Divide & Conquer

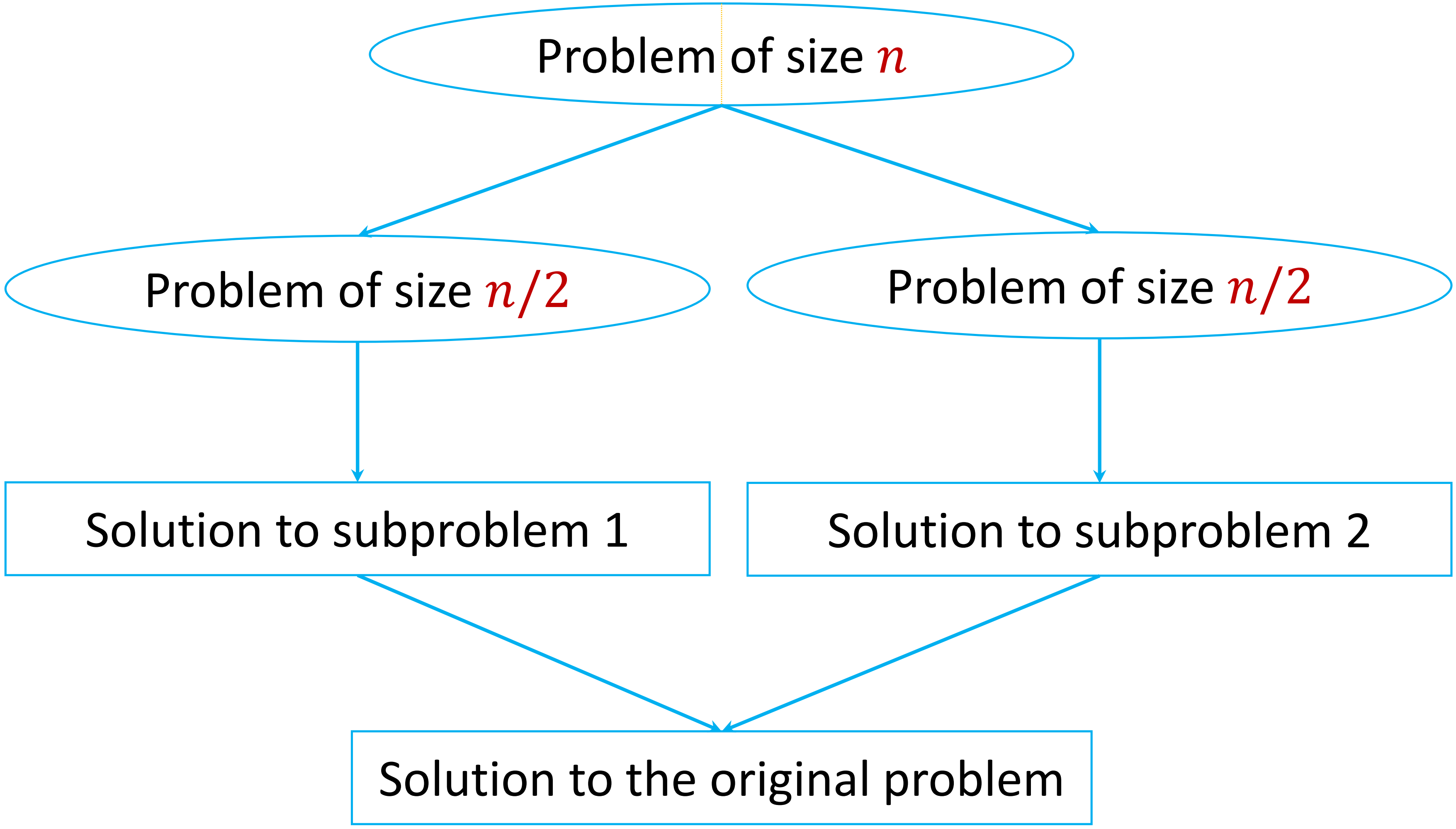
Divide-and-Conquer

1. A problem is **divided** into several subproblems of the same type

2. The subproblems are **solved** (**recursively**)

3. If necessary, the solutions of the subproblems are **combined** to get a solution to the original problem

Divide-and-Conquer



Divide-and-Conquer

- Generally, an instance of size n can be divided into b **instances of size** n/b, with a **of them needing to be solved** where a, b are constants and $a \geq 1, b > 1$.

