COMP 2402

Sorting

part 1

Alina Shaikhet

Carleton University

Sorting

- Searching in **ordered** data is much faster than searching in **unordered** data.
- Some sorting algorithms are closely related to data structures:
  - HeapSort → **heaps**
  - QuickSort → **random BST**
- Any of the **SSet** or **priority Queue** implementations (that we studied) can also be used to create an $O(n \log n)$ time sorting algorithm.

Is it worthwhile to sort the data before searching?

Is it worthwhile to build a data structure for searching only?

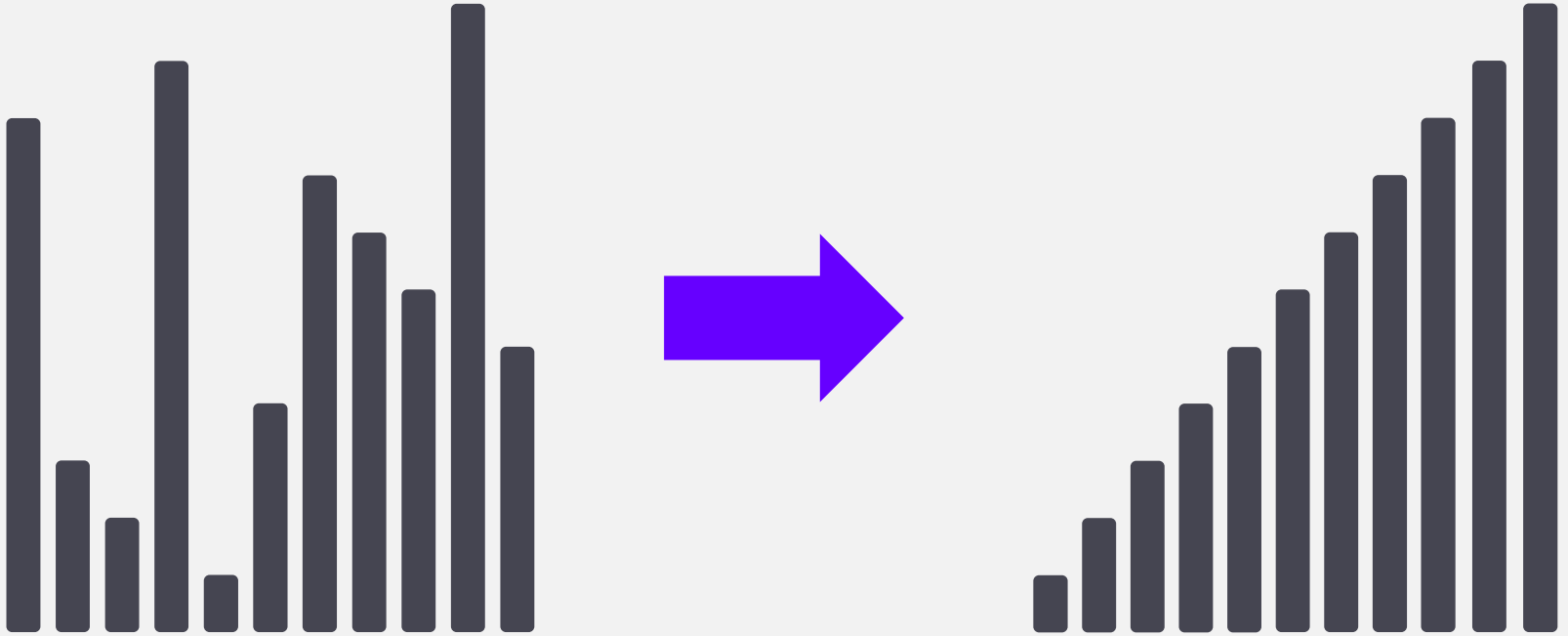
Searching in Sorted vs Unsorted Data

Linear Search – searching in **unordered** data – at most n comparisons

Binary Search – searching in **ordered/sorted** data – at most $\log_2 n + 1$ comparisons

n	$log_2 n$
$2^2 = 4$	2
$2^3 = 8$	3
$2^{10} = 1024$	10
$2^{20} = 1,048,576$	20
$2^{30} = 1,073,741,824$	30

Sorting



Selection Sort
$O(n^2)$

Bubble Sort
$O(n^2)$

Insertion Sort
$O(n^2)$

Merge Sort
$O(n \log n)$

Quick Sort
$O(n \log n)$ expected
$O(n^2)$ worst case

Heap Sort
$O(n \log n)$

Sorting

- Comparison-based Sorting:
  - **MergeSort** (worst-case)
  - **QuickSort** (expected)
  - **HeapSort** (worst-case)

  $\boldsymbol{O(n \log n)}$ — **asymptotically optimal**

- Using array indexing, it is possible to sort a set of $\boldsymbol{n}$ integers in the range $\{\boldsymbol{0}, \ldots, \boldsymbol{n^c - 1}\}$ in $\boldsymbol{O(cn)}$ time.

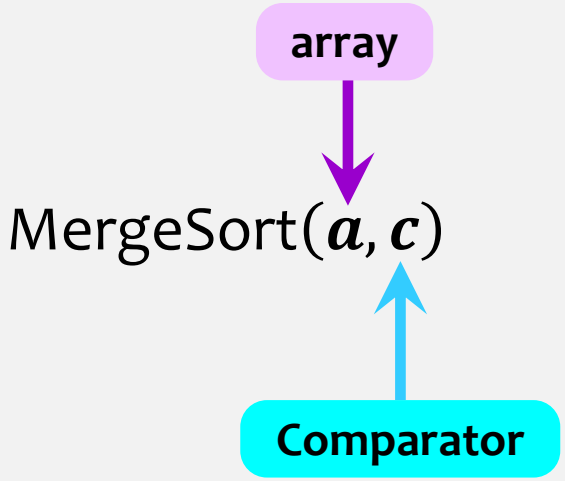
Sorting

- Comparison-based Sorting:

- **MergeSort** (worst-case)
- **QuickSort** (expected)
- **HeapSort** (worst-case)

$\boldsymbol{O(n \log n)}$ — **asymptotically optimal**

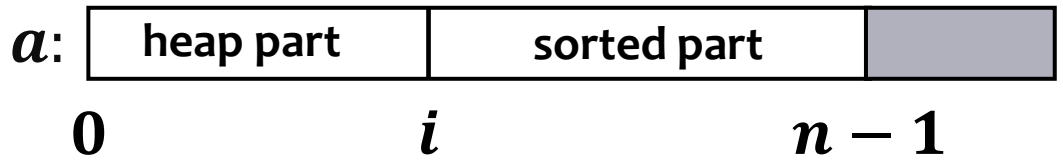
$$\text{compare}(x, y) \begin{cases} < 0 & \text{if } x < y \\ > 0 & \text{if } x > y \\ = 0 & \text{if } x = y \end{cases}$$



Review

Selection Sort and **Insertion Sort**

Recall **HeapSort**:



These sorting algorithms are each based on maintaining two lists:

1. one that is always sorted, and
2. one that is not.

The algorithms work by taking one element from the unsorted list and adding to the sorted list in the right place.

