哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

数据结构
Data Structures

Chapter 13 Sorting Algorithm

Prof. Yitian Shao
School of Computer Science and Technology

Sorting Algorithm

Course Overview

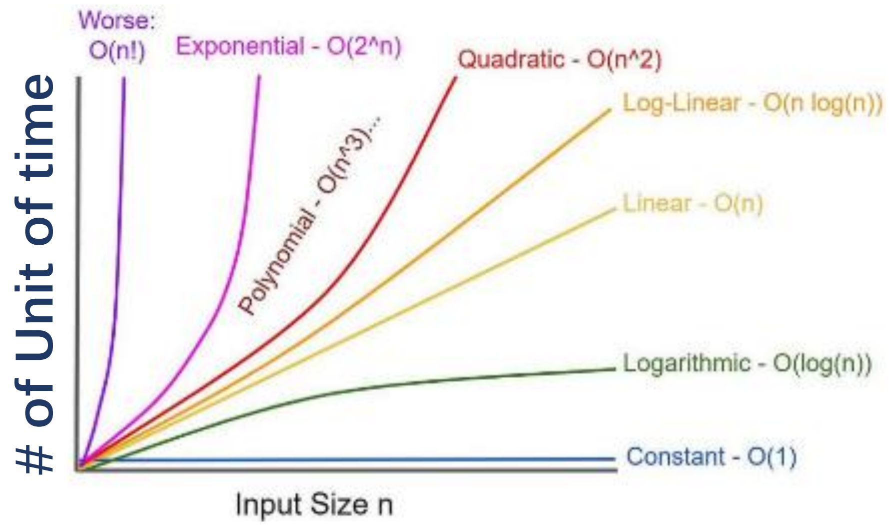
- Advanced sorting algorithms
  - Merge sort
  - Quick sort
  - Heap sort
  - Analyze time complexity

Previously: Introduction of Sorting

- One of the most common applications in computer science is **sorting**, which is the process by which **data is arranged according to its values**.
- For the beginning, we introduce three sorting algorithms: **selection sort**, **bubble sort** and **insertion sort**.
- These three sorting algorithms are the foundation of faster sorting algorithms used in computer science today.

Previously: Basic Sorting Algorithms

Algorithm	Time Complexity (Worst)	Space Complexity (Worst)
Selection Sort	$O(n^2)$	$O(1)$
Bubble Sort	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(1)$
Heap Sort (Week 12)	$O(n \log_2(n))$	$O(1)$
Quick Sort (Week 12)	$O(n^2)$	$O(n)$
Merge Sort (Week 12)	$O(n \log_2(n))$	$O(n)$



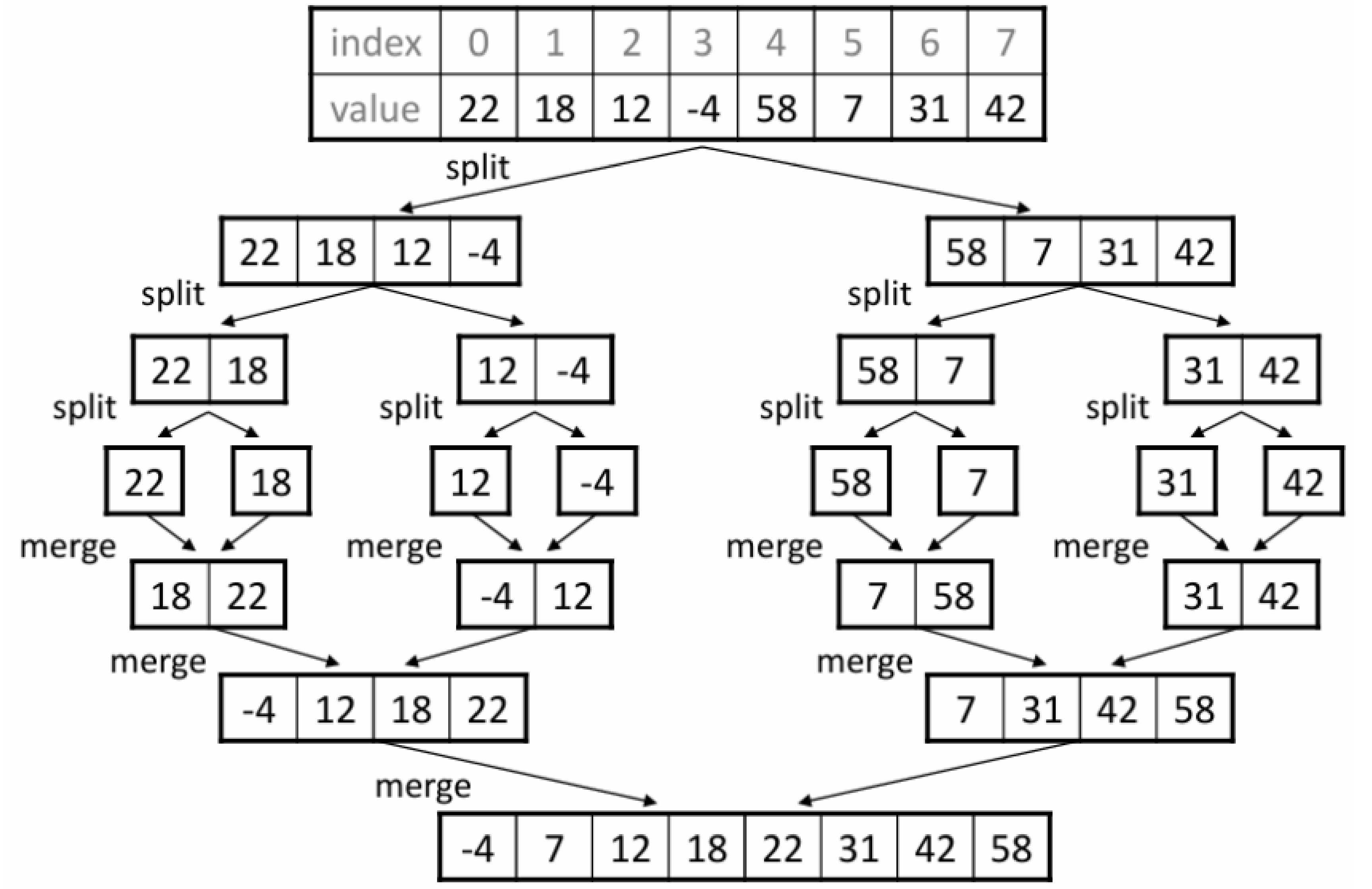
Previously: Basic Sorting Algorithms

- Divide a list into **sorted** and **unsorted sublist**, separated by a moving “wall”
- **Selection Sort**: Find the **smallest element** from the **unsorted sublist** and swap it with the element at the beginning of the **unsorted sublist**
- **Bubble Sort**: “Bubble up” (exchange with the neighbor) the **smallest element** from the **unsorted sublist**, moving toward the **sorted sublist**
- **Insertion Sort**: Pick one element from the **unsorted sublist** and insert it into the proper place of the **sorted sublist**

