Data Structures

Week 13 – Sorting II: merge sort and quicksort

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Simple sorting algorithms** (week 5): bubble, selection, insertion – **$O(n^2)$**
- **Merging** (weeks 3, 6): one sorted sequence from two sorted ones – **$O(n + m)$**
- **Recursion** (week 12): reducing the problem to smaller instances

Questions for today

- How can we sort in O(n log n) time?
- What is the “divide and conquer” principle?
- Why is quicksort the favourite of practice – and when is it slow?

Divide and conquer

Divide and conquer – three steps:

1. **Divide:** the problem is split into smaller subproblems of the same type
2. **Conquer:** the subproblems are solved **recursively** (very small ones directly)
3. **Combine:** the full solution is built from the partial solutions

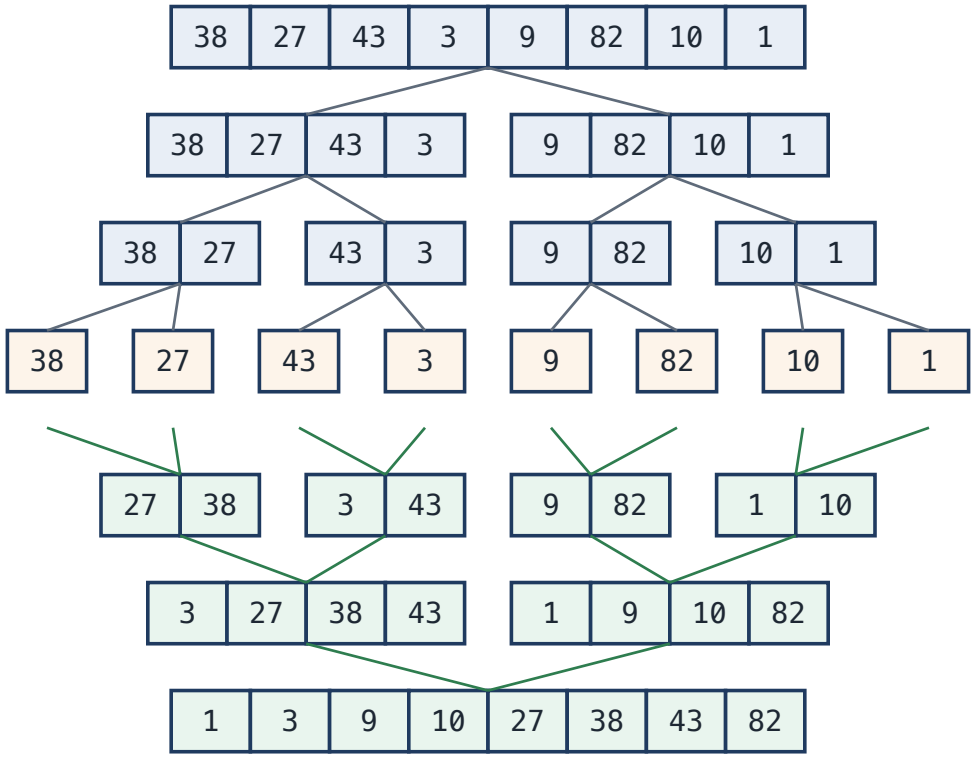
	Divide	Combine
Merge sort	trivial (cut in the middle)	**work:** merging
Quicksort	**work:** partitioning	trivial (nothing)
Binary search (week 2)	based on the middle element	– (only one half is examined)

Merge sort

Merge sort – the idea

split

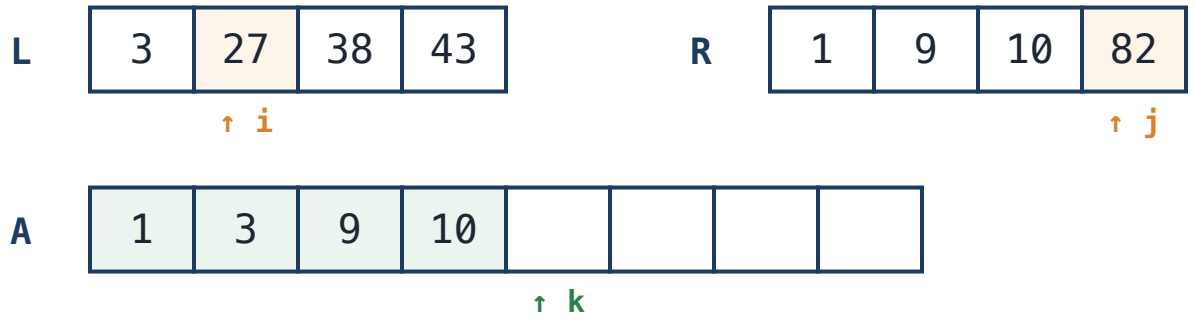
merge



$\log_2 n$ levels

O(n) work per level

The merge step



`L[i] = 27 < R[j] = 82`

`→ A[k] ← 27, i ← i + 1`

always take the smaller of
the two front elements

- One sorted array (A) from two **sorted** parts (L, R)
- We always move the smaller of the elements at the **front** of the two parts
- If one part runs out, the rest of the other is copied
- Steps: **O(n)**, extra space: **O(n)**