- Assuming that $n = b^k$, we get the recurrence for the **running time**

$$T(n) = a \cdot T(n/b) + f(n)$$

- $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions

Divide-and-Conquer

- The **order of growth** of $T(n)$ depends on the values of the constants a and b, and the order of growth of $f(n)$

Master Theorem: if $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if} \quad d > \log_b a \\ \Theta(n^d \log n) & \text{if} \quad d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if} \quad d < \log_b a \end{cases}$$

Divide-and-Conquer

Example:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Divide-and-Conquer

Example:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^1$$

$$a = 2$$

$$b = 2$$

$$d = 1$$

Divide-and-Conquer

Example:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^{1}$$

$$a = 2$$

$$b = 2$$

$$d = 1$$

Since $d = \log_b a$, $T(n) = O\left(n^d \log n\right)$

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

Divide-and-Conquer

Example:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Divide-and-Conquer

Example:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^{1}$$

$$a = 4$$

$$b = 2$$

$$d = 1$$

Divide-and-Conquer

Example:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^1$$

$$a = 4$$

$$b = 2$$

$$d = 1$$

Since $d < \log_b a$, $T(n) = O\left(n^{\log_b a}\right)$

$$T(n) = O\left(n^{\log_2 4}\right) = O(n^2)$$

Divide-and-Conquer

Example:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Divide-and-Conquer

Example:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

$$a = 2$$

$$b = 2$$

$$d = 2$$

Divide-and-Conquer

Example:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

$$a = 2$$

$$b = 2$$

$$d = 2$$

Since $d > \log_b a$, $T(n) = O\left(n^d\right)$

$$T(n) = O(n^2)$$

Divide-and-Conquer

Example:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Divide-and-Conquer

Example:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$a = 1$$

$$b = 2$$

$$d = 0$$

Divide-and-Conquer

Example:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$a = 1$$

$$b = 2$$

$$d = 0$$

Since $d = \log_b a$, $T(n) = O\left(n^d \log n\right)$

$$T(n) = O(n^0 \log n) = O(\log n)$$

Mergesort

Mergesort sorts a given array $A[0..n-1]$ by

- **dividing** it into two halves

$$A[0..\lfloor n/2 \rfloor - 1] \quad \text{and} \quad A[\lfloor n/2 \rfloor..n-1]$$

- **sorting** each of them **recursively**, then
- **merging** the two smaller sorted arrays into a single sorted one.

Mergesort

Algorithm MergeSort$(A[0..n-1])$

// Input: $A[0..n-1]$

//Output: $A[0..n-1]$ sorted in nondecreasing order

if $n > 1$

copy $A[0..\lfloor n/2\rfloor - 1]$ to $B[0..\lfloor n/2\rfloor - 1]$

copy $A[\lfloor n/2\rfloor..n-1]$ to $C[0..\lceil n/2\rceil - 1]$

MergeSort$(B[0..\lfloor n/2\rfloor - 1])$

MergeSort$(C[0..\lceil n/2\rceil - 1])$

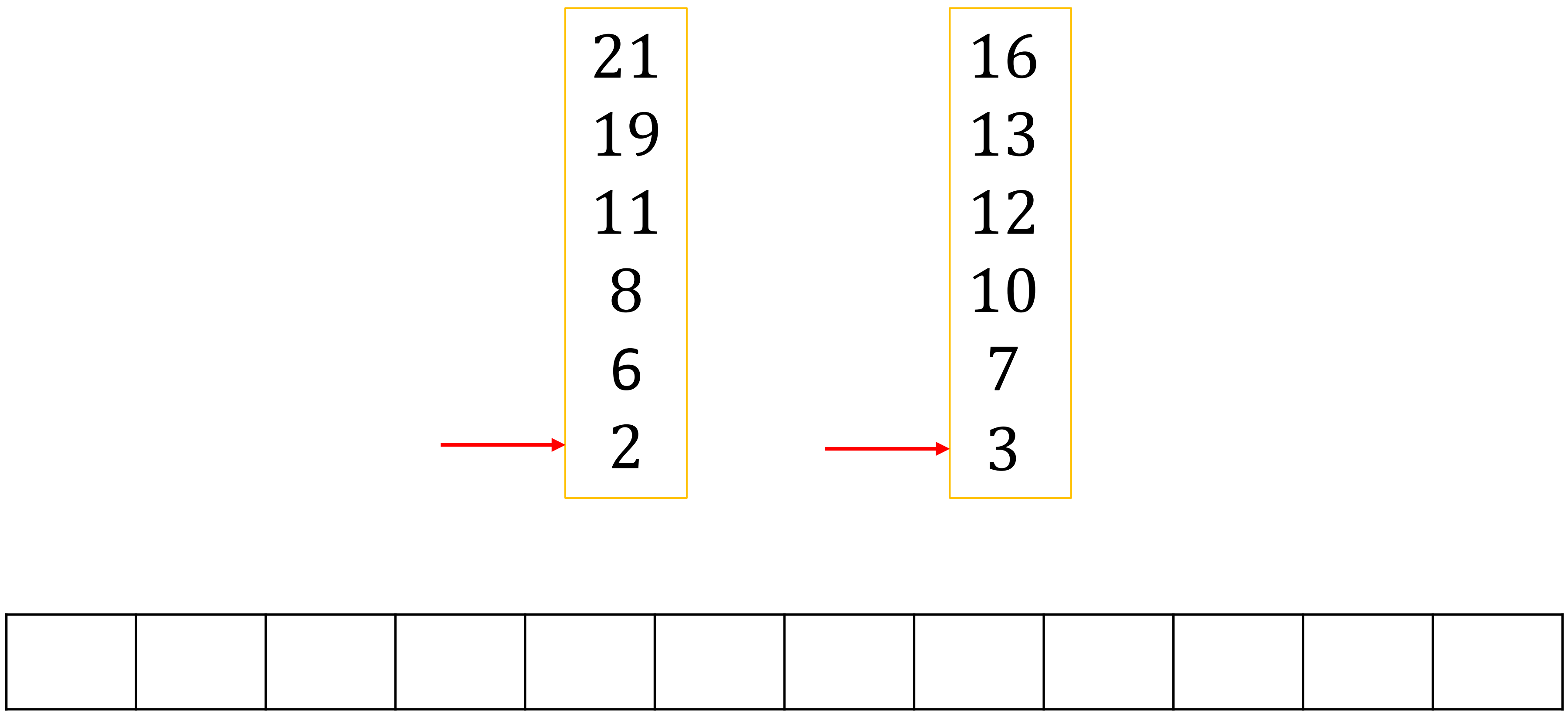
Merge(B, C, A)

