KAZAKH BRITISH TECHNICAL UNIVERSITY

LECTURE 6 & 7 – HEAP & QUICK, & MERGE SORTS

Prepared by: Izbassar Assylzhan

Course: Algorithms & Data Structures – Fall 2025

LECTURE 6 & 7

In this lecture, we'll explore three major efficient sorting algorithms: **Heap**, **Quick**, and **Merge** Sort. We will focus on the core ideas behind each algorithm, understand how they work step by step, analyze their time and space complexities, and discuss when to use each one.

TOPICS WE'LL COVER:

- Introduction to Efficient Sorting
- Heap Sort
- Quick Sort
- Merge Sort
- Practice Problems

GOALS FOR THIS LECTURE:

- Explain the core concepts behind each sorting, including how each algorithm works.
- Implement these sorting algorithms using recursion, heap operations, and merging techniques.
- Analyze and compare their time and space complexities, stability, and use cases.

INTRODUCTION TO EFFICIENT SORTING

KAZAKH BRITISH TECHNICAL UNIVERSITY

- **Introduction to Efficient Sorting**
- Heap Sort
- Quick Sort
- Merge Sort
- Practice Problems

Sorting is one of the most fundamental problems in computer science. While simple algorithms like **Bubble Sort**, **Selection Sort**, and Insertion Sort are easy to understand, they all have a time complexity of **$O(n^2)$**, which becomes inefficient for large datasets.

To handle bigger inputs efficiently, we use more advanced sorting algorithms with O(n log n) performance.

We focus on three of the most important ones:

- Heap Sort - O(n log(n));
- Quick Sort - O(n log(n));
- Merge Sort - O(n log(n)).

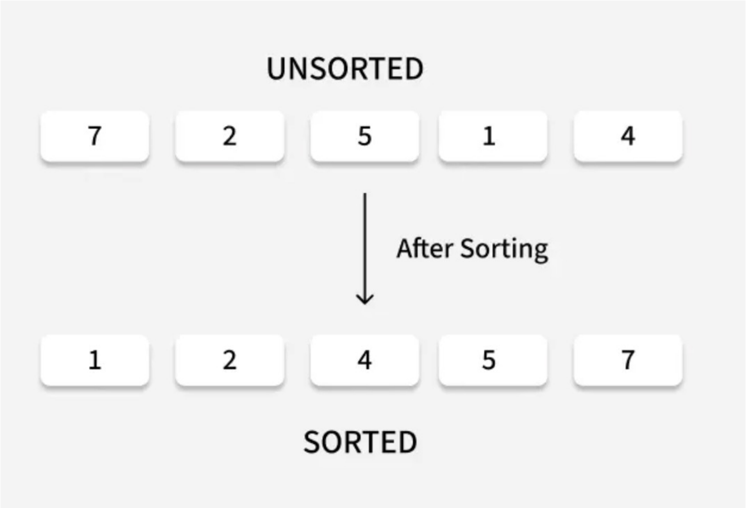

Figure 1 – Unsorted & Sorted arrays [1]

[1] https://www.geeksforgeeks.org/dsa/introduction-to-sorting-algorithm/

HEAP SORT

Heap Sort is a comparison-based sorting algorithm that works using a **Binary Heap** data structure.

Key implementations:

1. Use a Max-Heap or Min-Heap;
2. Transform the array into a valid Max-Heap;
3. Sorting Process:
   - Swap the root (maximum) with the last element of the heap;
   - Reduce the heap size by one;
   - Call heapify-down to restore the heap property.
4. Continue until the heap size becomes 1.

```
vector<int> sort() {
    int n = heap.size();
    vector<int> heapCopy;

    for (int i=0; i<n; ++i) {
        heapCopy.push_back(heap[i]);
    }
    vector<int> sorted;
    for (int i=0; i<n; ++i) {
        sorted.push_back(this->pop_front());
    }
    heap = heapCopy;
    return sorted;
}
```

Figure 2 – Heap-Sort using heapify down

QUICK SORT

Quick Sort follows the divide-and-conquer strategy and works by selecting a pivot element and rearranging the array so that:

- All elements less than the pivot go to its left;
- All elements greater than the pivot go to its right.

Steps of QS:

1. Choose a Pivot that can be first, last, middle, or random element;
2. Rearrange elements so smaller ones come before the pivot and larger ones after;
3. Apply quick sort to left and right partitions.