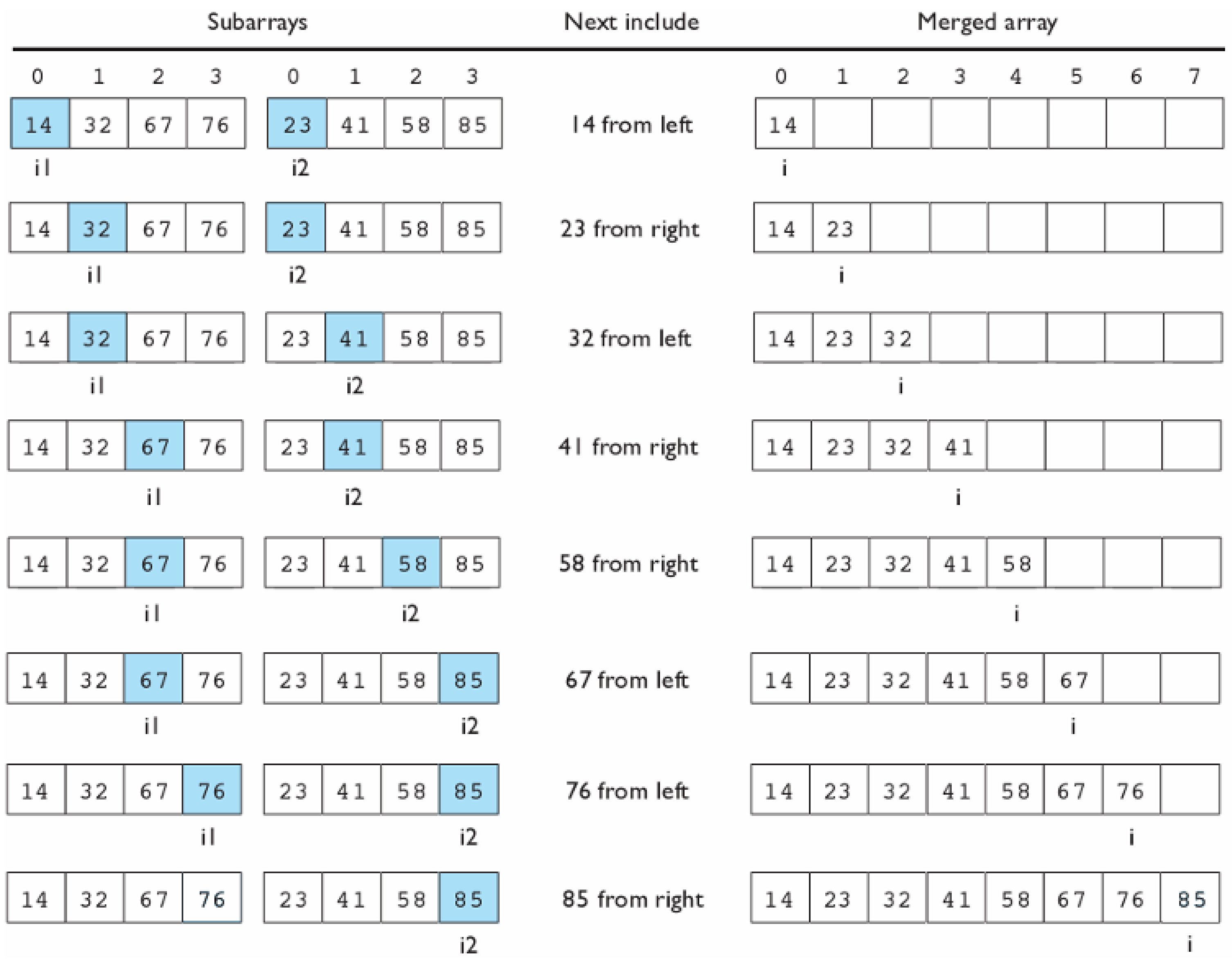
Merge Sort

- Merge sort: **Repeatedly** divides the data in half, **sorts each half**, and combines the sorted halves into a sorted whole. The procedures:
    - 1) Divide the list into two (roughly) equal halves.
    - 2) Sort the left half.
    - 3) Sort the right half.
    - 4) Merge the two sorted halves into one sorted list.
- Often implemented recursively. An example of a "divide and conquer" algorithm (Invented by John von Neumann in 1945).
- Runtime: $O(N\log_2 N)$. Somewhat faster for ascending/descending input.

Merge Sort Example



Merging Sorted Halves



Merge Sort Code

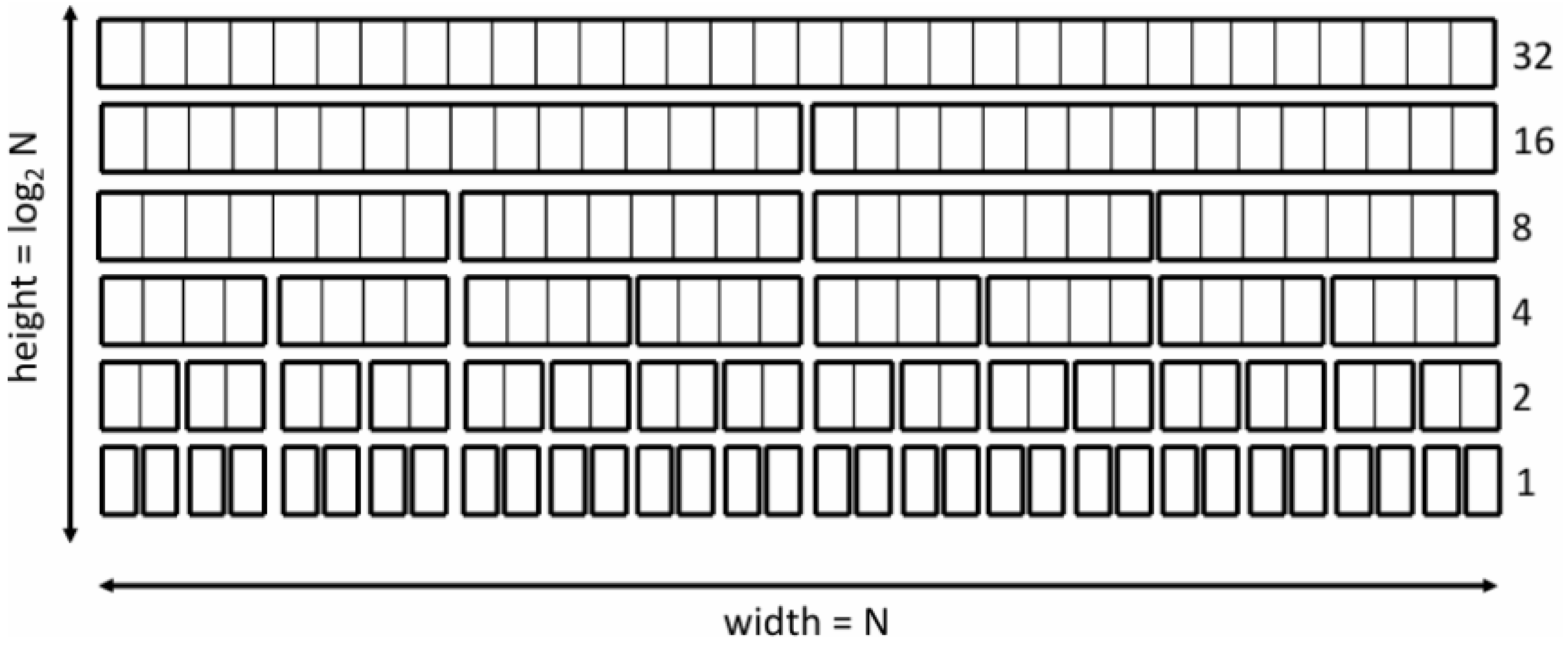
```
// Rearranges the elements of v into sorted order using merge sort
void mergeSort(vector<int>& v) {
    if (v.size() >= 2) {
       // split vector into two halves (by initializing two new vectors)
      vector<int> left(v.begin(), v.begin() + v.size()/2);
      vector<int> right(v.begin() + v.size()/2, v.end());
       // recursively sort the two halves
       mergeSort(left);
       mergeSort(right);
       // merge the sorted halves into a sorted whole
       v.clear();
       merge(v, left, right);
    }
}
```