Merging of Two Sorted Arrays

- **two pointers** are initialized to point to the **first elements** of the arrays
- the elements pointed to are **compared**, and the smaller of them is added to a new array being constructed
- the index of the smaller element **incremented** to point its immediate successor
- this operation is **repeated** until one of the arrays is exhausted
- the remaining elements of the other array are **copied** to the end of the new array

Merging of Two Sorted Arrays

Example:



Merging of Two Sorted Arrays

Example:

21
19
11
8
6

16
13
12
10
7
3

2											

Merging of Two Sorted Arrays

Example:

21
19
11
8
6

16
13
12
10
7

2	3										

Merging of Two Sorted Arrays

Example:

21
19
11
8

16
13
12
10
7

2	3	6									

Merging of Two Sorted Arrays

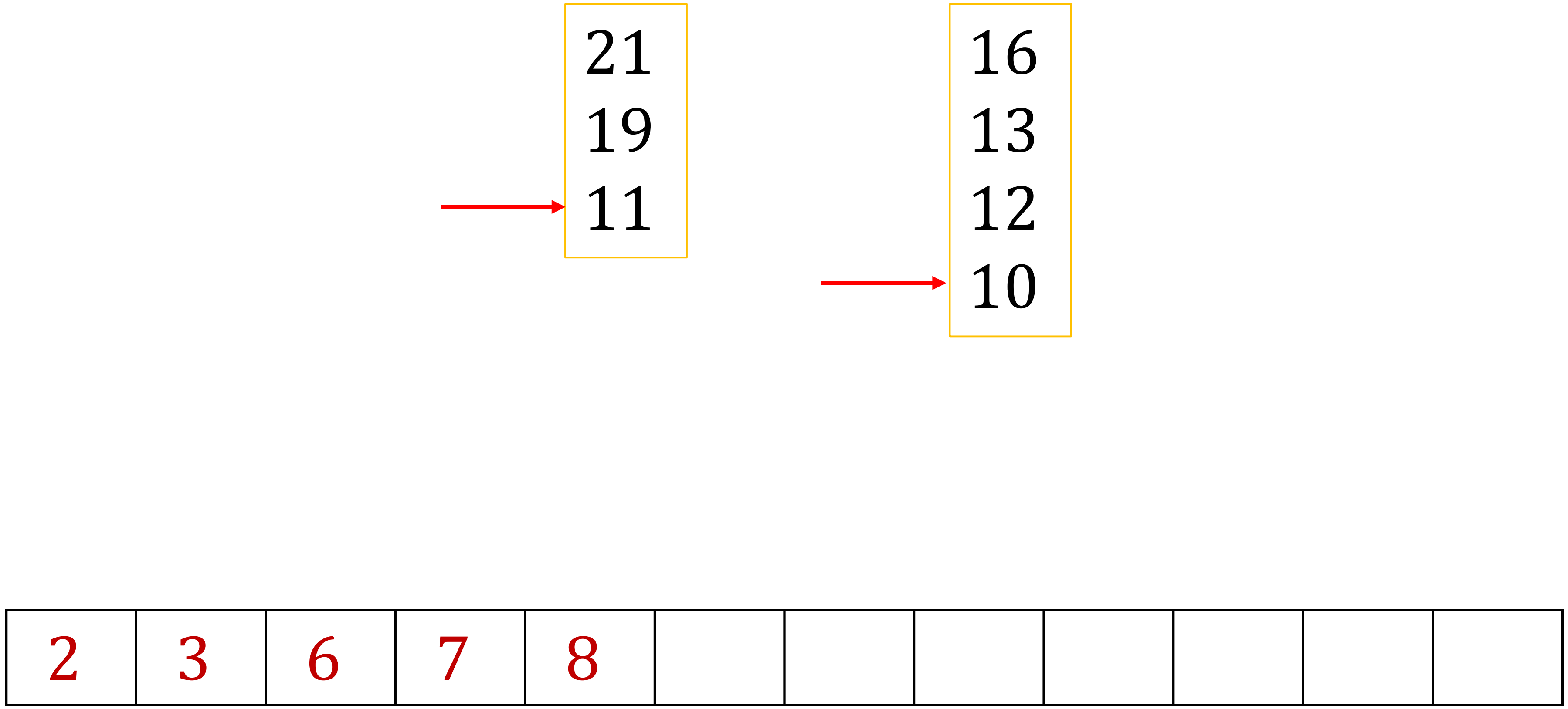
Example:

