Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

CHAI
Humanities-Centered AI

Transformer-based Algorithmics

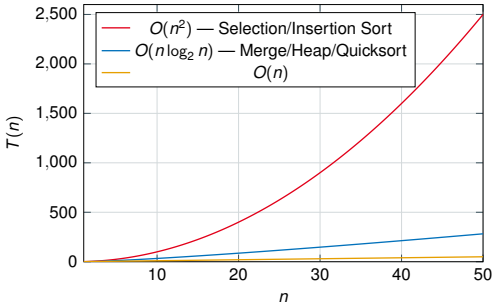
Malte Luttermann – Institute for Humanities-Centered Artificial Intelligence (CHAI)

April 22, 2026

Version from May 5, 2026

Topics for Today

- O, Ω, Θ-notation
- Merge Sort
- Quicksort
- Heap Sort



Acknowledgement

- Some of the upcoming content is taken from the following lectures and corresponding exercises and has been translated and partially modified:
  - Prof. Dr. Ralf Möller: »Algorithmen und Datenstrukturen – Vorlesung«
  - Dr. Magnus Bender: »Algorithmen und Datenstrukturen – Übung«

O, Ω, Θ-Notation

Asymptotic Notation

Upper bound (O-notation):

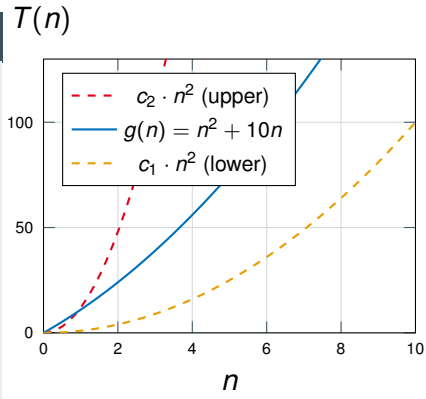
$$O(f) = \left\{ g \colon \mathbb{N} \mapsto \mathbb{N} \;\middle|\; \exists\, n_0, c > 0 \colon \forall n \geq n_0 \colon g(n) \leq c \cdot f(n) \right\}$$

Lower bound (Ω-notation):

$$\Omega(f) = \left\{ g \colon \mathbb{N} \mapsto \mathbb{N} \;\middle|\; \exists\, n_0, c > 0 \colon \forall n \geq n_0 \colon g(n) \geq c \cdot f(n) \right\}$$

Tight bound (Θ-notation):

$$\Theta(f) = O(f) \cap \Omega(f)$$



$$g(n) = n^2 + 10n \in \Theta(n^2)$$

Exercise 1 – Ω and Θ-Notation

Prove the following claims:

(1) $T_1(n) = 3^n + 9n \notin \Omega(4^n)$
(2) $T_2(n) = n^3 + 11n \in \Theta(n^3)$

Asymptotic Notation

$$O(f) = \left\{ g \colon \mathbb{N} \mapsto \mathbb{N} \;\middle|\; \exists\, n_0, c > 0 \colon \forall n \geq n_0 \colon g(n) \leq c \cdot f(n) \right\}$$
$$\Omega(f) = \left\{ g \colon \mathbb{N} \mapsto \mathbb{N} \;\middle|\; \exists\, n_0, c > 0 \colon \forall n \geq n_0 \colon g(n) \geq c \cdot f(n) \right\}$$
$$\Theta(f) = O(f) \cap \Omega(f)$$

Reminder

- $g \notin \Omega(f) \iff \forall\, n_0, c > 0 \colon \exists\, n \geq n_0 \colon g(n) < c \cdot f(n)$

Exercise 1 – Solution

(1) Claim: $T_1(n) = 3^n + 9n \notin \Omega(4^n)$

Show $\forall\, n_0, c > 0 \colon \exists\, n \geq n_0 \colon\ T_1(n) < c \cdot 4^n$:

$$3^n + 9n < c \cdot 4^n$$

$$\Leftrightarrow \left(\tfrac{3}{4}\right)^n + \tfrac{9n}{4^n} < c$$

Both $\left(\frac{3}{4}\right)^n \to 0$ and $\frac{9n}{4^n} \to 0$ as $n \to \infty$.