Selection Sort

Selection Sort Animation:
https://yongdanielliang.github.io/animation/web/SelectionSortNew.html

```
SelectionSort(A[1..n]) {
    for i from 1 to n-1 do {
        // find the smallest element in the unsorted part
        // of the array, and add it to the sorted part
        nextIndex := findSmallest(A[i..n])
        swap A[i] and A[nextIndex]
        // assert: A[1..i] is sorted
    }
}
```

$(n - i)$ comparisons

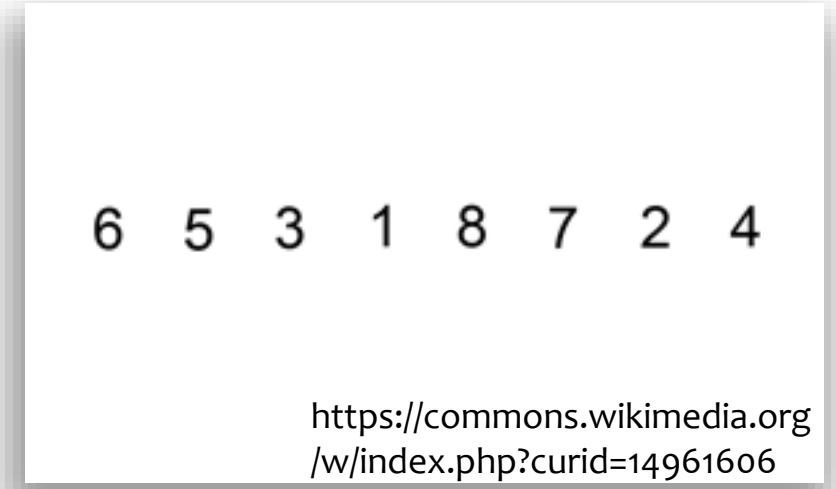
Number of comparisons: $\sum_{i=1}^{n-1}(n-i) = n\sum_{i=1}^{n-1}1 - \sum_{i=1}^{n-1}i = n(n-1) - \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$

Insertion Sort

Insertion Sort Animation:
https://yongdanielliang.github.io/animation/web/InsertionSortNew.html

```
InsertionSort(A[1..n]) {
    for i from 2 to n do {
        // assert: A[1..i-1] is sorted

        // insert A[i] into A[1..i-1] such that A[1..i] is sorted
        k := i
        while (k > 1 AND A[k-1] > A[k]) do {
            swap A[k-1] and A[k]
            k := k-1
        }
        // assert: A[1..i] is sorted
    }
}
```



https://commons.wikimedia.org/w/index.php?curid=14961606

Number of comparisons: $\sum_{i=2}^{n}(i-1) = \sum_{i=2}^{n} i - \sum_{i=2}^{n} 1 = \frac{n(n+1)}{2} - 1 - (n-2+1) = \frac{n^2}{2} - \frac{n}{2}$

$= O(n^2)$

Sorting

SelectionSort, InsertionSort – at most $n^2/2 - n/2$ comparisons

MergeSort – at most $n \log_2 n$ comparisons

HeapSort – at most $2n \log_2 n$ comparisons.

n^2	$n \log_2 n$	n
16	8	$2^2 = 4$
64	32	$2^3 = 8$
$1,048,576$	10240	$2^{10} = 1024$
$2^{40} > 1000$ billions	$20,971,520$	$2^{20} = 1,048,576$
$2^{60} >$ billion of billions	$32,212,254,720$	$2^{30} = 1,073,741,824$

Merge-Sort

John von Neumann (1945)

The **Merge-Sort** algorithm is a classic example of recursive **divide & conquer**.

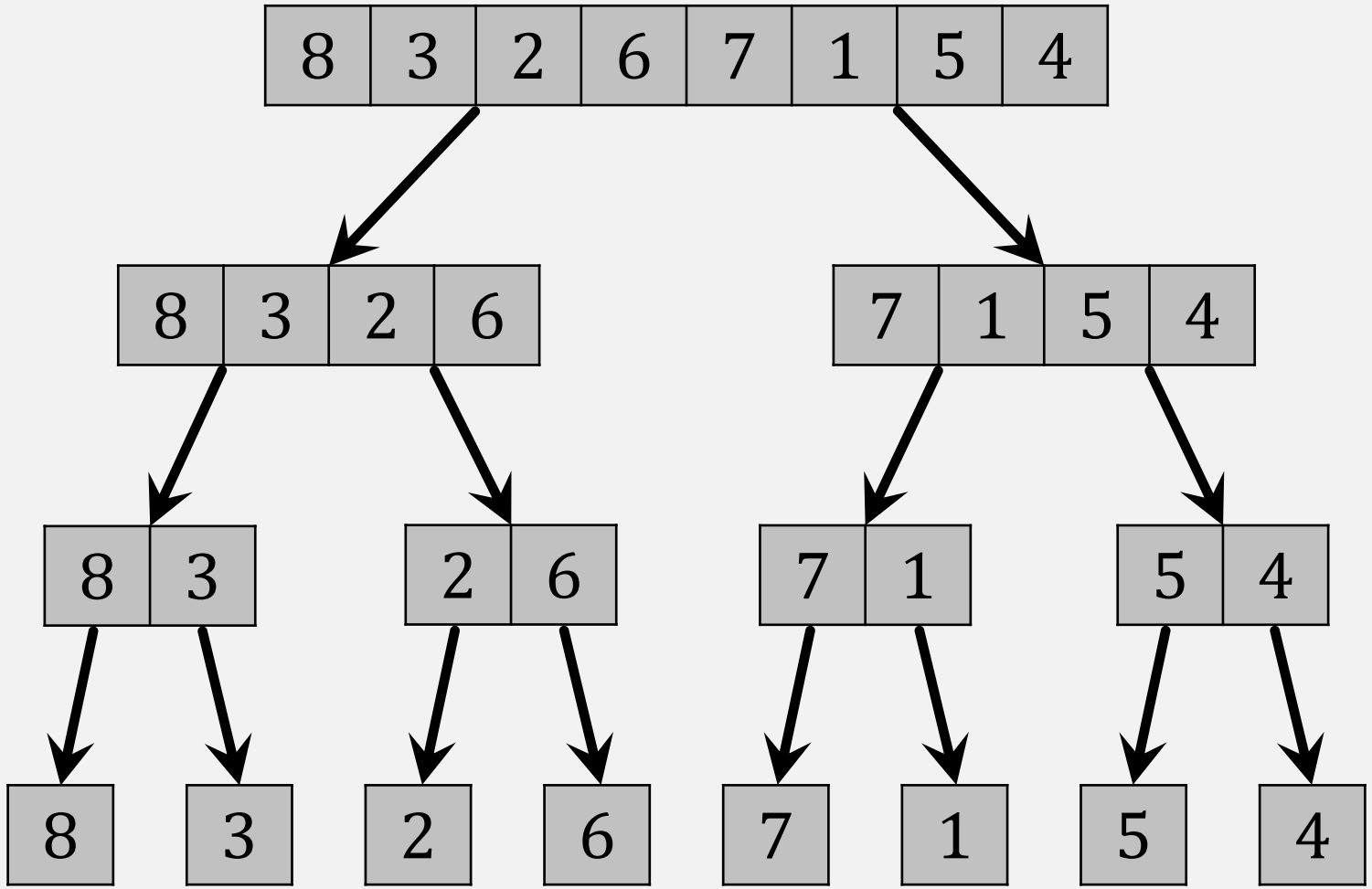
It is a **stable** sort, which means that the order of equal elements is the same in the input and output: If two elements $a[i]$ and $a[j]$ have the same value, and $i < j$ then $a[i]$ will appear before $a[j]$ in the sorted output.