21	16
19	13
11	12
8	10

2	3	6	7								

Merging of Two Sorted Arrays

Example:



Merging of Two Sorted Arrays

Example:

21	16
19	13
11	12

2	3	6	7	8	10						

Merging of Two Sorted Arrays

Example:

21	16
→ 19	13
	→ 12

2	3	6	7	8	10	11					

Merging of Two Sorted Arrays

Example:

21
19

16
13

2	3	6	7	8	10	11	12				

Merging of Two Sorted Arrays

Example:

21
19

16

2	3	6	7	8	10	11	12	13			

Merging of Two Sorted Arrays

Example:

21
19

2	3	6	7	8	10	11	12	13	16		

Merging of Two Sorted Arrays

Example:

2	3	6	7	8	10	11	12	13	16	19	20

Merge

Algorithm Merge($B[0..p-1], C[0..q-1], A[0..p+q-1]$)

// Input: $B[0..p-1], C[0..q-1]$ sorted

//Output: $A[0..p+q-1]$ sorted

$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

  if $B[i] \leq C[j]$: $A[k] \leftarrow B[i]; i \leftarrow i + 1$

  else $A[k] \leftarrow C[j]; j \leftarrow j + 1$

  $k = k + 1$

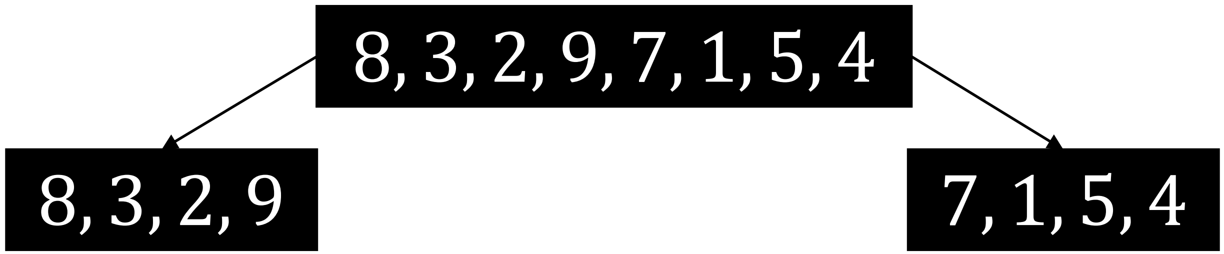
if $i = p$ **copy** $C[j..q-1]$ **to** $A[k..p+q-1]$