Merge Sort Code

```
// Merges the left and right sorted array into a sorted result.
void merge(vector<int>& result, vector<int>& left, vector<int>& right) {
    int leftIndex = 0;
    int rightIndex = 0;
    for (int i = 0; i < left.size() + right.size(); i++) {
        if (rightIndex >= right.size() ||
        (leftIndex < left.size() && left[leftIndex] <= right[rightIndex])){
                result.push_back(left[leftIndex]); // take from left array
                leftIndex++;
        } else {
                result.push_back(right[rightIndex]); // take from right array
                rightIndex++;
        }
    }
}
```

Runtime Intuition

- Merge sort performs **N** operations on each level. (width)
- Each level splits the data in 2, so there are **$\log_2$N** levels. (height)
- Product of these = **N* log2N**= O(N$\log_2$N). (area)
- Example: If N= **32**, then performs ~ **$\log_2 32$** = **5** levels of 32 operations each:



Quick Sort

- Quick sort: Orders a list of values by **partitioning the list around one element** called a pivot, then sorting each partition.
  - invented by British computer scientist C.A.R. Hoare in 1960
- Quick sort is another divide and conquer algorithm:
  - 1) Choose one element in the list to be the pivot.
  - 2) Divide the elements so that all elements less than the pivot are to its left and all greater (or equal) are to its right.
  - 3) Conquer by applying quick sort (recursively) to both partitions.
- Runtime: $O(N\log_2 N)$ on average, but $O(N^2)$ worst case.
  - Generally somewhat faster than merge sort.

Choosing a “Pivot"

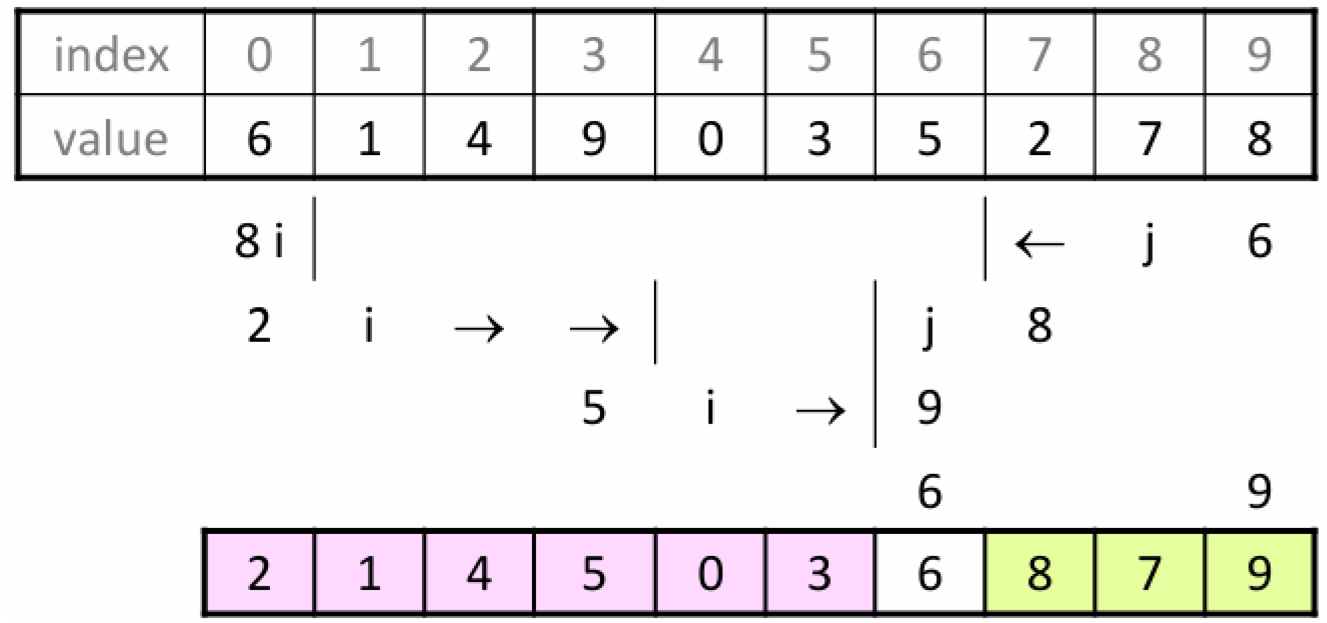
- The algorithm will work correctly no matter which element you choose as the pivot.
    - A simple implementation can just use the first element.
- But for efficiency, it is better if the pivot divides up the array into roughly equal partitions.
    - What kind of value would be a good pivot? A bad one?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

Partitioning an Array

- Swap the pivot to the last array slot, temporarily.
- Repeat until done partitioning (until i and j meet):
    - Starting from i = 0, find an element a[i] ⩾ pivot.
    - Starting from j = N-1, find an element a[j] ⩽ pivot.
    - These elements are out of order, so swap a[i] and a[j].
- Swap the pivot back to index i to place it between the partitions.