To sort n items:

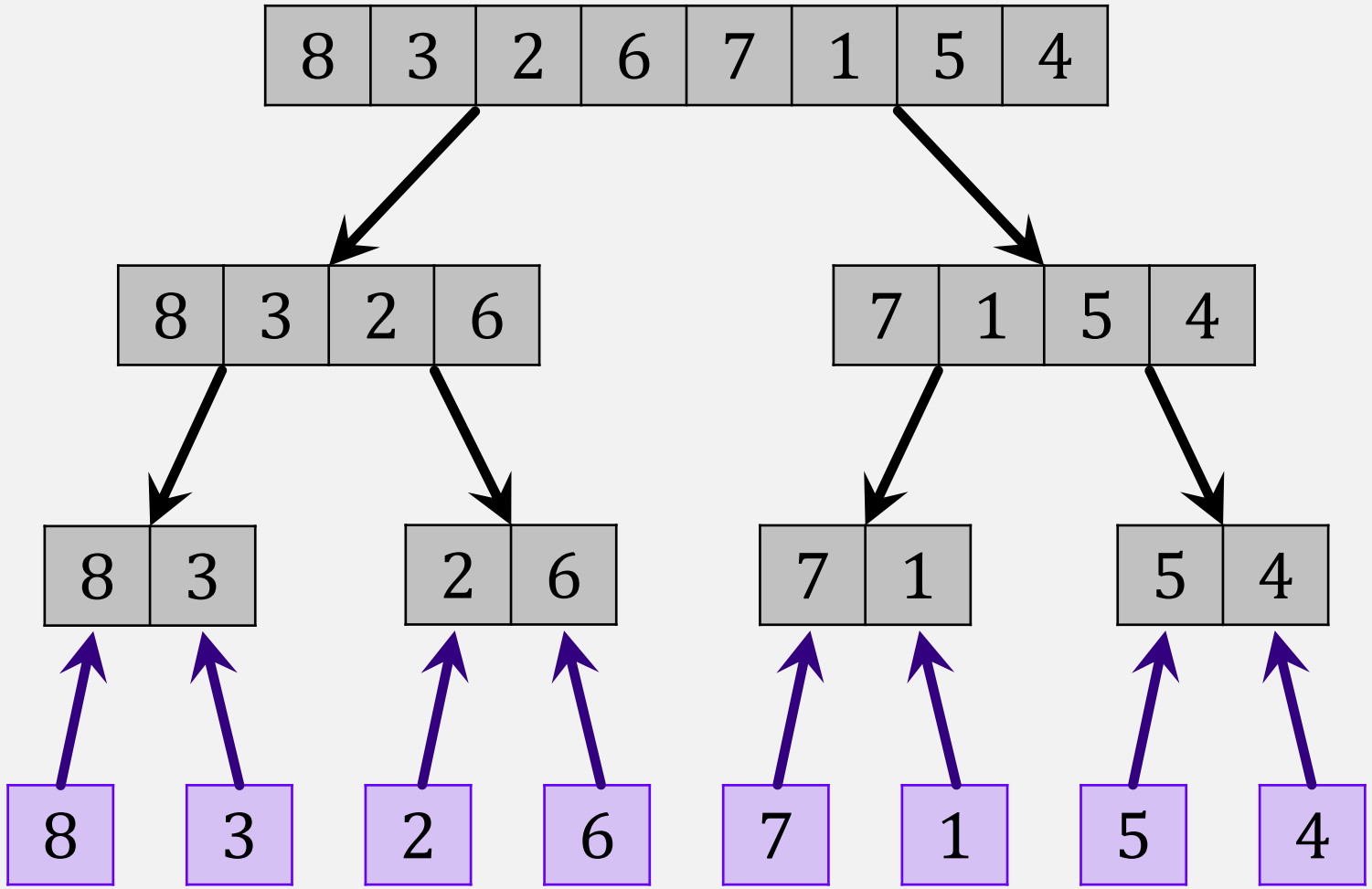
- if $n \leq 1$: nothing to do – just return.
- if $n \geq 2$:
  1. divide the n items arbitrarily into 2 sequences, both of size $n/2$;
  2. run Merge-Sort twice, once for each sequence;
  3. merge the two sorted sequences into one sorted sequence.

Merge-Sort requires an auxiliary array,
unless you are sorting a linked-list, in which case it requires only $\Theta(1)$ extra space.