else copy $B[i..p-1]$ **to** $A[k..p+q-1]$

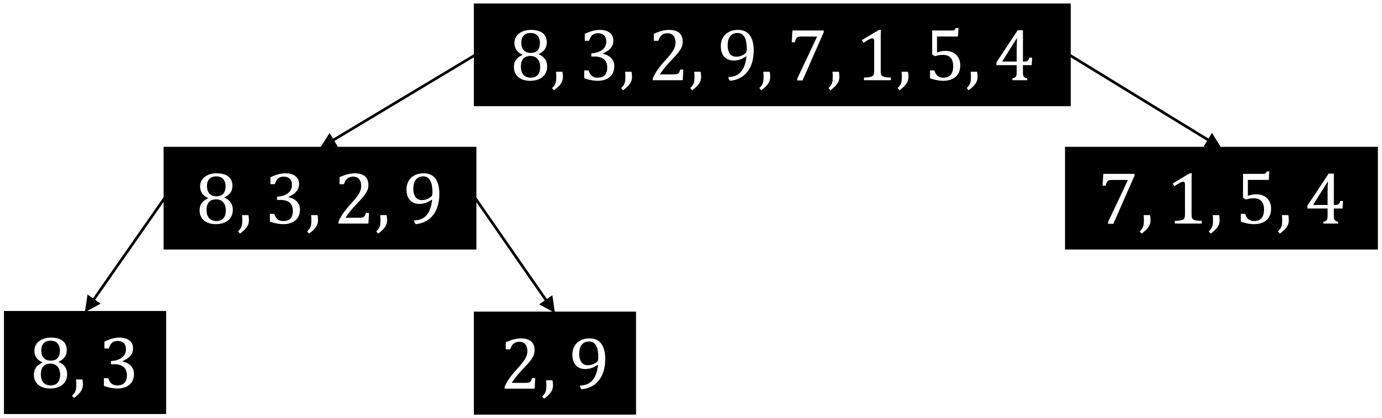
Example: Mergesort