Time Complexity: O(n log n)

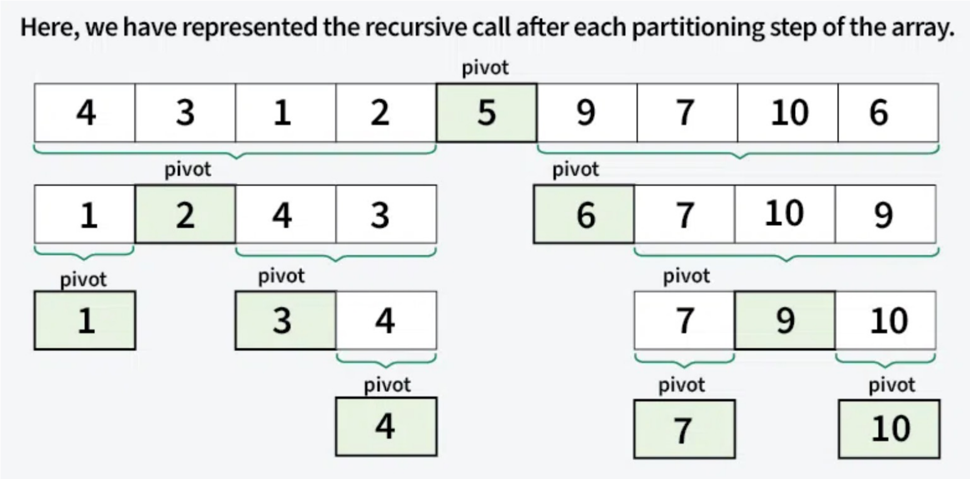

Figure 3 – Working principles of Quick Sort [2]

[2] https://www.geeksforgeeks.org/dsa/introduction-to-sorting-algorithm/

MERGE SORT

Merge Sort is a divide-and-conquer sorting algorithm that guarantees O(n log n) time in all cases.

Instead of rearranging elements in-place like Quick Sort or Heap Sort, it works by dividing the array into smaller parts and merging them back in sorted order.

Core Idea:

1. Split the array into two halves until each subarray has one element;
2. Recursively sort the left half and the right half;
3. Combine (merge) the two sorted halves into one sorted array.

Time Complexity: O(n log n)

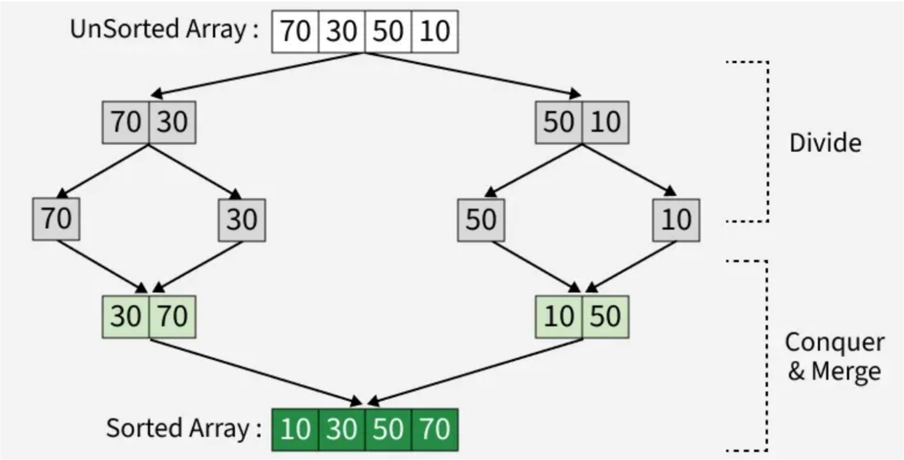

Figure 4 – Representation of working the Merge Sort [2]

[2] https://www.geeksforgeeks.org/dsa/merge-sort/

PRACTICE PROBLEMS

We have two tasks to solve:

1. https://leetcode.com/problems/assign-cookies/description/?envType=problem-list-v2&envId=sorting
2. https://leetcode.com/problems/sort-list/description/?envType=problem-list-v2&envId=merge-sort
3. https://leetcode.com/problems/sort-an-array/description/?envType=problem-list-v2&envId=merge-sort

Q & A