Thus, for any $c > 0$ and n_0, we can always find $n \geq n_0$ satisfying $\left(\frac{3}{4}\right)^n + \frac{9n}{4^n} < c$. $\square$

Exercise 1 – Solution

(2) Claim: $T_2(n) = n^3 + 11n \in \Theta(n^3)$

$T_2 \in O(n^3)$: Choose $c = 12,\ n_0 = 1$:

$$n^3 + 11n \leq 12n^3 \quad \forall n \geq 1$$

since $11n \leq 11n^3$ for all $n \geq 1$.

$T_2 \in \Omega(n^3)$: Choose $c = 1,\ n_0 = 1$:

$$n^3 + 11n \geq n^3 \quad \forall n \geq 1$$

since $11n \geq 0$.

Since, $T_2 \in O(n^3)$ and $T_2 \in \Omega(n^3)$, we have $T_2(n) \in \Theta(n^3)$. □

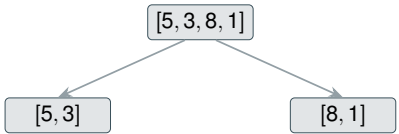
Merge Sort – Divide and Conquer

(1) **Divide:**
Split $A[p \dots r]$ at midpoint
$q = \lfloor (p + r) / 2 \rfloor$

[5, 3, 8, 1]

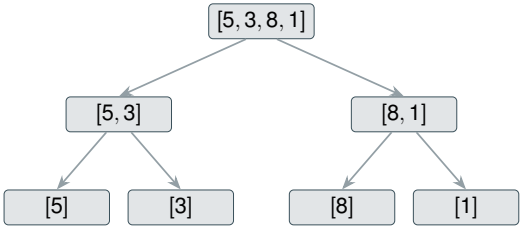
Merge Sort – Divide and Conquer

(1) **Divide:**
Split $A[p \dots r]$ at midpoint
$q = \lfloor (p + r) / 2 \rfloor$



Merge Sort – Divide and Conquer

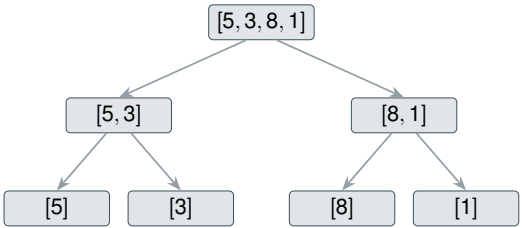
(1) **Divide:**
Split $A[p \dots r]$ at midpoint
$q = \lfloor (p + r) / 2 \rfloor$



Merge Sort – Divide and Conquer

(1) **Divide:**
Split $A[p \dots r]$ at midpoint $q = \lfloor (p + r) / 2 \rfloor$

(2) **Conquer:**
Recursively sort each half

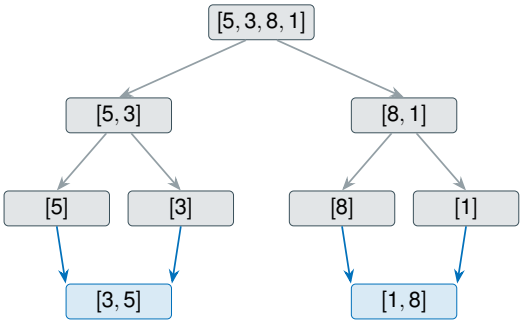


Merge Sort – Divide and Conquer

(1) **Divide:**
Split $A[p \dots r]$ at midpoint $q = \lfloor (p + r) / 2 \rfloor$

(2) **Conquer:**
Recursively sort each half

(3) **Combine:**
Merge two sorted halves in $O(n)$

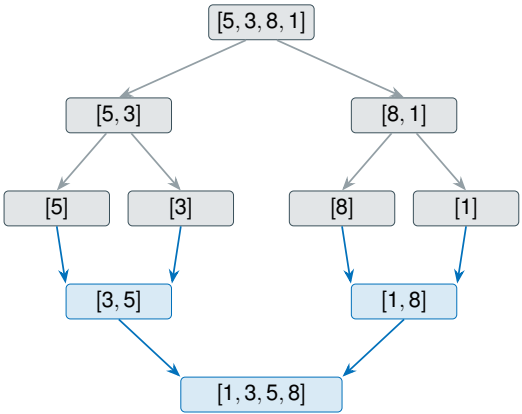


Merge Sort – Divide and Conquer

(1) **Divide:**
Split $A[p \dots r]$ at midpoint $q = \lfloor (p + r) / 2 \rfloor$