8, 3, 2, 9, 7, 1, 5, 4

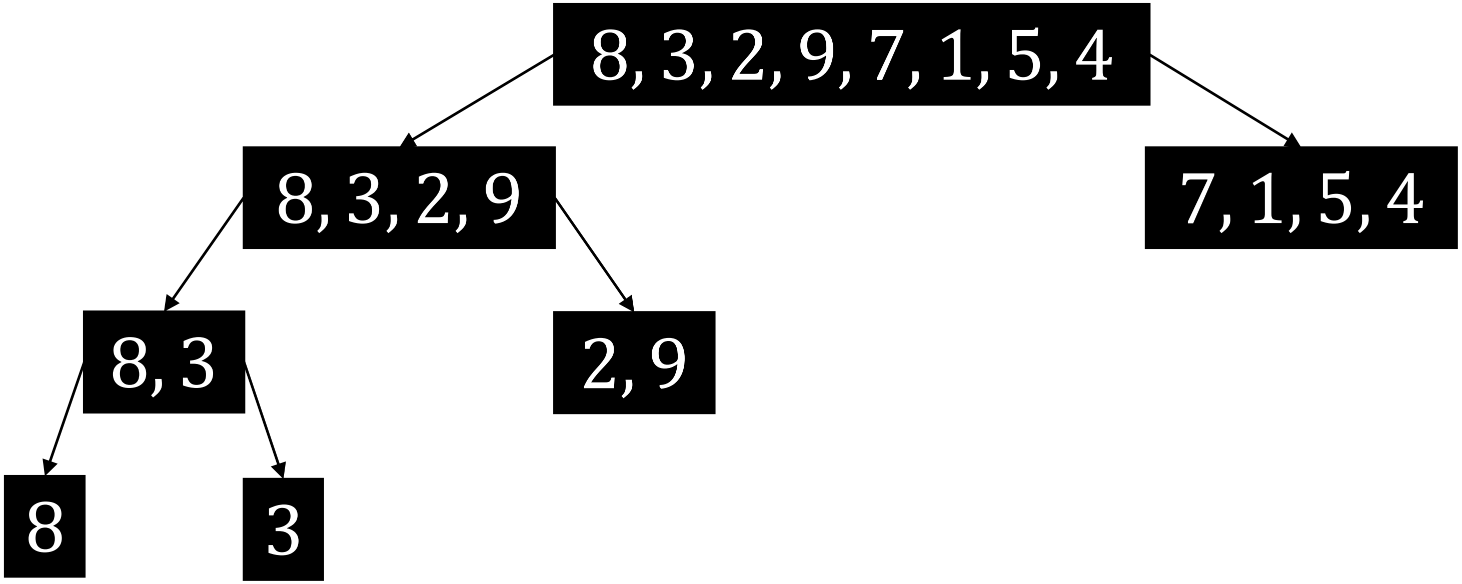
Example: Mergesort



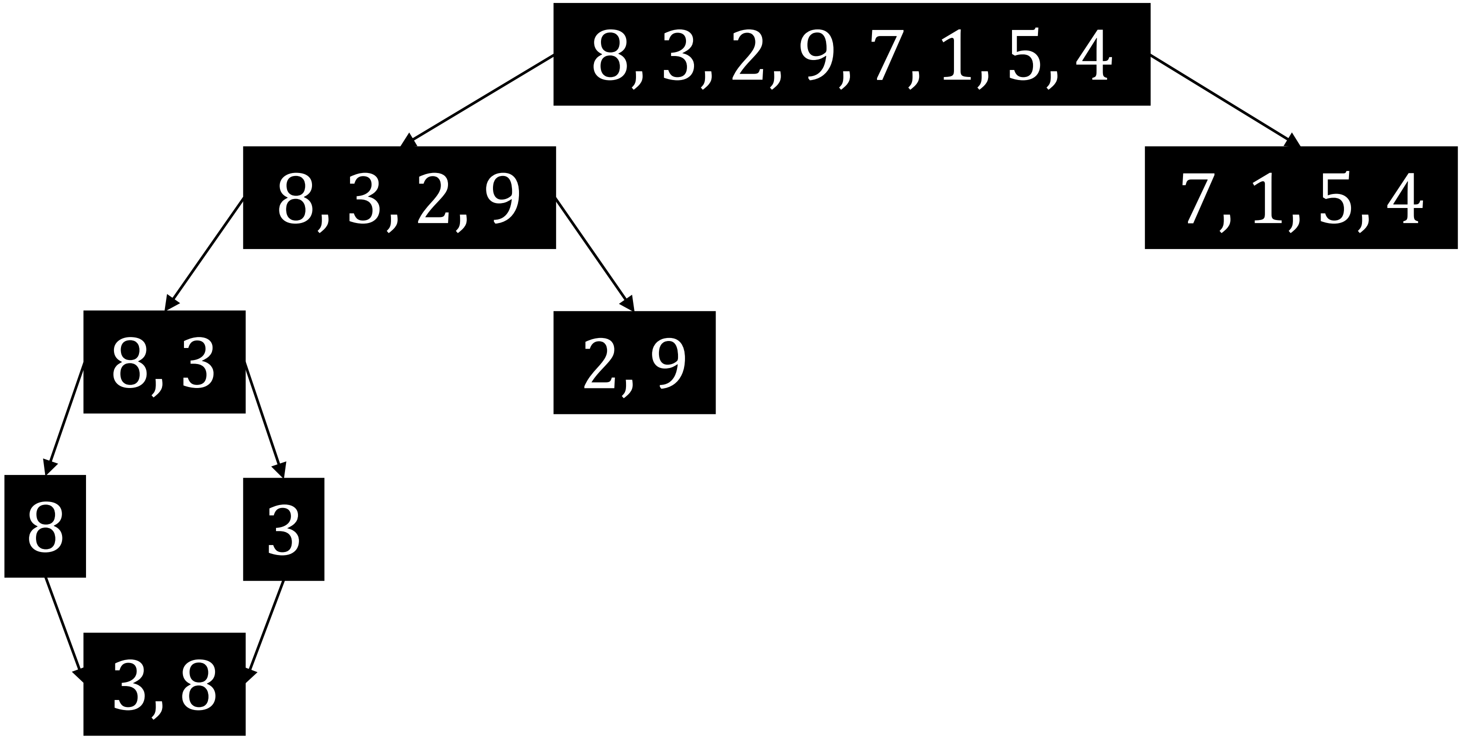