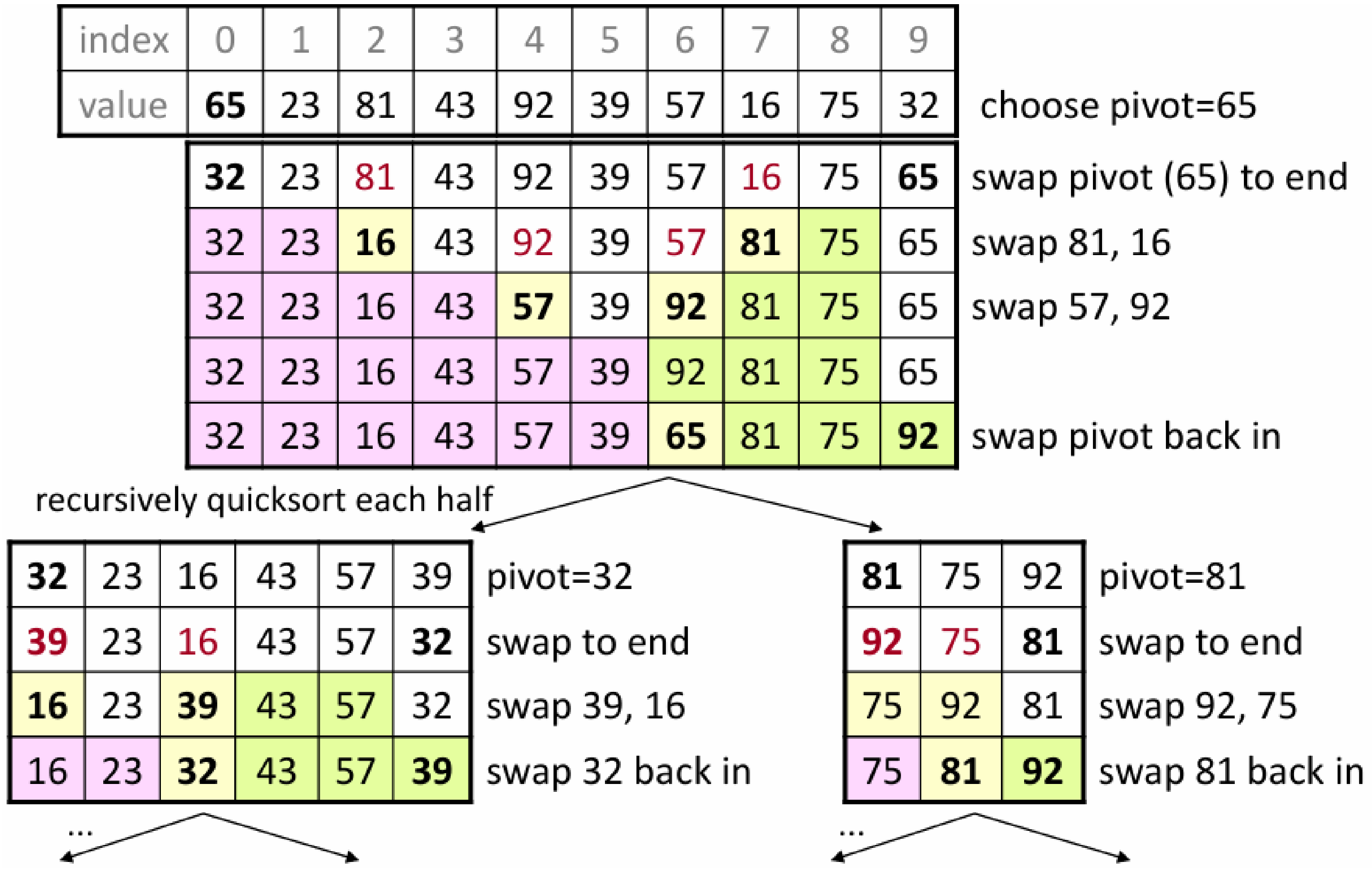
Partitioning an Array



index	0	1	2	3	4	5	6	7	8	9
value	6	1	4	9	0	3	5	2	7	8

2	1	4	5	0	3	6	8	7	9

Quick Sort Example



index	0	1	2	3	4	5	6	7	8	9	
value	**65**	23	81	43	92	39	57	16	75	32	choose pivot=65
	32	23	81	43	92	39	57	16	75	**65**	swap pivot (65) to end
	32	23	**16**	43	92	39	57	**81**	75	65	swap 81, 16
	32	23	16	43	**57**	39	**92**	81	75	65	swap 57, 92
	32	23	16	43	57	39	92	81	75	65	
	32	23	16	43	57	39	**65**	81	75	**92**	swap pivot back in

recursively quicksort each half

32	23	16	43	57	39	pivot=32
39	23	16	43	57	**32**	swap to end
16	23	**39**	43	57	32	swap 39, 16
16	23	**32**	43	57	**39**	swap 32 back in

...

81	75	92	pivot=81
92	75	**81**	swap to end
75	92	81	swap 92, 75
75	**81**	**92**	swap 81 back in

...

Quick Sort Code

```
void quickSort(vector<int>& v) {
      quickSortHelper(v, 0, v.size() - 1);
}

void quickSortHelper(vector<int>& v, int min, int max) {
   if (min >= max) {return;}  // end of recursion; no sorting
   int pivot = v[min]; // choose pivot as the first element (might be bad)
   swap(v[min], v[max]);   // move pivot to end
   int middle = partition(v, min, max - 1, pivot); // partition two sides
   swap(v[middle], v[max]);   // restore pivot to proper location
   // recursively sort the left and right partitions
   quickSortHelper(v, min, middle - 1);
   quickSortHelper(v, middle + 1, max);
}
```

Quick Sort Code

```
// Partitions a with elements < pivot on left and elements > pivot on right;
// returns index of element that should be swapped with pivot
int partition(vector<int>& v, int i, int j, int pivot) {
    while (i <= j) {
        // move index i,j toward center until find an unsorted pair
        while (i <= j && v[i] < pivot) { i++; }
        while (i <= j && v[j] > pivot) { j--; }
        if (i <= j) {
                swap(v[i], v[j]);
                i++;
                j--;
        }
    }
    return i;
}
```

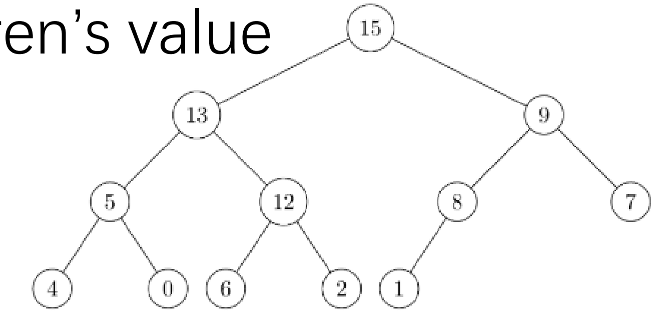
Choosing a Better Pivot

- Choosing the first element as the pivot leads to very poor performance on certain inputs (ascending, descending): Not partition the array into **roughly-equal size chunks**!
- Alternative methods of picking a pivot:
  - **Random**: Pick a random index from [min .. max]
  - **Median-of-3**: look at left/middle/right elements and pick the one with the medium value of the three:
    - v[min], v[(max+min)/2], and v[max]
    - Better performance than picking random numbers every time
    - Provides near-optimal runtime for almost all input orderings

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	91	-4	27	30	86	50	65	78	5	56	2	25	42	98	**31**

Heap Sort

- **Recall**:
  - Heap, a complete binary tree with max/min heap property
  - Max Heap Property: parent node value ⩾ the children's value



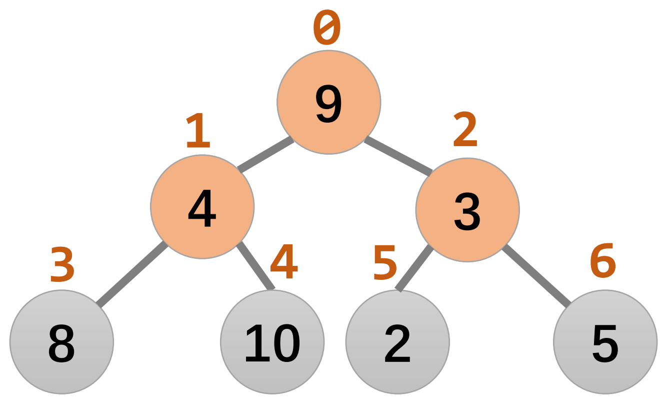
  - Heap dequeue
    - Remove the root, and replace it with the furthest-right bottom node
    - To restore heap order, the improper root is shifted ("bubbled") down the tree by swapping with its larger child [max-heap].

Heap Sort

- Sorting Strategy:
  - 1) **Visualize the array as a complete binary tree**: For an array of size n, the root is at index 0, the left child of an element at index i is at 2i + 1, and the right child is at 2i + 2.
  - 2) **Build a max heap**: Compare each node with its children, ensuring parent nodes are larger, causing smaller nodes to “bubble down”.
  - 3) Sort the array by **moving the largest element at the end of unsorted array**.