Merge sort – algorithm

Pseudocode

```
Procedure MergeSort(A, left, right)
  If left < right then
    k ← (left + right) div 2
    MergeSort(A, left, k)
    MergeSort(A, k + 1, right)
    Merge(A, left, k, right)
  End if
End procedure

Procedure Merge(A, left, k, right)
  L ← A[left..k] ; R ← A[k+1..right]   // copies
  i ← 1 ; j ← 1 ; m ← left
  While i ≤ length(L) and j ≤ length(R)
    If L[i] ≤ R[j] then
      A[m] ← L[i] ; i ← i + 1
    else
      A[m] ← R[j] ; j ← j + 1
    End if
    m ← m + 1
  End while
  copy the rest of L and R into A
End procedure
```

Java

```java
static void mergeSort(int[] a, int lo, int hi) {
    if (lo >= hi) return;
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, lo, mid);
    mergeSort(a, mid + 1, hi);
    merge(a, lo, mid, hi);
}

static void merge(int[] a, int lo, int mid, int hi) {
    int[] l = Arrays.copyOfRange(a, lo, mid + 1);
    int[] r = Arrays.copyOfRange(a, mid + 1, hi + 1);
    int i = 0, j = 0, k = lo;
    while (i < l.length && j < r.length)
        a[k++] = (l[i] <= r[j]) ? l[i++] : r[j++];
    while (i < l.length) a[k++] = l[i++];
    while (j < r.length) a[k++] = r[j++];
}
```

Merge sort – analysis

- Number of halvings: **$\log_2 n$** levels
- On every level **n** elements are merged in total → **O(n)** per level
- In total: **O(n log n)** – in **every** case (best, average, worst)

Property	Merge sort
Steps	**O(n log n)** always
Extra space	**O(n)** – does not sort in place
Stable	**yes** (`≤` in the merge)
On a linked list	works well (no direct access needed)

For $n = 10^6$: $n^2 = 10^{12}$ steps, $n \log_2 n \approx 2 \cdot 10^7$ steps – **50 000 times** fewer!