Example: Mergesort



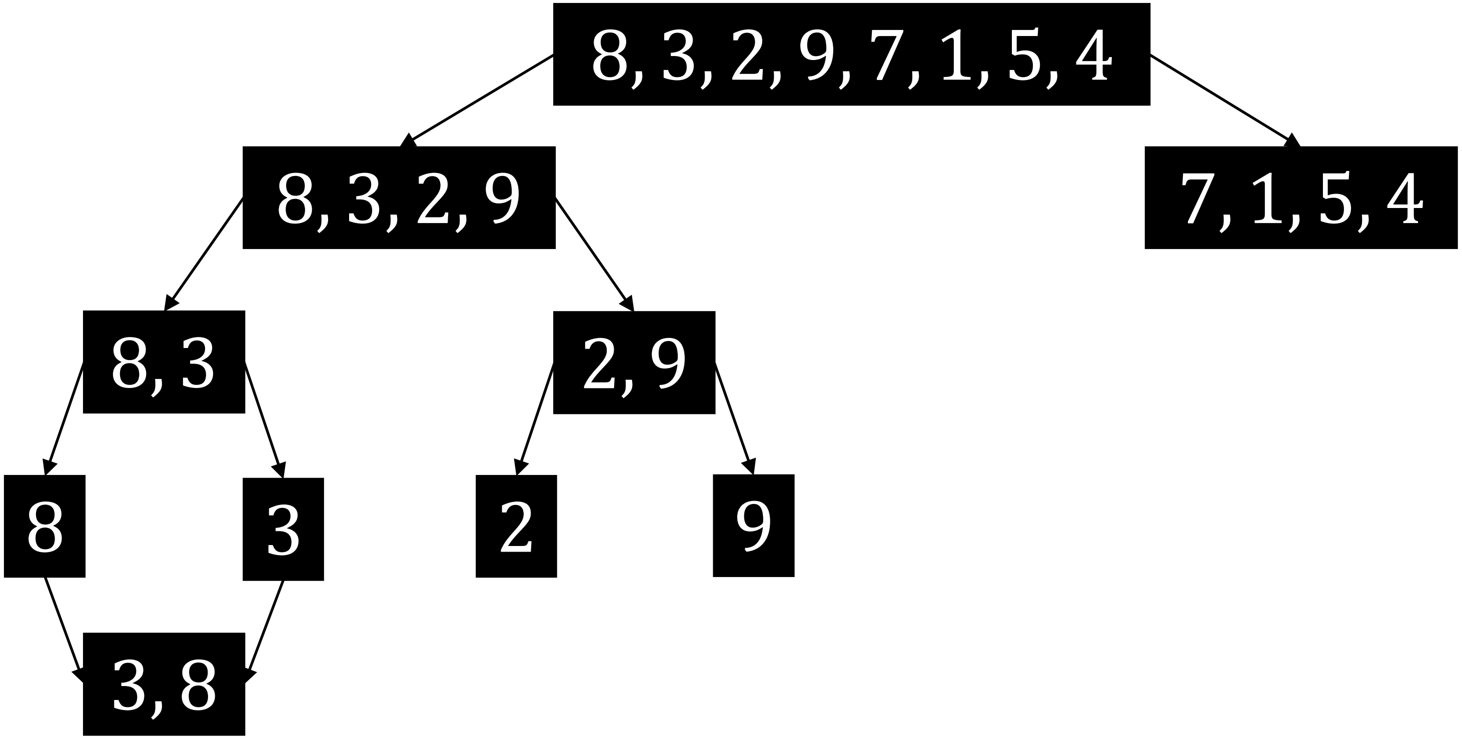
Example: Mergesort