Heap Sort

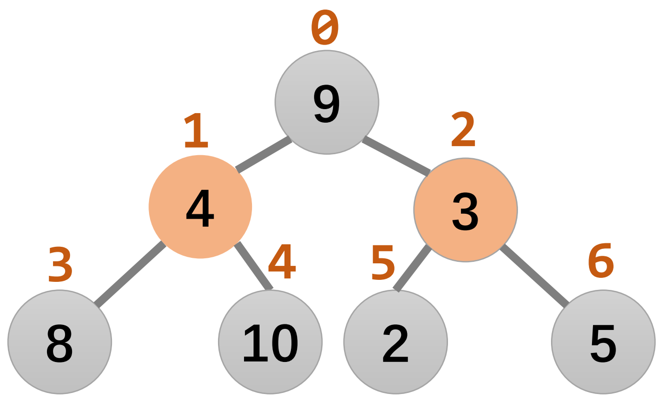
- 1) Visualize **`arr`** as a **complete binary tree**: For an array of size n, the **root** is at index **`0`**, the **left** child of an element at index **`i`** is at **`2i + 1`**, and the **right** child is at **`2i + 2`**



index	0	1	2	3	4	5	6
arr	9	4	3	8	10	2	5

Heap Sort

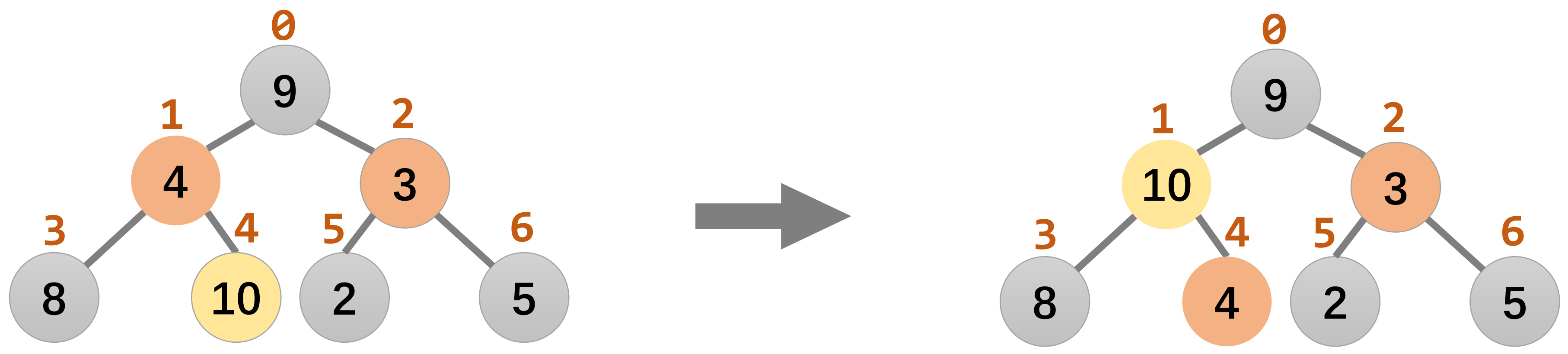
- 2) Build a max heap: Compare each node with its children, ensuring parent nodes are larger, causing smaller nodes to “bubble down”.



index	0	1	2	3	4	5	6
arr	9	4	3	8	10	2	5

Heap Sort

- 2) Build a max heap: Compare each node with its children, ensuring parent nodes are larger, causing smaller nodes to “bubble down”.

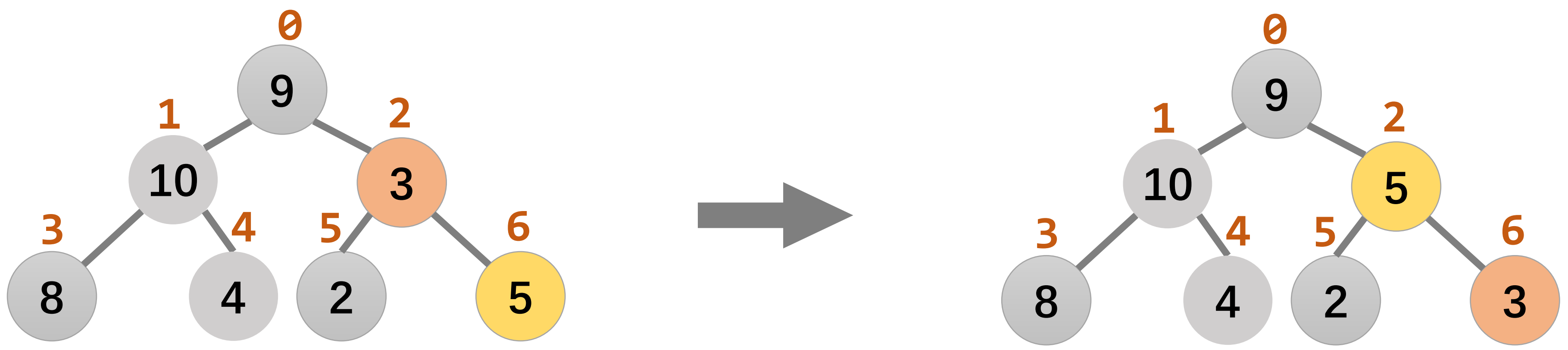


index	0	1	2	3	4	5	6
arr	9	4	3	8	10	2	5

index	0	1	2	3	4	5	6
arr	9	10	3	8	4	2	5

Heap Sort

- 2) Build a max heap: Compare each node with its children, ensuring parent nodes are larger, causing smaller nodes to “bubble down”.

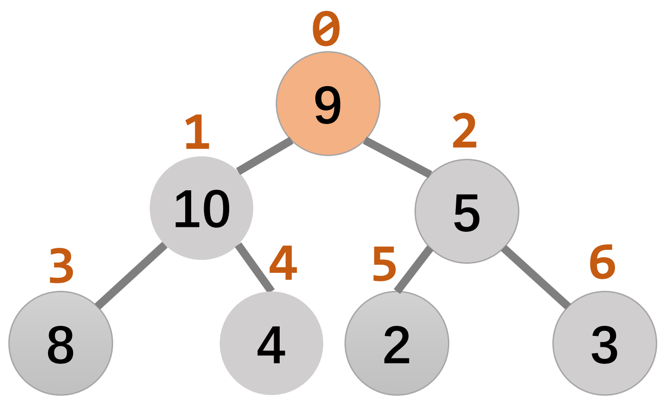


index	0	1	2	3	4	5	6
arr	9	10	3	8	4	2	5

index	0	1	2	3	4	5	6
arr	9	10	5	8	4	2	3

Heap Sort

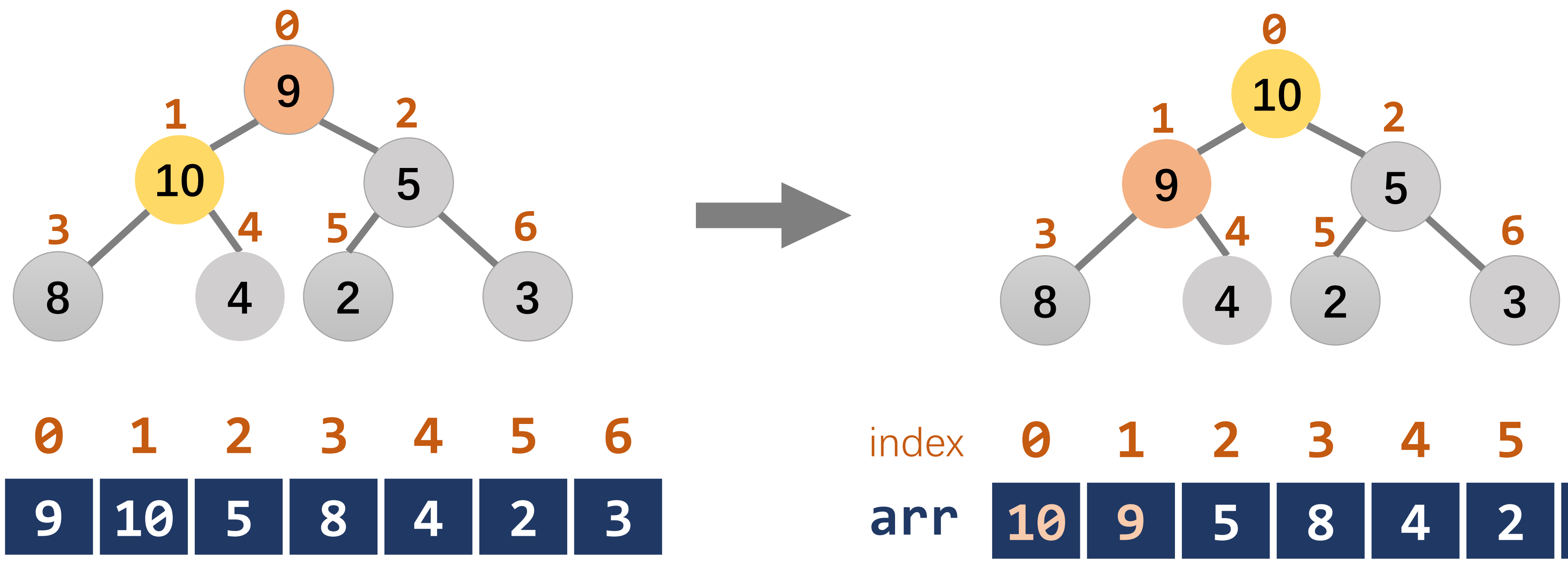
- 2) Build a max heap: Compare each node with its children, ensuring parent nodes are larger, causing smaller nodes to “bubble down”.