Quicksort

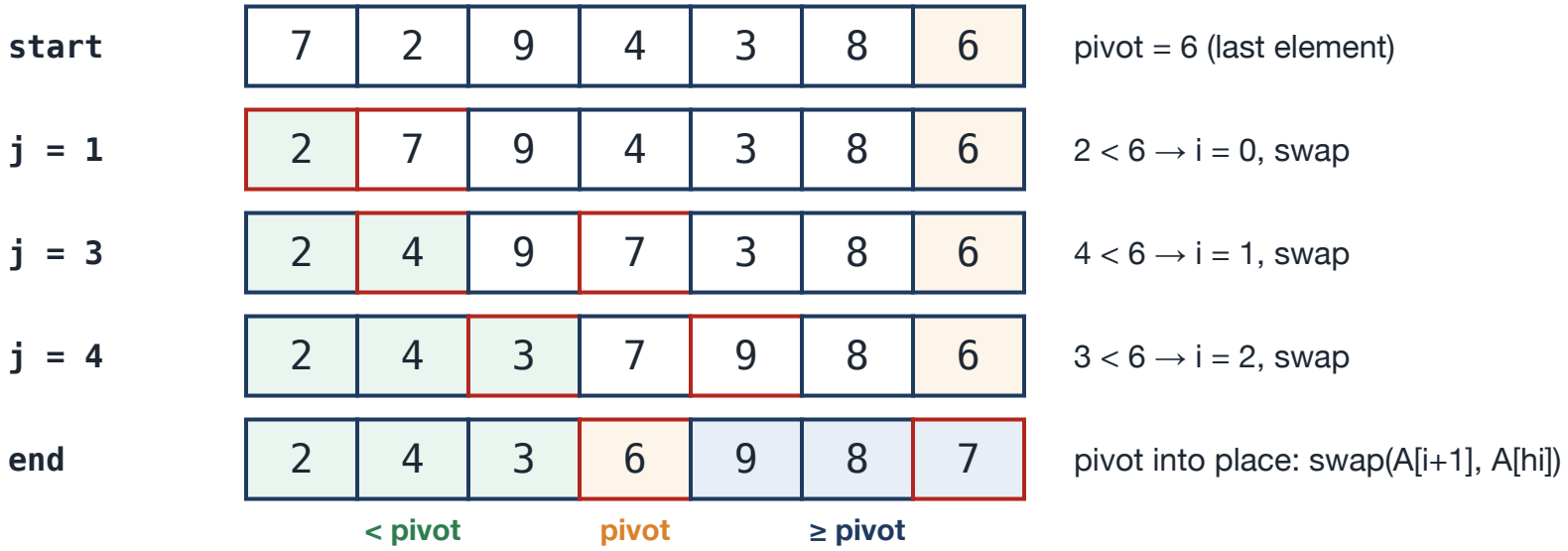
Quicksort – the idea

C. A. R. Hoare (1959)

1. Choose an element: the **pivot**
2. **Partitioning:** rearrange the array so that the elements **smaller** than the pivot go **left** and the **larger** ones go **right** – this way the pivot gets to its **final position**
3. Sort the left and the right part **recursively**

> Unlike merge sort, here the **dividing** is the work, and combining is “free”: if both parts are sorted, the whole is sorted.

Partitioning (Lomuto scheme)



Quicksort – algorithm

Pseudocode

```
Procedure QuickSort(A, left, right)
  If left < right then
    p ← Partition(A, left, right)
    QuickSort(A, left, p - 1)
    QuickSort(A, p + 1, right)
  End if
End procedure

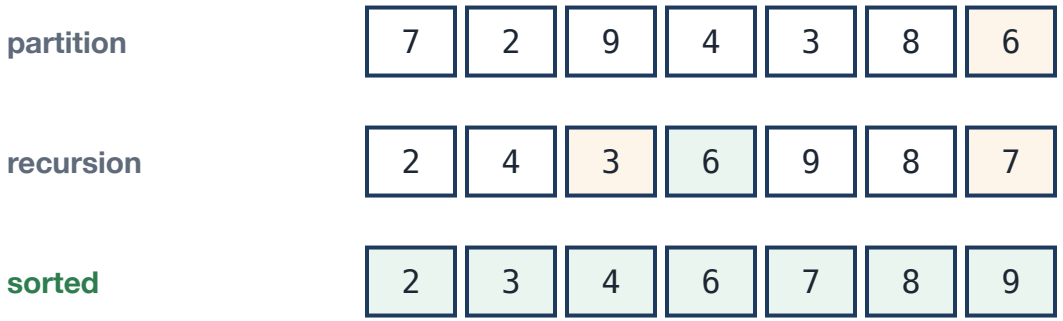
Function Partition(A, left, right)
  pivot ← A[right]
  i ← left - 1
  For j ← left to right - 1
    If A[j] < pivot then
      i ← i + 1
      swap(A[i], A[j])
    End if
  End for
  swap(A[i + 1], A[right])
  Return i + 1
End function
```

Java

```java
static void quickSort(int[] a, int lo, int hi) {
    if (lo >= hi) return;
    int p = partition(a, lo, hi);
    quickSort(a, lo, p - 1);
    quickSort(a, p + 1, hi);
}

static int partition(int[] a, int lo, int hi) {
    int pivot = a[hi];
    int i = lo - 1;
    for (int j = lo; j < hi; j++)
        if (a[j] < pivot) {
            i++;
            swap(a, i, j);
        }
    swap(a, i + 1, hi);
    return i + 1;
}
```

The importance of choosing the pivot



Good pivot (near the median): halves → log n levels → O(n log n)

Bad pivot (always min or max): n levels → $O(n^2)$

Quicksort – analysis

Case	When?	Steps
Best	the pivot is always the median	O(n log n)
Average	random data	**O(n log n)** (~1.39 · n $\log_2$ n comparisons)
Worst	the pivot is always the min. or the max. (e.g. a **sorted** array, last element as pivot)	**O(n^2)**

Measurement (n = 5000, sorted input): fixed pivot **12 497 500**, random pivot **68 648** comparisons

Protection against the worst case:

- **Random pivot** – the probability of the bad case is negligible
- **Median of three** (the median of the first, the middle and the last element)
- Recursion on the smaller part, a loop for the larger one → stack depth O(log n)

Quicksort – properties

Property	Quicksort
Steps	O(n log n) on average, O(n^2) in the worst case
Extra space	**O(log n)** (the stack of the recursion) – sorts **in place**
Stable	**no** (because of the distant swaps)
In practice	usually the **fastest** – little data movement, cache-friendly

> Java `Arrays.sort` uses **Dual-Pivot Quicksort** for primitive arrays, Go `slices.Sort` uses **pdqsort** (an improved version of quicksort).