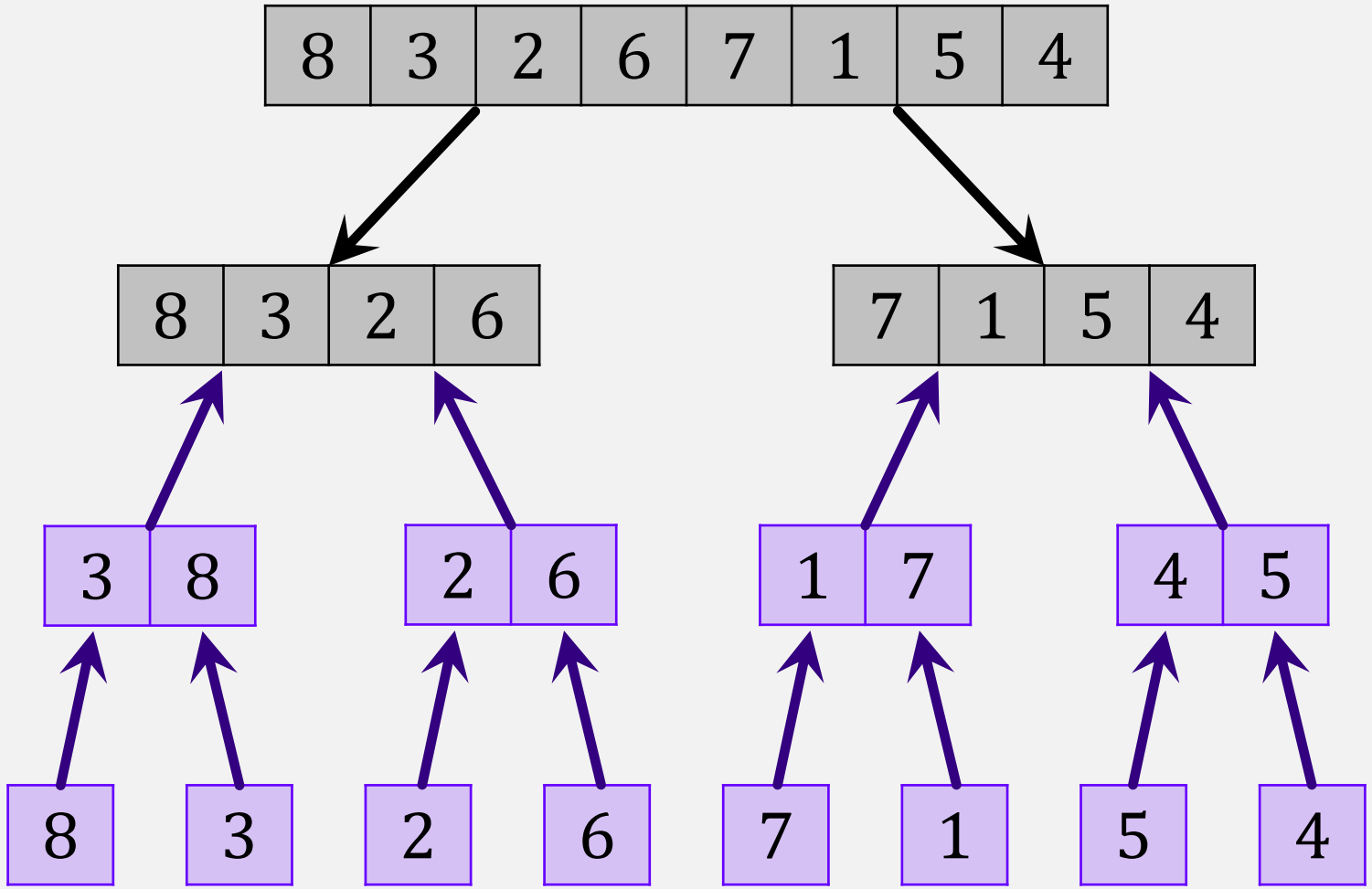
Merge-Sort



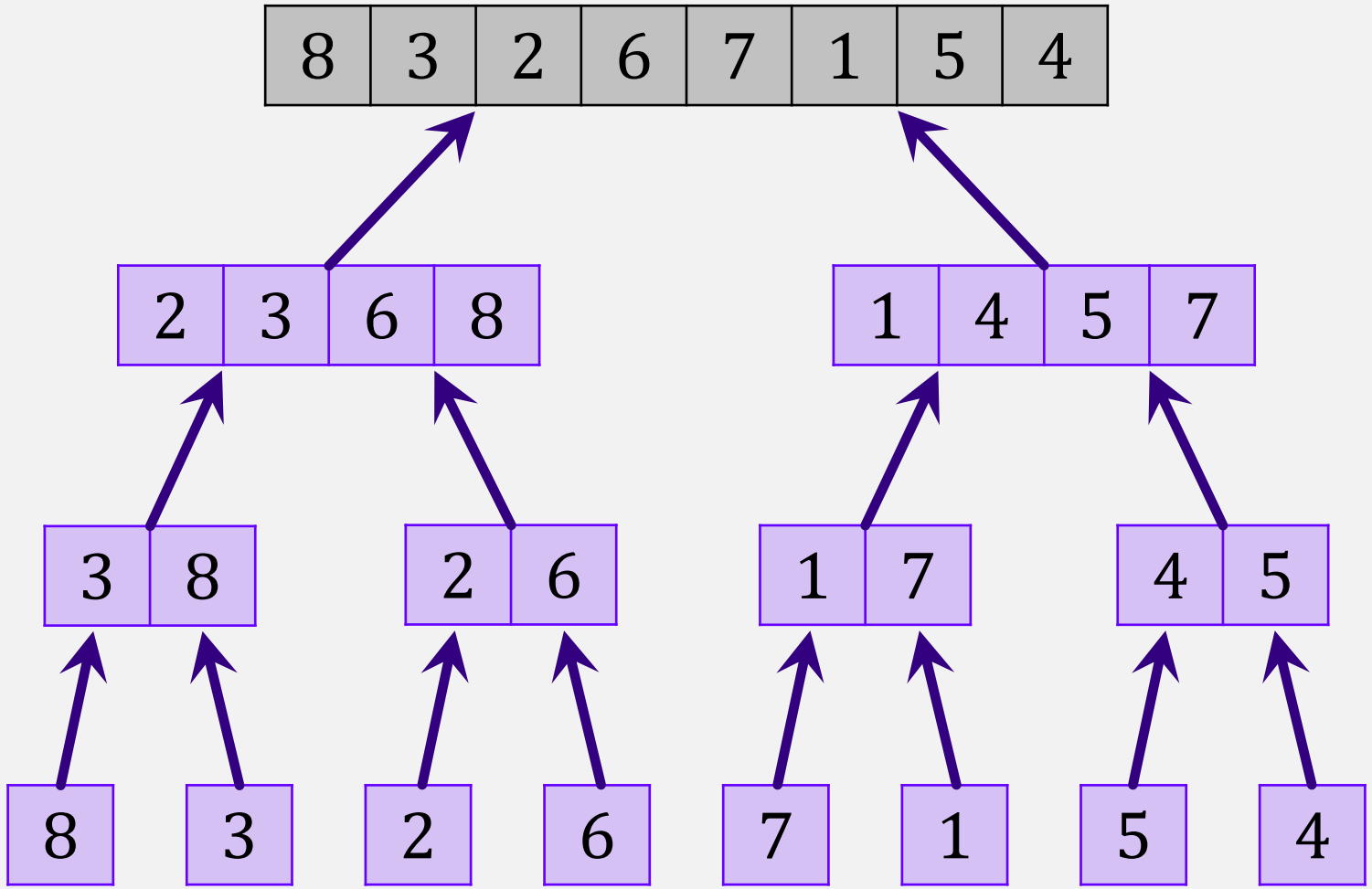
Merge-Sort



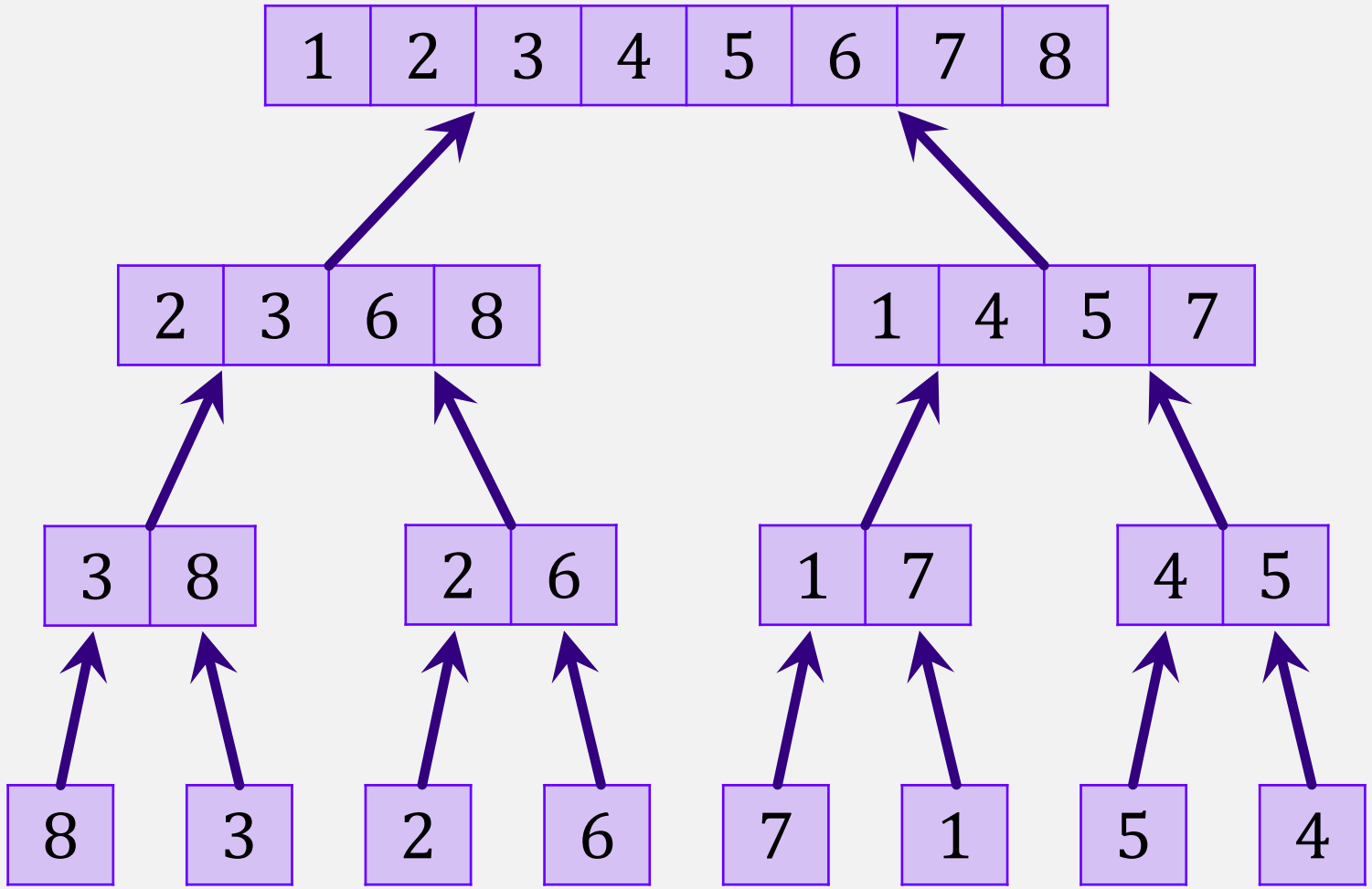
Merge-Sort



Merge-Sort

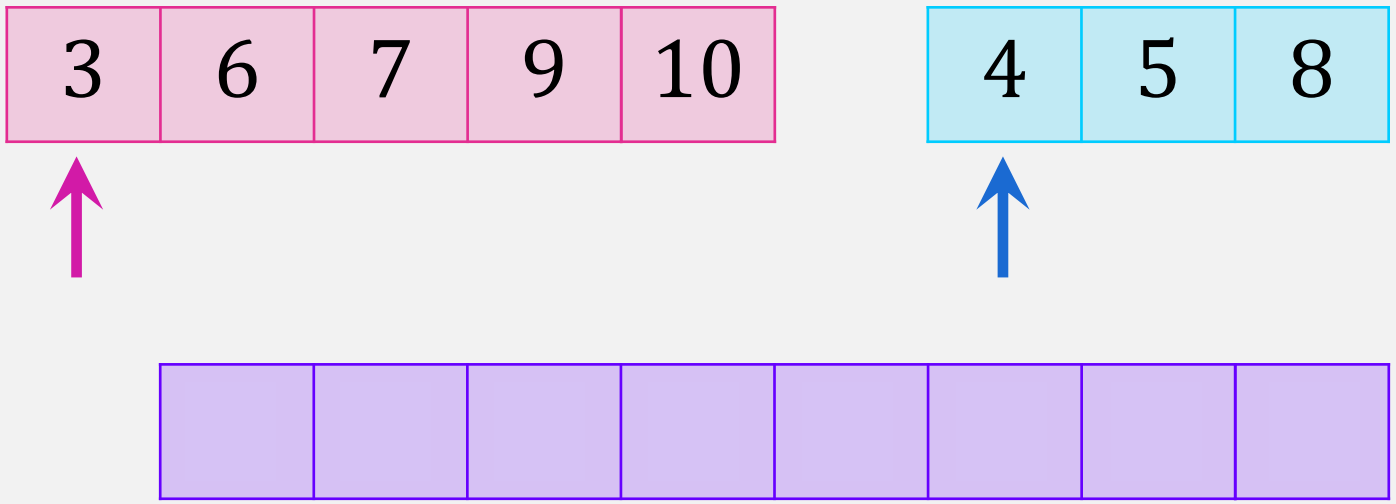


Merge-Sort



Merge

To merge two sorted arrays (or lists) into one sorted