index	0	1	2	3	4	5	6
arr	9	10	5	8	4	2	3

Heap Sort

- 2) Build a max heap: Compare each node with its children, ensuring parent nodes are larger, causing smaller nodes to “bubble down”.

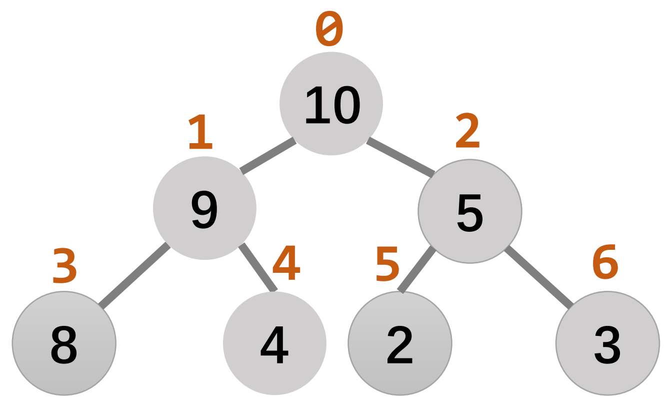


index	0	1	2	3	4	5	6
arr	9	10	5	8	4	2	3

index	0	1	2	3	4	5	6
arr	10	9	5	8	4	2	3

Heap Sort

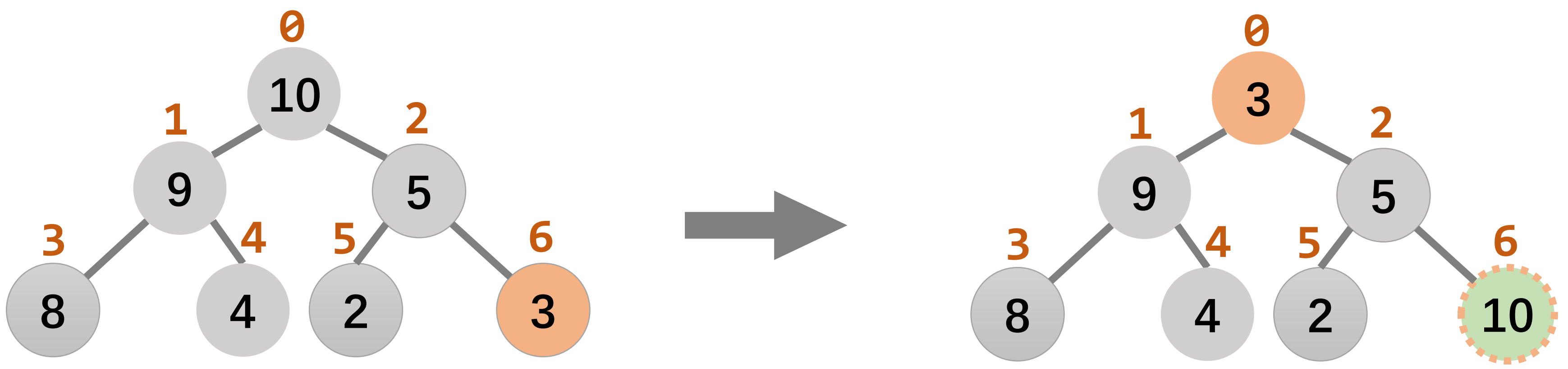
- 3) Sort the array by moving the largest element at the end of unsorted array.



index	0	1	2	3	4	5	6
arr	10	9	5	8	4	2	3

Heap Sort

- 3) Swap the largest element, 10, with last element (bottom rightmost), then decrease the size of the heap by one (ignore the last element, as it is sorted after swapping)