Comparison

Comparison of the sorting algorithms

	Worst	Average	Best	Extra space	Stable
Bubble	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	no
Insertion	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	yes
Merge	**$O(n \log n)$**	$O(n \log n)$	$O(n \log n)$	$O(n)$	**yes**
Quick	$O(n^2)$	**$O(n \log n)$**	$O(n \log n)$	$O(\log n)$	no
Heapsort (week 15)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	no
Counting (week 5)	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	yes

Lower bound: a sorting algorithm based only on comparisons **cannot be faster than $\Omega(n \log n)$** in the worst case. Counting sort is faster because it counts instead of comparing (only for a small range of values).

Measurements (random data)

n	insertion	merge	quick	built-in
1 000	0.8 ms	0.2 ms	0.2 ms	0.9 ms
10 000	13.3 ms	0.7 ms	0.6 ms	1.5 ms
100 000	**1 159 ms**	16.5 ms	4.4 ms	5.7 ms
1 000 000	(~2 minutes)	65.6 ms	49.8 ms	43.6 ms

Java, `SortBenchmark` program. For small **n** the built-in sort looks slow because of the JVM warm-up.

External sorting

What if the data **does not fit into memory** (e.g. a 100 GB file, 8 GB RAM)?

External merge sort:

1. Read as much as fits (e.g. 1 GB), sort it in memory and write it to a temporary file → 100 sorted “runs”
2. **Merge** the runs: keep only the beginning of each one in memory, and always write out the smallest (**k**-way merge – with a priority queue)

- The number of slow disk operations is minimal: every data item is read and written **twice**
- Database systems and big data processing systems sort this way

Summary

- **Divide and conquer:** divide → solve recursively → combine
- **Merge sort:** halving + merging · always **O(n log n)** · stable · O(n) extra space
- **Quicksort:** pivot + partitioning · **O(n log n)** on average, $O(n^2)$ in the worst case · in place · not stable
- Choosing the pivot is critical – **random pivot** or median of three
- A comparison sort cannot be faster than **O(n log n)**
- **External sorting:** runs + **k**-way merge

Next week: trees – concepts, binary trees, representation, traversals.

Review questions

1. Draw the splitting and merging tree of merge sort for the array `[5, 1, 4, 2, 8, 3]`!
2. Partition the array `[4, 8, 1, 6, 3, 7, 5]` with the Lomuto scheme (pivot = 5)! Where does the pivot go?
3. True or false: quicksort sorts in O(n log n) steps even in the worst case.
4. Which sorting algorithm would you choose if stability is important and guaranteed O(n log n) is needed?
5. Why is quicksort slow on a sorted array if the last element is always the pivot?
6. How many levels does the recursion tree of merge sort have for n = 1024?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Merge sort	`MergeSort.java`	`w13/mergesort`	the order of the merges, recursion depth
Quicksort	`QuickSort.java`	`w13/quicksort`	partitioning, bad and random pivot
Measurement	`SortBenchmark.java`	`w13/benchmark`	$O(n^2)$ vs $O(n \log n)$ vs built-in

Run the `SortBenchmark` program! How many times longer does it take when **n** grows tenfold – for insertion sort and for merge sort?

✎ Exercises

1. **Descending order:** modify merge sort to sort in descending order!
2. **Stable sorting:** sort words by length so that the order of words of equal length is kept!
3. **Inversions in O(n log n):** count the inversions while merging! (compare with the $O(n^2)$ solution of week 5)
4. **Quickselect:** find the **k**-th smallest element without sorting, with partitioning – O(n) on average!
5. **Dutch national flag:** sort an array containing only the values 0, 1 and 2 in a single pass!
6. **Median:** the common median of two sorted arrays by merging.

Solutions: `Exercises13.java`, `w13/exercises`

✎ Homework

1. Implement quicksort with **median-of-three** pivot selection! Measure it on sorted and reversed input!
2. **Hybrid sorting:** if the part has fewer than 16 elements, use insertion sort! How much faster does it get according to the measurements?
3. Write merge sort for a **linked list** (without an auxiliary array)! (Hint: split the list with the tortoise and hare method of week 9.)
4. Implement the **Hoare** partition scheme (two indexes moving towards each other)! Compare the number of swaps with the Lomuto scheme!