list scan them sequentially and perform comparisons

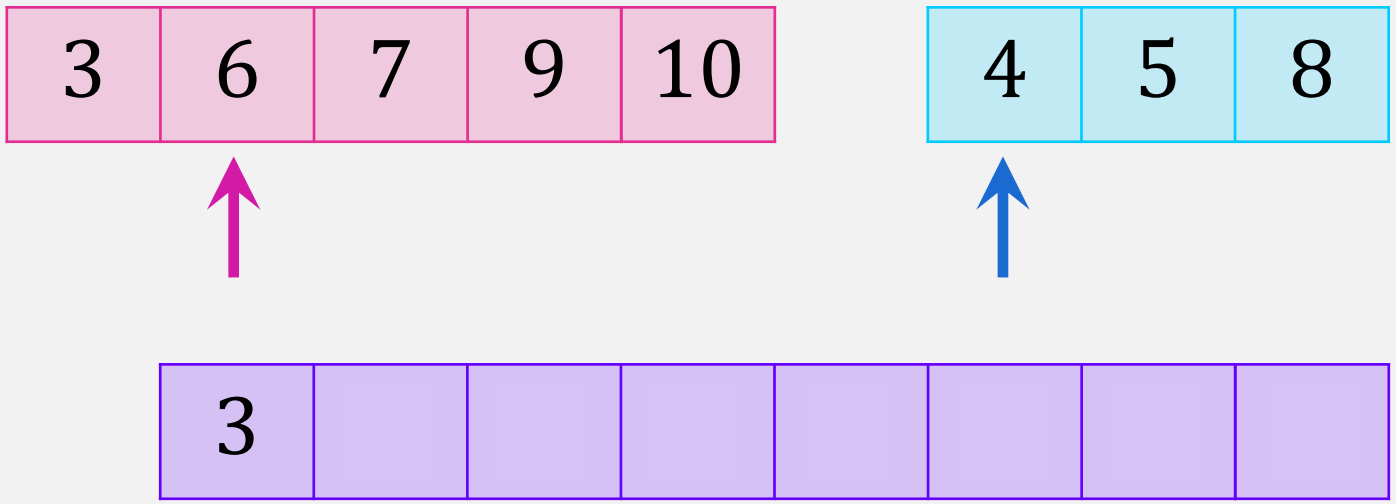


Comparisons:

$3 < 4$

Merge

To merge two sorted arrays (or lists) into one sorted list scan them sequentially and perform comparisons



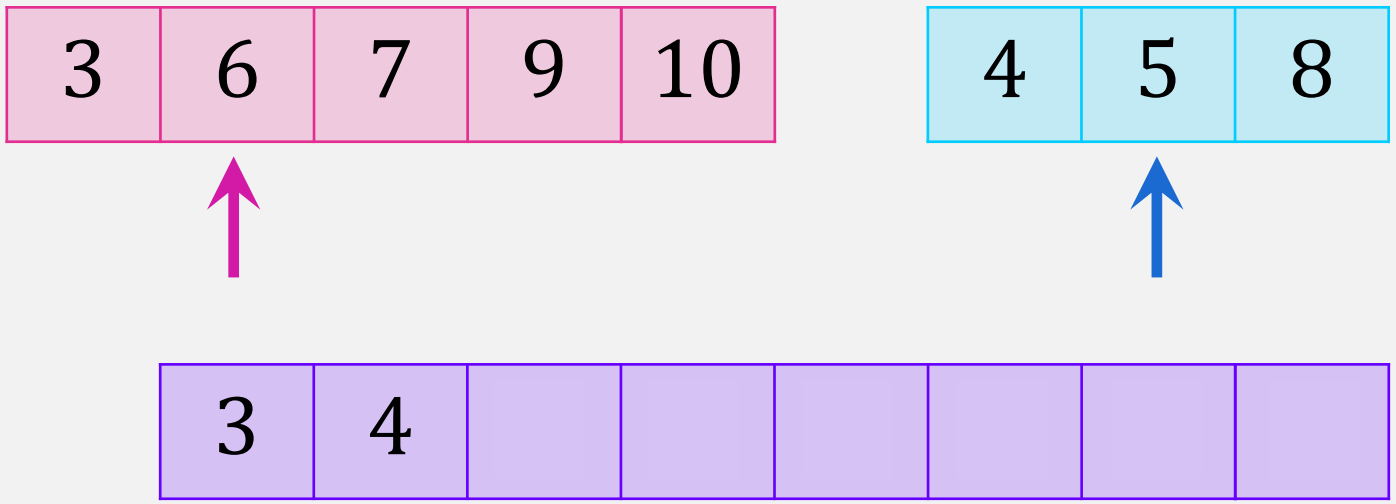
Comparisons:

$3 < 4$

$6 > 4$

Merge

To merge two sorted arrays (or lists) into one sorted list scan them sequentially and perform comparisons



Comparisons:

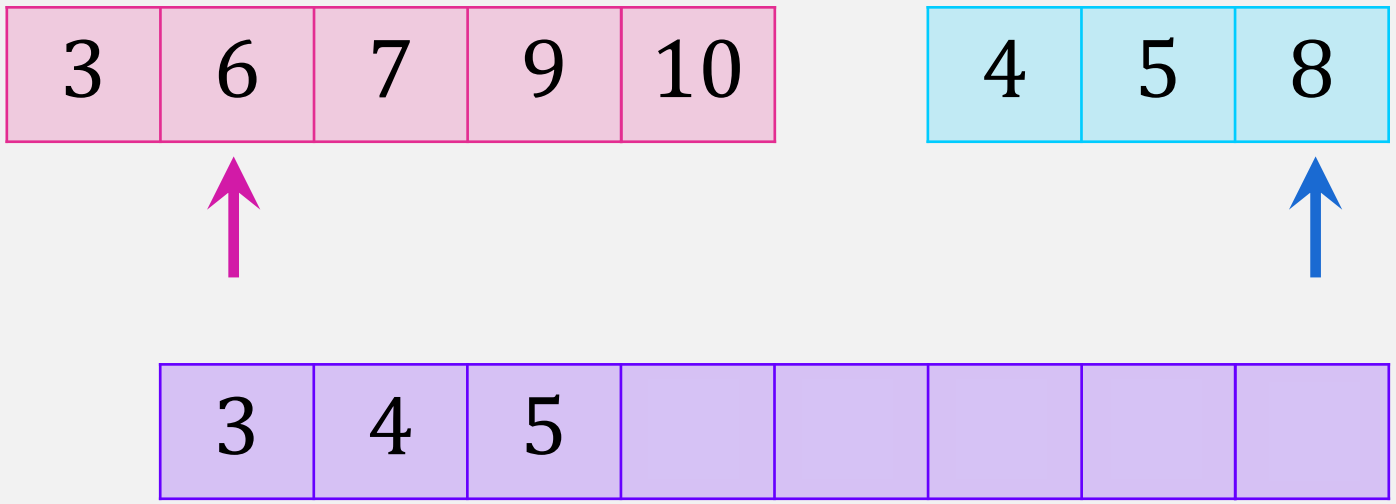
$3 < 4$

$6 > 4$

$6 > 5$

Merge

To merge two sorted arrays (or lists) into one sorted list scan them sequentially and perform comparisons

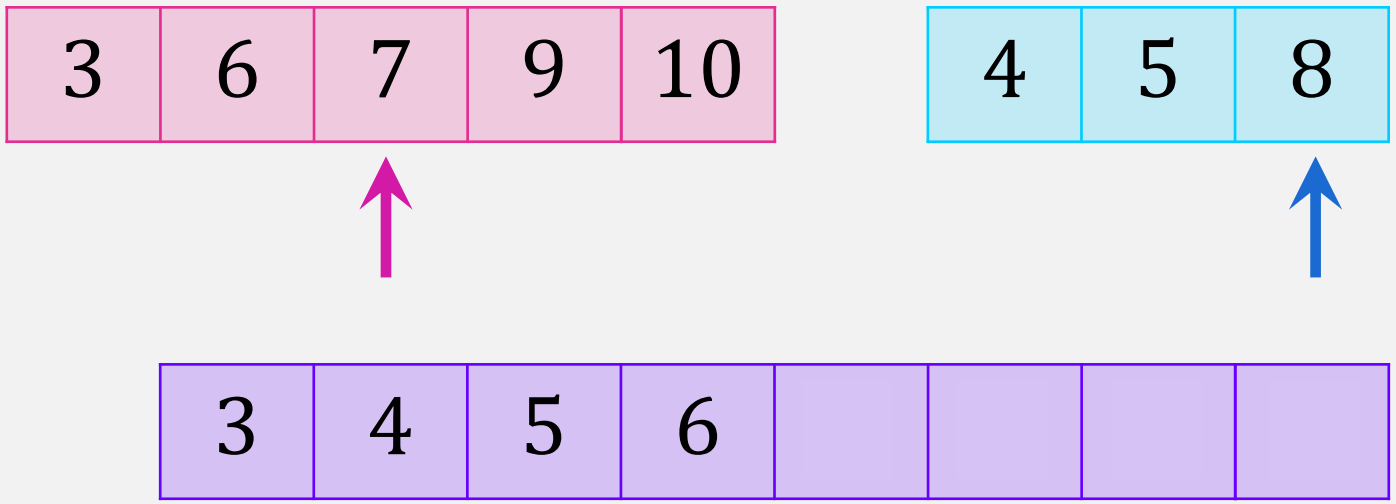


Comparisons:

$$3 < 4$$
$$6 > 4$$
$$6 > 5$$
$$6 < 8$$

Merge

To merge two sorted arrays (or lists) into one sorted list scan them sequentially and perform comparisons



Comparisons:

$3 < 4$

$6 > 4$

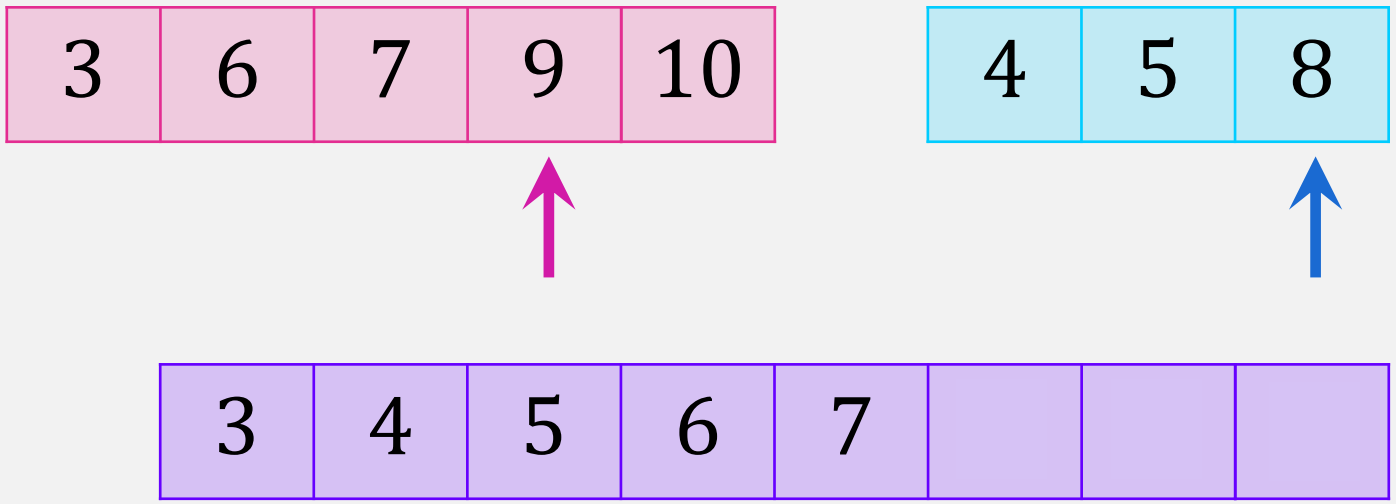
$6 > 5$

$6 < 8$

$7 < 8$

Merge

To merge two sorted arrays (or lists) into one sorted list scan them sequentially and perform comparisons