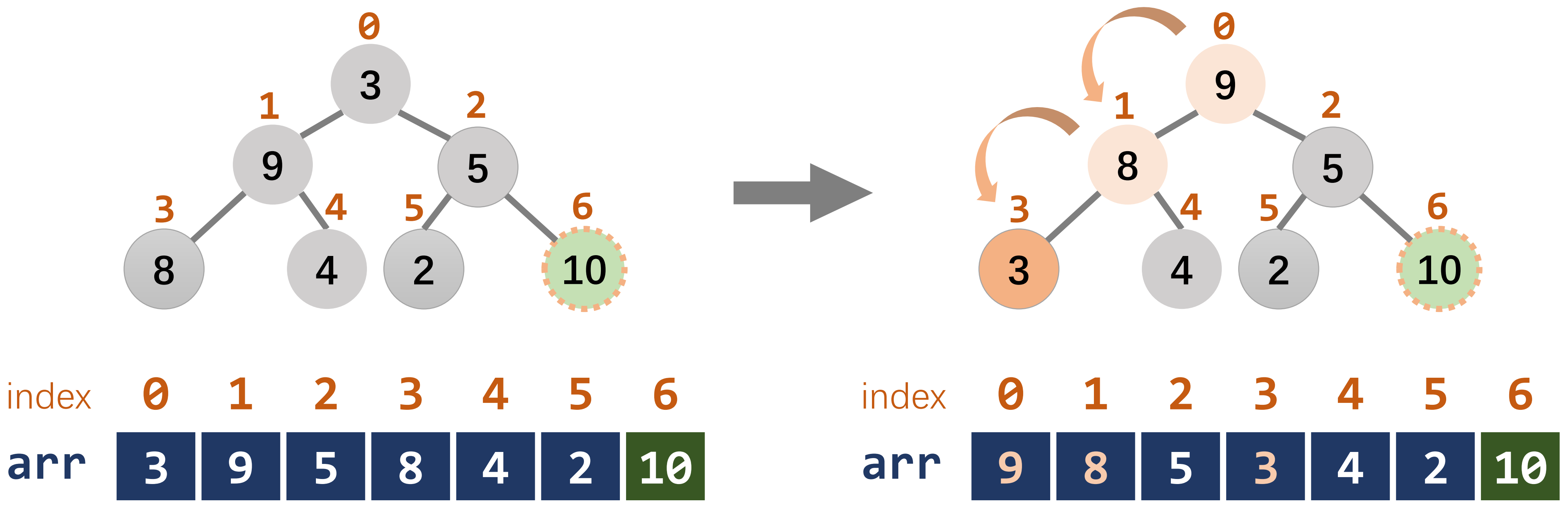


index	0	1	2	3	4	5	6
arr	10	9	5	8	4	2	3

index	0	1	2	3	4	5	6
arr	3	9	5	8	4	2	10

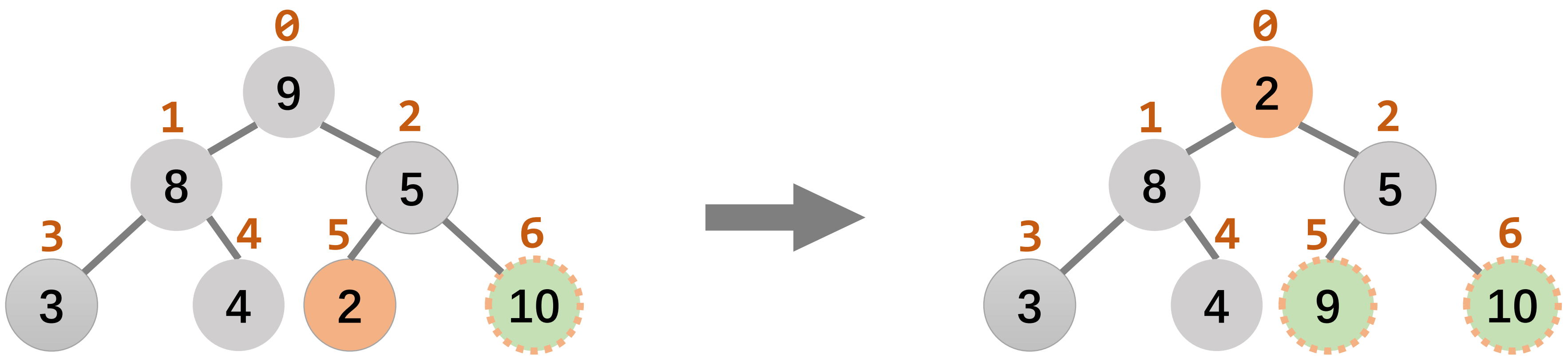
Heap Sort

- 3) Maintaining the max heap property with the root “bubble down”.



Heap Sort

- 3) Maintaining the max heap property with the root “bubble down”.

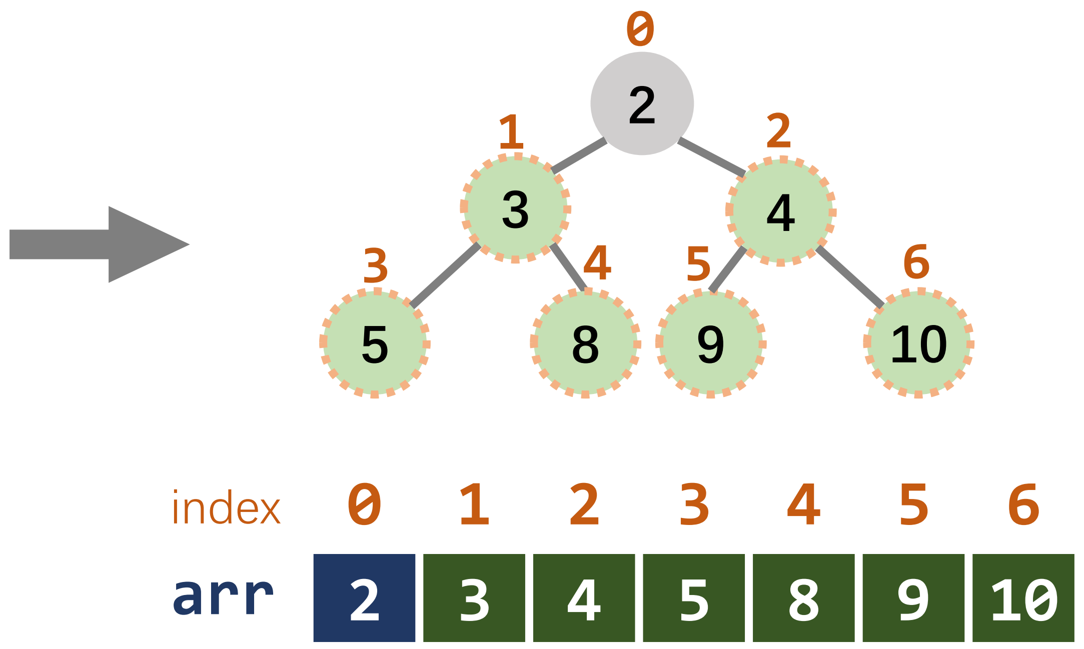


index	0	1	2	3	4	5	6
arr	9	8	5	3	4	2	10

index	0	1	2	3	4	5	6
arr	2	8	5	3	4	9	10

Heap Sort

- 3) Keep repeating these steps until there's only one element left in the heap.



Heap Sort Code

```
void heapSort(vector<int>& arr){
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--) // Build heap (rearrange vector)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move current root to end
        heapify(arr, i, 0); // Call max heapify on the reduced heap
    }
}
```

Heap Sort Code

```
void heapify(vector<int>& arr, int n, int i){
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left index = 2*i + 1
    int r = 2 * i + 2; // right index = 2*i + 2
    // If left child is larger than root (largest so far)
    if (l < n && arr[l] > arr[largest]){ largest = l; }
    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest]){ largest = r; }
    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest); // Recursively heapify the affected sub-tree
    }
}
```

Analyze Time Complexity

Algorithm	Time Complexity (Worst)	Time Complexity (Average)
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Heap Sort (Week 12)	$O(n \log_2(n))$	$O(n \log_2(n))$
Quick Sort (Week 12)	$O(n^2)$	$O(n \log_2(n))$
Merge Sort (Week 12)	$O(n \log_2(n))$	$O(n \log_2(n))$

IMPORTANT! MUST READ CAREFULLY!

Announcement of Exam Policy

Exam Policy: 8 Strictly Prohibited Actions

Prohibited Actions During Exam:

1. Do **not** use/hide unauthorized materials (e.g., cheat sheets, notes on hands, electronics); all items must be inside your bag and monitored by proctors
2. Do **not** borrow items (e.g., calculators, stationery) from classmates during the exam; instead, ask proctor for help
3. Do **not** look at another student’s papers/answers
4. Do **not** communicate with others (verbally or via gestures) during the exam.
5. Do **not** cover your mouth during the exam. If you have to wear a mask, <u>sit in the first row</u>
6. Do **not** leave the exam room without the proctor's permission
7. Do **not** spend more than **4 minutes** on restroom breaks
8. Do **not** take electronics outside the exam room

Any violation of the rules will result in immediate expulsion from the exam and will be deemed as cheating the exam!

Exam Policy: 8 Rules for Pausing/Ending Exam

Exam Pause Protocol (Restroom Break):

9. Raise hand and **must wait for permission** before leaving the exam room
10. Only **one person** may pause and leave the exam room at any given time
11. You must be escorted by a proctor to the restroom
12. Do **not** talk to anyone during the break
13. Do **not** use any electronics during the break

Exam End Protocol

14. If you finish early or when the exam ends, stay seated; do **not** talk, look around, or interact with others until all exam sheets are collected
15. Do **not** take any exam materials outside the exam room, including draft papers
16. Stop writing immediately when the proctor ends the exam

Any violation of the rules will result in immediate expulsion from the exam and will be deemed as cheating the exam!

