

1. How many comparisons and swaps does bubble sort make on a reverse-sorted array of n = 6 items?
2. Give a three-item example showing that selection sort is unstable.
3. If insertion sort finds the correct position by **binary search**, what is its time complexity?
4. What happens when quicksort (last-item pivot) is called on an already sorted array of 1000 items? How can it be fixed?
5. Sort [329, 457, 657, 839, 436, 720, 355] with LSD radix sort; write the array after each pass.
6. Propose an O(n log n) method for counting the **inversions** in an array.

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapters 2, 6, 7, 8).
- Sedgewick and Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Chapter 2: Sorting).
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., 1998.
- E. D. Demaine and C. E. Leiserson, MIT 6.046J *Introduction to Algorithms* lecture notes, 2005 (partitioning and counting sort examples).
- C. A. R. Hoare, "Quicksort", *The Computer Journal* 5(1), 1962; D. L. Shell, "A High-Speed Sorting Procedure", *CACM* 2(7), 1959.
- T. Peters, "listsort.txt" (Timsort design notes), CPython, 2002.

The End

Chapter 3: Sorting Algorithms

Previous: Chapter 2: Algorithm Complexity · Next: Chapter 4: Searching Algorithms