Comparisons:

$\mathbf{3 < 4}$

$\mathbf{6 > 4}$

$\mathbf{6 > 5}$

$\mathbf{6 < 8}$

$\mathbf{7 < 8}$

$\mathbf{9 > 8}$

Assume that the merged sorted list is of size $\boldsymbol{n}$. How many comparisons are sufficient to merge two sorted lists into one? $\boldsymbol{n - 1}$

Merge

```
<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {
  int i0 = 0, i1 = 0;
  for (int i = 0; i < a.length; i++) {
    if (i0 == a0.length)
      a[i] = a1[i1++];
    else if (i1 == a1.length)
      a[i] = a0[i0++];
    else if (c.compare(a0[i0], a1[i1]) < 0)
      a[i] = a0[i0++];
    else
      a[i] = a1[i1++];
  }
}
```

$O(n)$

Merge-Sort

$T(n)$

$T(n/2)$

$T(n/2)$

$O(n)$

```
<T> void mergeSort(T[] a, Comparator<T> c) {
  if (a.length <= 1) return;
  T[] a0 = Arrays.copyOfRange(a, 0, a.length/2);
  T[] a1 = Arrays.copyOfRange(a, a.length/2, a.length);
  mergeSort(a0, c);
  mergeSort(a1, c);
  merge(a0, a1, a, c);
}
```

MergeSort requires auxiliary array(s).

$T(n)$ – running time of Merge-Sort on an input of n numbers.

Running Time of Merge-Sort

$$T(n) \leq \begin{cases} c & \text{if } n = 1, \\ cn + 2T\left(\frac{n}{2}\right) & \text{if } n \geq 2. \end{cases}$$

for some constant $c > 0$.

$T(n)$ – running time of Merge-Sort on an input of n numbers.

Assume $n = 2^k$, $c = 1$.

$$T(n) \le \begin{cases} c & \text{if } n = 1, \\ cn + 2T\left(\frac{n}{2}\right) & \text{if } n \ge 2. \end{cases}$$

Solve this recurrence by **unfolding**:

$$\begin{aligned}
T(n) &\le n + 2 \cdot T\left(\frac{n}{2}\right) \\
&\le n + 2 \cdot \left[\frac{n}{2} + 2 \cdot T\left(\frac{n}{2^2}\right)\right] \\
&= 2n + 2^2 \cdot T\left(\frac{n}{2^2}\right) \\
&\le 2n + 2^2 \cdot \left[\frac{n}{2^2} + 2 \cdot T\left(\frac{n}{2^3}\right)\right] \\
&= 3n + 2^3 \cdot T\left(\frac{n}{2^3}\right) \\
&\le 3n + 2^3 \cdot \left[\frac{n}{2^3} + 2 \cdot T\left(\frac{n}{2^4}\right)\right] \\
&= 4n + 2^4 \cdot T\left(\frac{n}{2^4}\right) \\
&\cdots \\
&\le kn + 2^k \cdot T\left(\frac{n}{2^k}\right) \\
&= kn + n \cdot T(1) \\
&= kn + n \\
&= n \log n + n \\
&\le 2n \log n
\end{aligned}$$

for general $c > 0$:
$T(n) \le 2cn \log n$

Recursion Tree

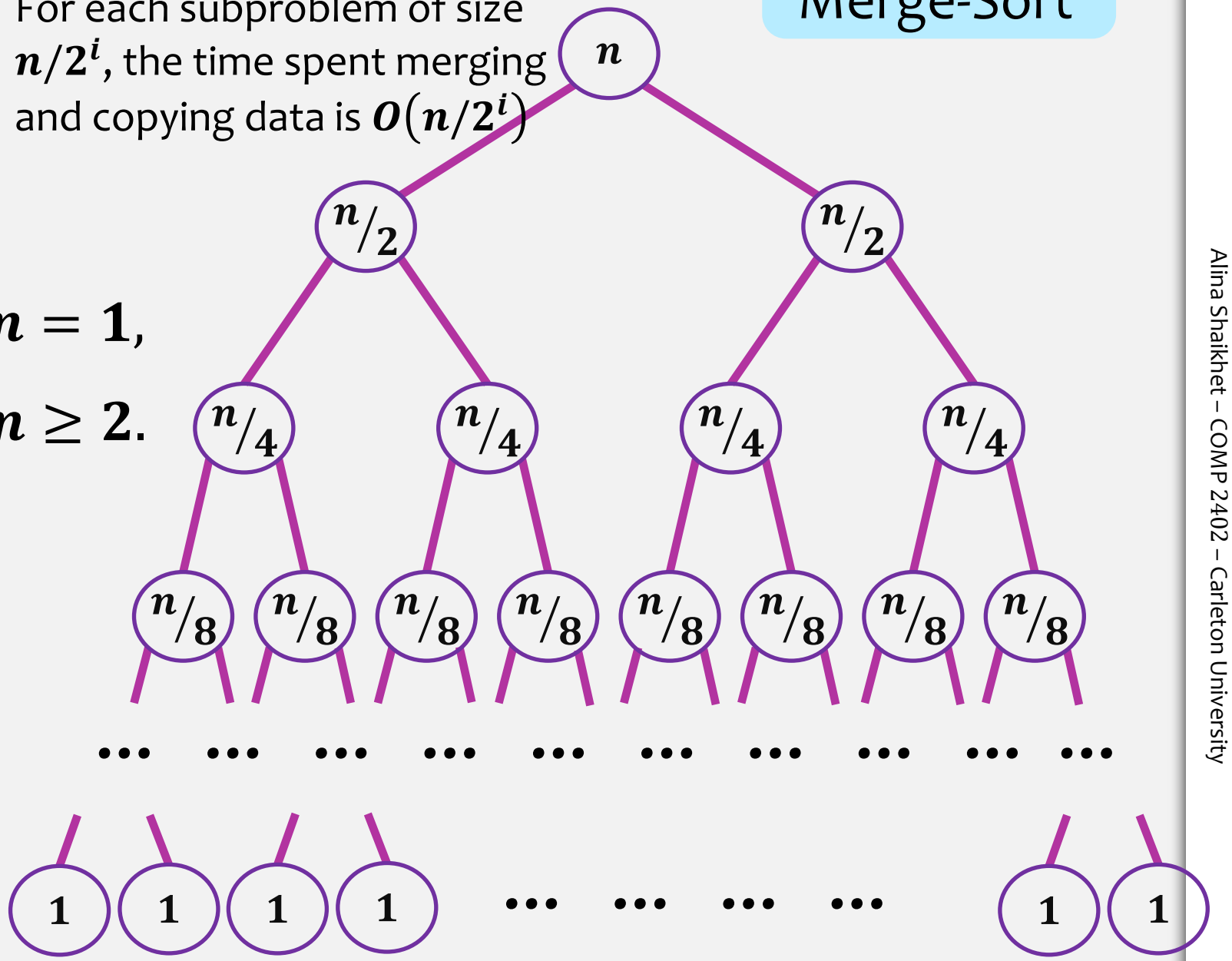
Merge-Sort

For each subproblem of size $n/2^i$, the time spent merging and copying data is $O(n/2^i)$

Assume that:

- $c = 1$
- $n = 2^k$

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ n + 2T\left(\frac{n}{2}\right) & \text{if } n \geq 2. \end{cases}$$



Recursion Tree

For each subproblem of size $n/2^i$, the time spent merging and copying data is $O(n/2^i)$