6 Ways to Uphold Academic Integrity

Take Action to Uphold Academic Integrity:

1. During the exam, only ask the proctors for any assistance (e.g., borrow pens)
2. Keep your eyes only on your own exam sheets
3. Ignore anyone who tries to communicate with you; report to the proctors immediately if someone distracts you (Do not confront them directly)
4. Protect your answers from being copied by others; never leave your exam materials unattended
5. No post-exam discussion until all students have submitted their exam sheets; stop anyone who violates this rule
6. If you suspect cheating, report the misconduct along with the ID/seat location/name to the leading proctor after the exam

Self-Test: Ensure the Exam Policies Is Clear to You

	Scenario	Consequence	Proper Conduct
1	You forgot to bring a pen / eraser / calculator with you but need to use it; you decided to borrow it from other students.	**NO! This is deemed as cheating!**	**Raise your hand** and wait until a proctor borrows it for you.
2	You feel bored during the exam and you chat with others.	**NO! This is deemed as cheating!**	Submit your exam and **leave the exam room**.
3	You finished your exam early. You meet your friend who pause the exam for the restroom (toilet) and chat with them or borrow your electronics.	**NO! This is deemed as cheating!**	**No interaction** is allowed with those who have not submitted the exam.
4	Talk to yourself, sing/make sound during the exam	**NO! This may be deemed as cheating!**	**No mouth movement**
5	The exam time is over (announced by the proctor). You realize that you forget to write down your name/ID and you rush to do it	**NO! This is deemed as cheating!**	**Stop writing immediately** and quietly wait until all exam sheets are collected. Ask proctors to mark your name and ID

Self-Test: Ensure the Exam Policies Is Clear to You

	Scenario	Consequence	Proper Conduct
6	The exam time is over (announced by the proctor). You stand up and put away your pens, you start chatting with your friends while waiting for the exam sheets to be collected	**NO! This is deemed as cheating!**	**Quietly wait** until all exam sheets are collected. Or both of you can leave the exam room and chat.
7	Take your exam sheets or draft papers outside the exam room, even with some blank draft papers	**NO! This is deemed as cheating!**	**Return all** exam sheets and draft papers to the proctor
8	Fill in the exam sheets for others	**NO! This is deemed as cheating!**	**Do not help others** in any situation, the proctors will help!
9	Accidentally/inadvertently look at other people's exam sheets during the test	**NO! This is deemed as cheating!**	**Focus on your own sheets** for the entire exam period

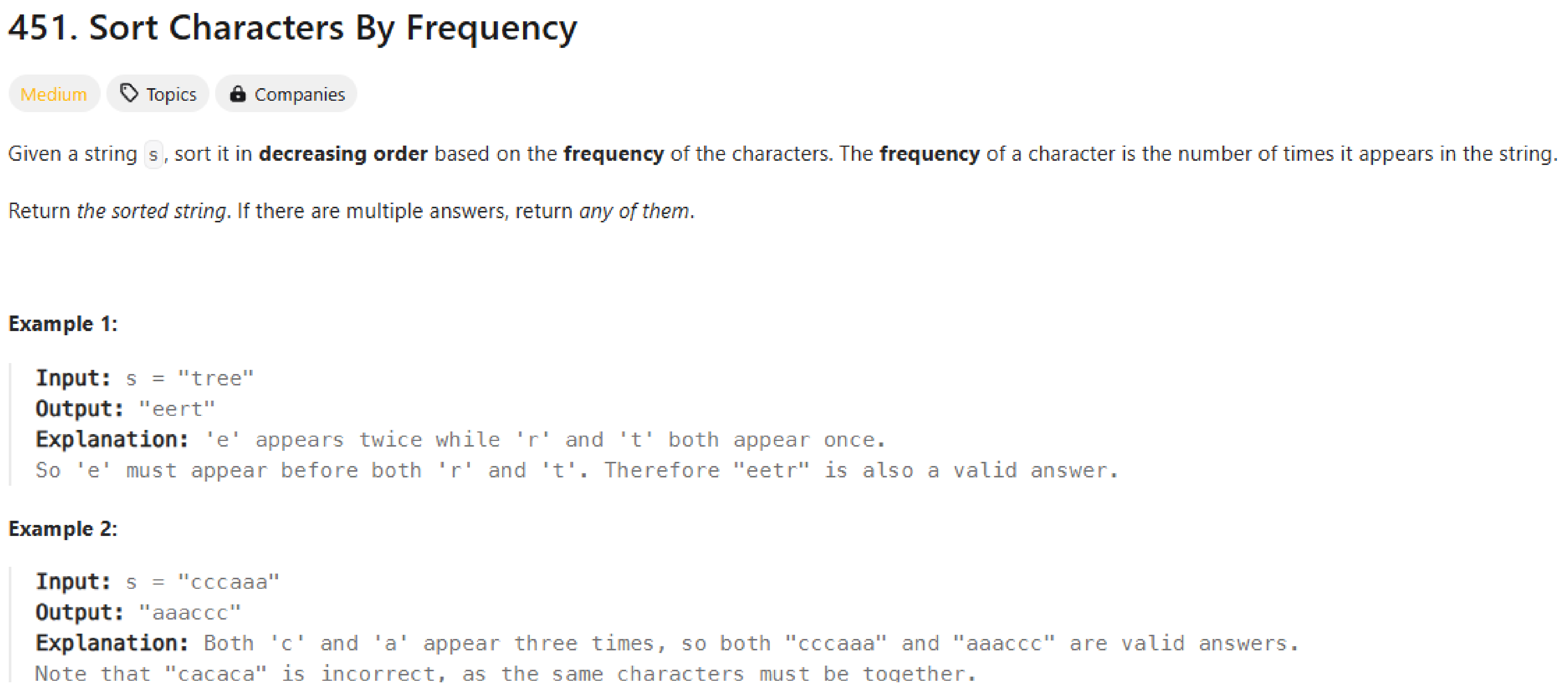
Self-Test: Ensure the Exam Policies Is Clear to You

	Scenario	Proper Conduct
1	You need to go to the restroom during exam	1. Raise your hand and wait for the proctor’s approval 2. Don’t talk to/interact with anyone during break 3. Come back to the exam in 4 minutes
2	Someone sitting next to you asked you for help during the exam	Do not respond, or both of you will be caught. Ignore them and stay focused on your own sheets. If the person keeps bothering you, raise hand and report to the proctor
3	If you get sick and have to wear a mask	Sit in the front row (first row) assigned by the leading proctor

Exercise 4.1

- Complete LeetCode 451

Try solving it using hash map and heap



451. Sort Characters By Frequency

Medium Topics Companies

Given a string `s`, sort it in **decreasing order** based on the **frequency** of the characters. The **frequency** of a character is the number of times it appears in the string.

Return *the sorted string*. If there are multiple answers, return *any of them*.

Example 1:

```
Input: s = "tree"
Output: "eert"
Explanation: 'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.
```

Example 2:

```
Input: s = "cccaaa"
Output: "aaaccc"
Explanation: Both 'c' and 'a' appear three times, so both "cccaaa" and "aaaccc" are valid answers.
Note that "cacaca" is incorrect, as the same characters must be together.
```

Exercise 4.2

- Try implementing **merge sort**, **quick sort**, and **heap sort** all by yourself **without any hint**!
- Write a testing code to validate your implementation, the code should generate at least three arrays containing unsorted numbers and then check whether your implementations are successful or not. Moreover, try analyzing the time complexity by yourself.