(2) **Conquer:**
Recursively sort each half

(3) **Combine:**
Merge two sorted halves in $O(n)$

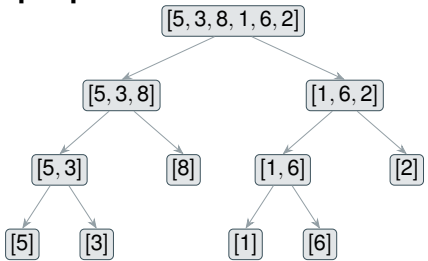


Exercise 2 – Merge Sort

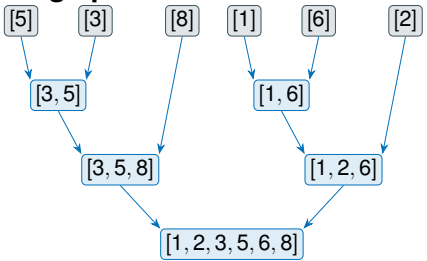
Sort the given sequence using *Merge Sort* (ascending).
Show the full recursion tree (split and merge phases).

Exercise 2 – Solution

Split phase:



Merge phase:



Merge Sort – Runtime Analysis

Recurrence relation for $T(n)$:

$$T(n) = \underbrace{2T(n/2)}_{\text{recursive calls}} + \underbrace{n}_{\text{merge step}}$$

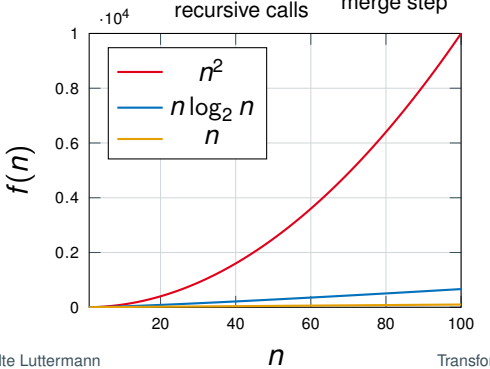
Iterative expansion (assume $n = 2^k$):

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^k T(1) + k \cdot n \\ &= n \cdot T(1) + n \cdot \log_2 n \end{aligned}$$

Merge Sort – Runtime Analysis

Recurrence relation for $T(n)$:

$$T(n) = \underbrace{2T(n/2)}_{\text{recursive calls}} + \underbrace{n}_{\text{merge step}}$$



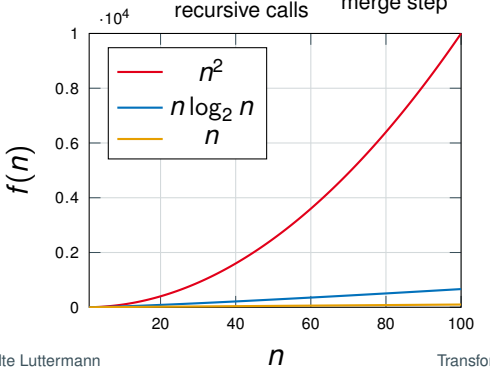
Iterative expansion (assume $n = 2^k$):

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^k T(1) + k \cdot n \\ &= n \cdot T(1) + n \cdot \log_2 n \end{aligned}$$

Merge Sort – Runtime Analysis

Recurrence relation for $T(n)$:

$$T(n) = \underbrace{2T(n/2)}_{\text{recursive calls}} + \underbrace{n}_{\text{merge step}}$$



Iterative expansion (assume $n = 2^k$):

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^k T(1) + k \cdot n \\ &= n \cdot T(1) + n \cdot \log_2 n \end{aligned}$$

$$\Rightarrow T(n) \in O(n \cdot \log_2 n)$$

Exercise 3 – Quicksort

Sort the following sequence using *Quicksort* as introduced in the lecture.

For each call to `partition(A, i, j)`, state the **pivot element**, the resulting sub-sequences, and the index *p*.


```
function quicksort(A, i, j)
  if i < j
    p = partition(A, i, j)
    quicksort(A, i, p-1)
    quicksort(A, p+1, j)
  end
end
```

```
function partition(A, i, j)
  x = A[j]
  b = i - 1
  for k = i:j
    temp = A[k]
    A[k] = A[b+1]
    A[b+1] = temp
    if A[b+1] <= x
      b = b + 1
    end
  end
  return b
end
```