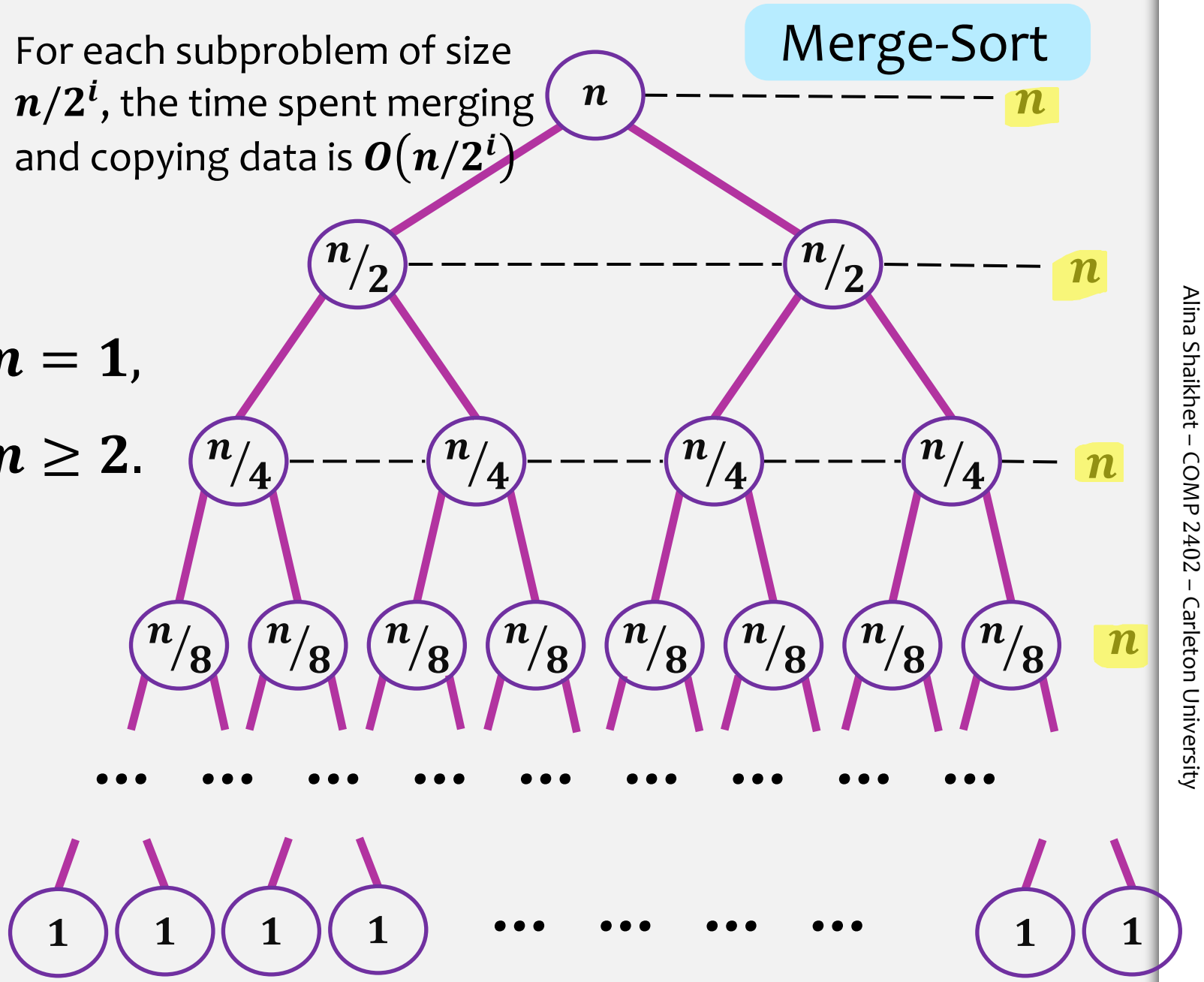
Assume that:
- $c = 1$
- $n = 2^k$

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ n + 2T\left(\frac{n}{2}\right) & \text{if } n \geq 2. \end{cases}$$

Amount of work at each level of the tree: n

Number of levels = recursion depth: $\log_2 n$

$$\therefore T(n) = O(n \log n)$$



Theorem 11.1

Merge-Sort sorts an array of n elements in $O(n \log n)$ **worst-case** time using at most $n \log n$ comparisons.

The proof is by induction on n.

QuickSort

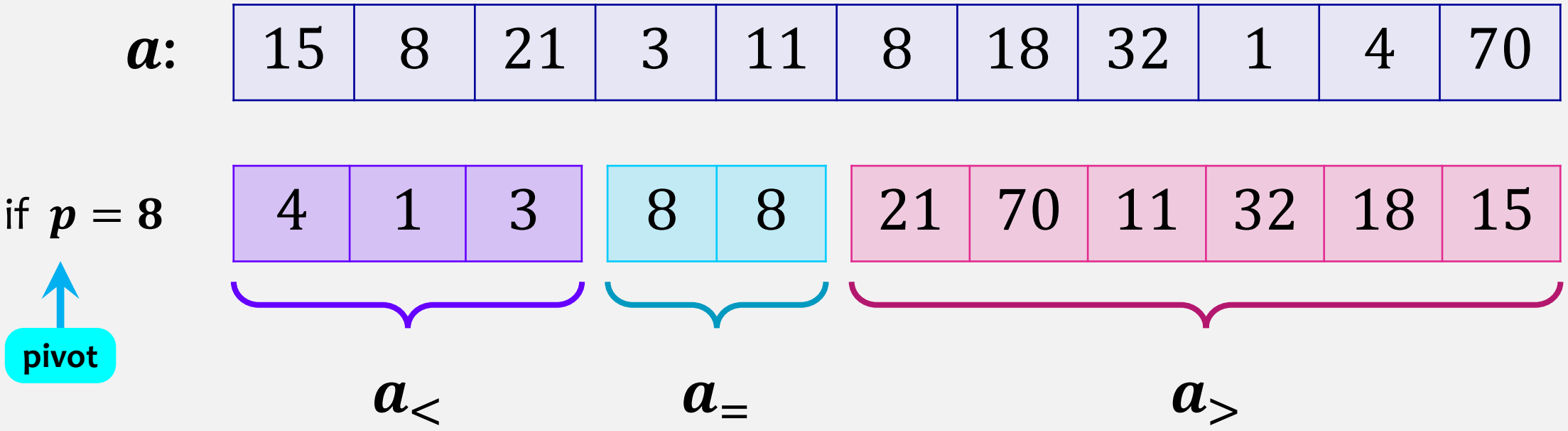
Tony Hoare (1959)

The **QuickSort** algorithm is an **in-place** sorting **divide & conquer** algorithm that is based on a partitioning routine.

To sort n items:

- if $n \leq 1$: nothing to do – just return.
- if $n \geq 2$:
  1. pick a random pivot element x from the list a;
  2. partition a into:
     - the set of elements less than x,
     - the set of elements equal to x, and
     - the set of elements greater than x.
  3. recursively sort the first and third sets in this partition.

QuickSort



If the input to **QuickSort** consists of n distinct elements, then the **QuickSort** recursion tree is a **random binary search tree.**

Theorem 11.3

The **QuickSort** algorithm sorts an array containing $\boldsymbol{n}$ distinct elements in $\boldsymbol{O(n \log n)}$ expected time and the expected number of comparisons it performs is at most $\boldsymbol{2n \ln n + O(n)}$.

$\boldsymbol{1.38n \log_2 n}$