Exercise 3 – Solution

pivot $= A[8] = 9$

Exercise 3 – Solution

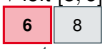
pivot $= A[8] = 9$

$\hookrightarrow$ left $[8, 6]$: **pivot** $= A[2] = 6$

Exercise 3 – Solution

pivot $= A[8] = 9$

$\hookrightarrow$ left $[8, 6]$: **pivot** $= A[2] = 6$

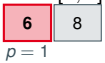


$\hookrightarrow$ right $[43, 34, 12, 87, 13]$: **pivot** $= A[8] = 13$

Exercise 3 – Solution

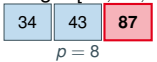
pivot $= A[8] = 9$

$\hookrightarrow$ left $[8, 6]$: **pivot** $= A[2] = 6$

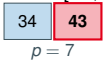


$\hookrightarrow$ right $[43, 34, 12, 87, 13]$: **pivot** $= A[8] = 13$

$\hookrightarrow$ right $[34, 43, 87]$: **pivot** $= A[8] = 87$



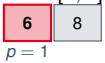
$\hookrightarrow$ left $[34, 43]$: **pivot** $= A[7] = 43$



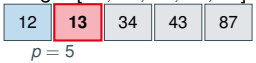
Exercise 3 – Solution

pivot $= A[8] = 9$

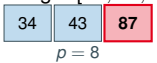
↪ left $[8, 6]$: **pivot** $= A[2] = 6$



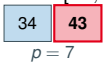
↪ right $[43, 34, 12, 87, 13]$: **pivot** $= A[8] = 13$



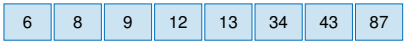
↪ right $[34, 43, 87]$: **pivot** $= A[8] = 87$



↪ left $[34, 43]$: **pivot** $= A[7] = 43$



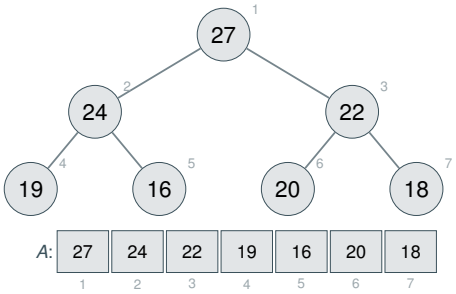
Result:



Heap Sort – Binary Tree as Array

A binary tree stored in array $A[1..n]$:

Tree concept	Array index
`root`	1
`left_child(v)`	$2v$
`right_child(v)`	$2v + 1$
`parent(v)`	$\lfloor v / 2 \rfloor$
`lastnode`	n (initially)
`is_leaf(v)`	$2v > n / 2$



Max-Heap Property

For every node v: $A[v] \geq A[\text{left\_child}(v)]$ and $A[v] \geq A[\text{right\_child}(v)]$
$\Rightarrow$ The **largest element** is always at the root.

Exercise 4 – Heap Sort

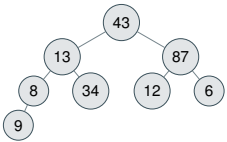
Sort the following sequence using *Heap Sort* as introduced in the lecture.


```
function make_heap(A, n)
    p = parent(n)
    root = 1
    for v = p:-1:root
        heapify(A, v, n)
    end
end
```

```
function heapify(A, k, lastnode)
    if !is_leaf(A, k, lastnode)
        left = left_child(k)
        right = right_child(k)
        maxc = left
        if exists(A, right, lastnode) && A[right]
             > A[left]
            maxc = right
        end
        if A[maxc] > A[k]
            temp = A[maxc]
            A[maxc] = A[k]
            A[k] = temp
            heapify(A, maxc, lastnode)
        end
    end
end
```

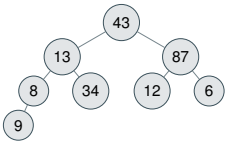
Exercise 4 – Solution: Building the Max-Heap

Initial tree:

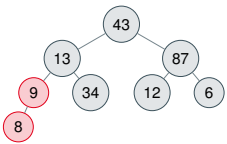


Exercise 4 – Solution: Building the Max-Heap

Initial tree:

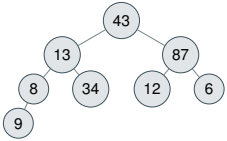


`heapify`(8): swap $8 \leftrightarrow 9$:

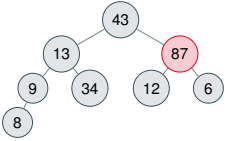


Exercise 4 – Solution: Building the Max-Heap

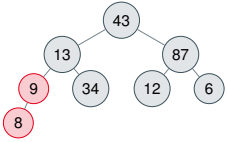
Initial tree:



`heapify`(87): no swap:

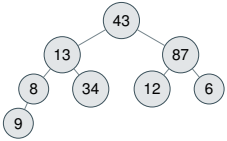


`heapify`(8): swap 8 $\leftrightarrow$ 9:

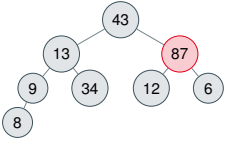


Exercise 4 – Solution: Building the Max-Heap

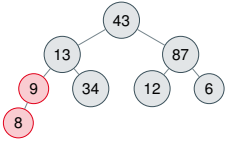
Initial tree:



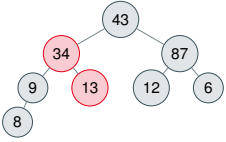
`heapify`(87): no swap:



`heapify`(8): swap 8 $\leftrightarrow$ 9:

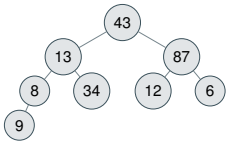


`heapify`(13): swap 13 $\leftrightarrow$ 34:

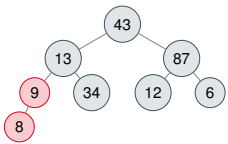


Exercise 4 – Solution: Building the Max-Heap

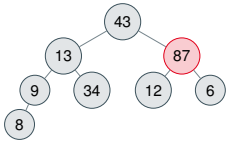
Initial tree:



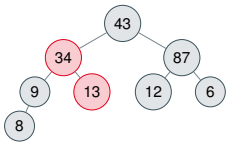
`heapify`(8): swap 8 ↔ 9:



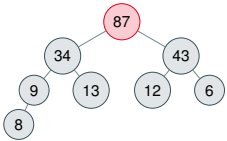
`heapify`(87): no swap:



`heapify`(13): swap 13 ↔ 34:



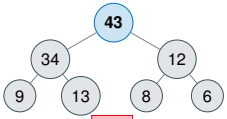
`heapify`(43): swap 43 ↔ 87:



⇒ Max-Heap

Exercise 4 – Solution: Sorting (1/2)

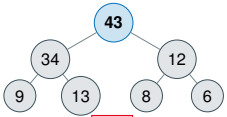
8 sifted down:



Sorted: 87

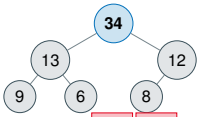
Exercise 4 – Solution: Sorting (1/2)

8 sifted down:



Sorted: 87

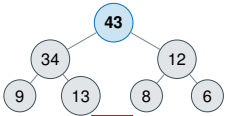
6 sifted down:



Sorted: 43 87

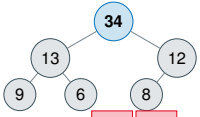
Exercise 4 – Solution: Sorting (1/2)

8 sifted down:



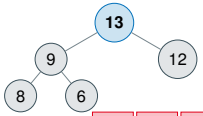
Sorted: 87

6 sifted down:



Sorted: 43 87

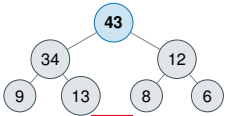
8 sifted down:



Sorted: 34 43 87

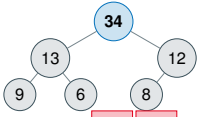
Exercise 4 – Solution: Sorting (1/2)

8 sifted down:



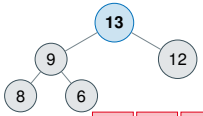
Sorted: 87

6 sifted down:



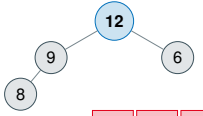
Sorted: 43 87

8 sifted down:



Sorted: 34 43 87

6 sifted down:



Sorted: 13 34 43 87

Exercise 4 – Solution: Sorting (2/2)

8 sifted down:

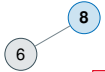
Sorted:

Exercise 4 – Solution: Sorting (2/2)

8 sifted down:

Sorted: 12 | 13 | 34 | 43 | 87

6 sifted down:



Sorted: 9 | 12 | 13 | 34 | 43 | 87

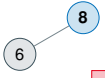
Exercise 4 – Solution: Sorting (2/2)

8 sifted down:

Sorted:

12	13	34	43	87

6 sifted down:



Sorted:

9	12	13	34	43	87

root = lastnode

Sorted:

8	9	12	13	34	43	87

Final result:

6	8	9	12	13	34	43	87