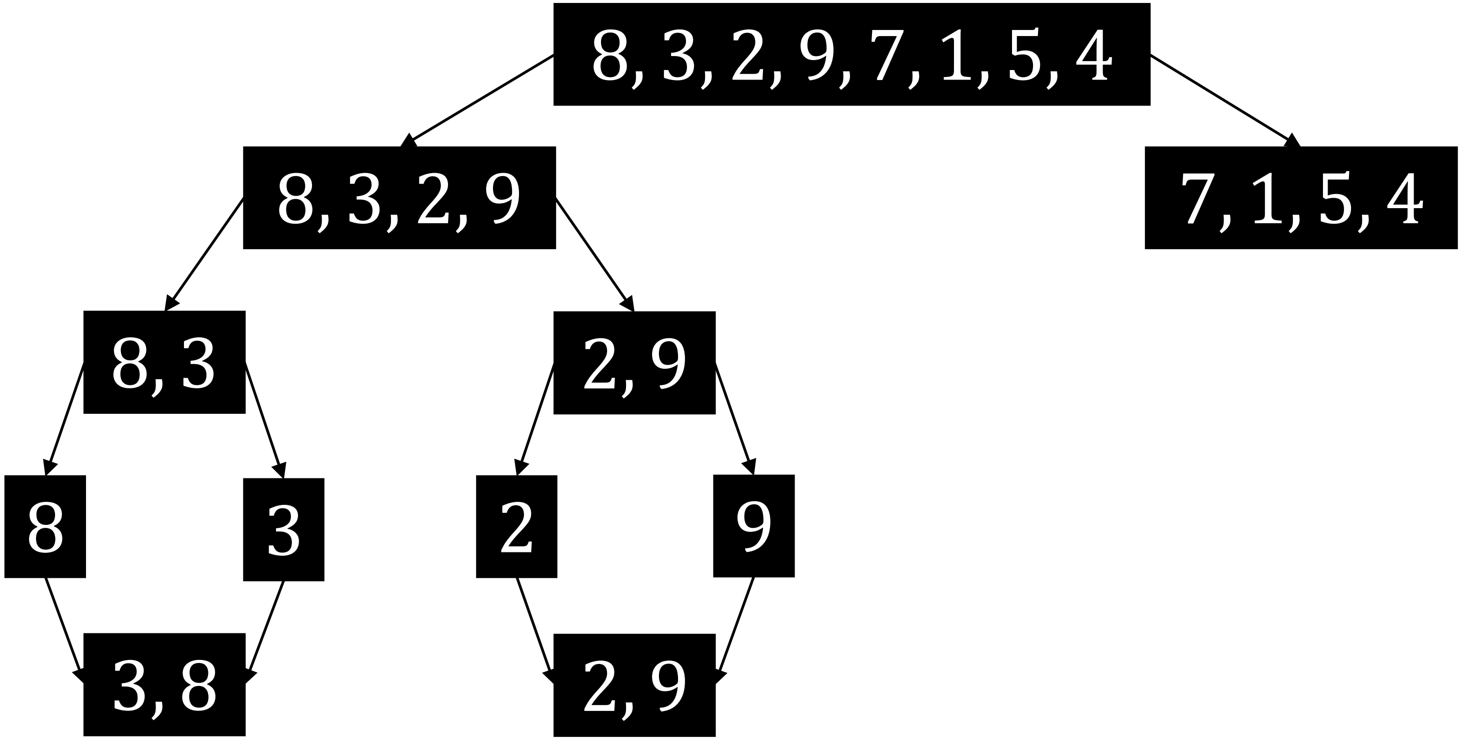
Example: Mergesort



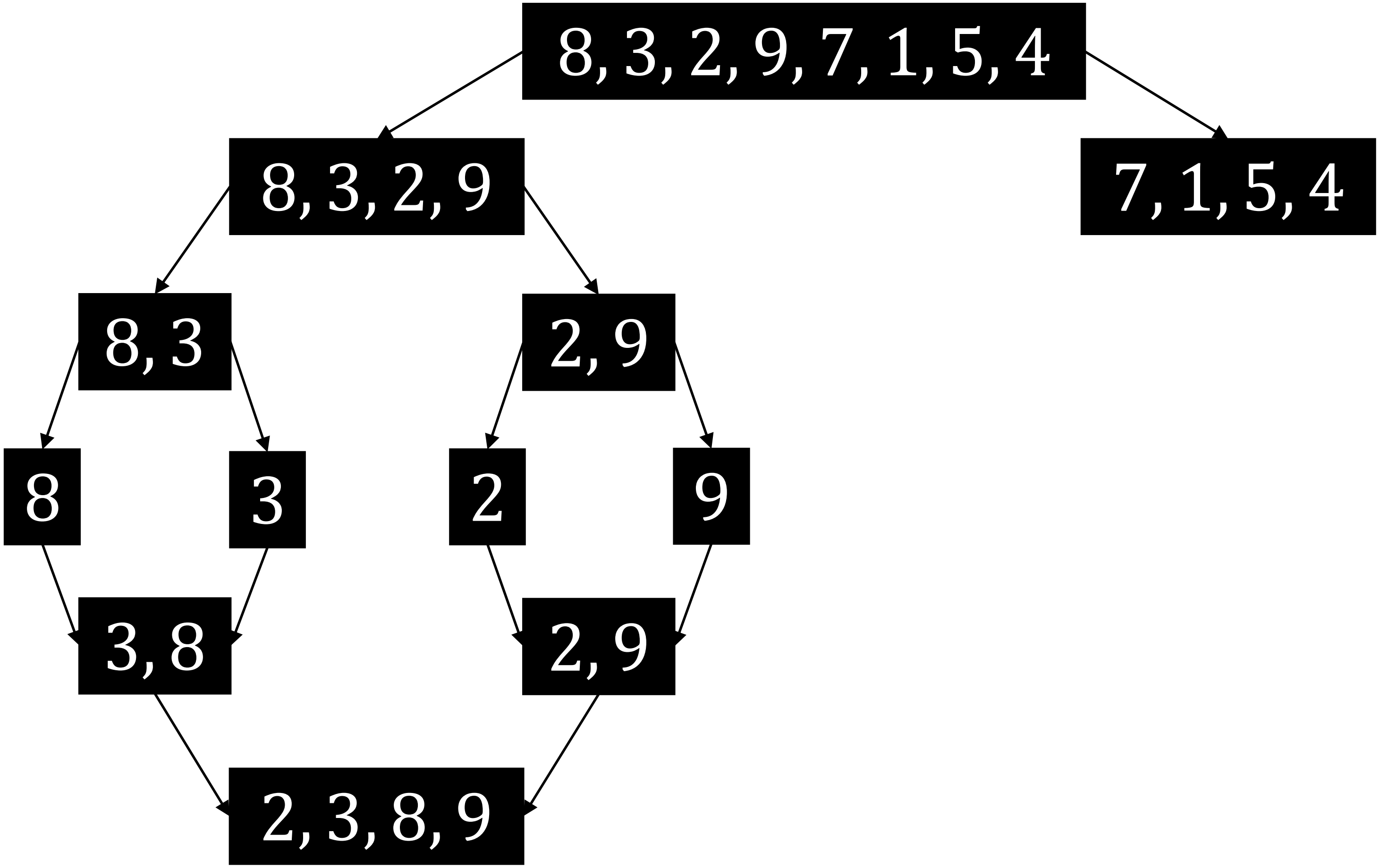
Example: Mergesort



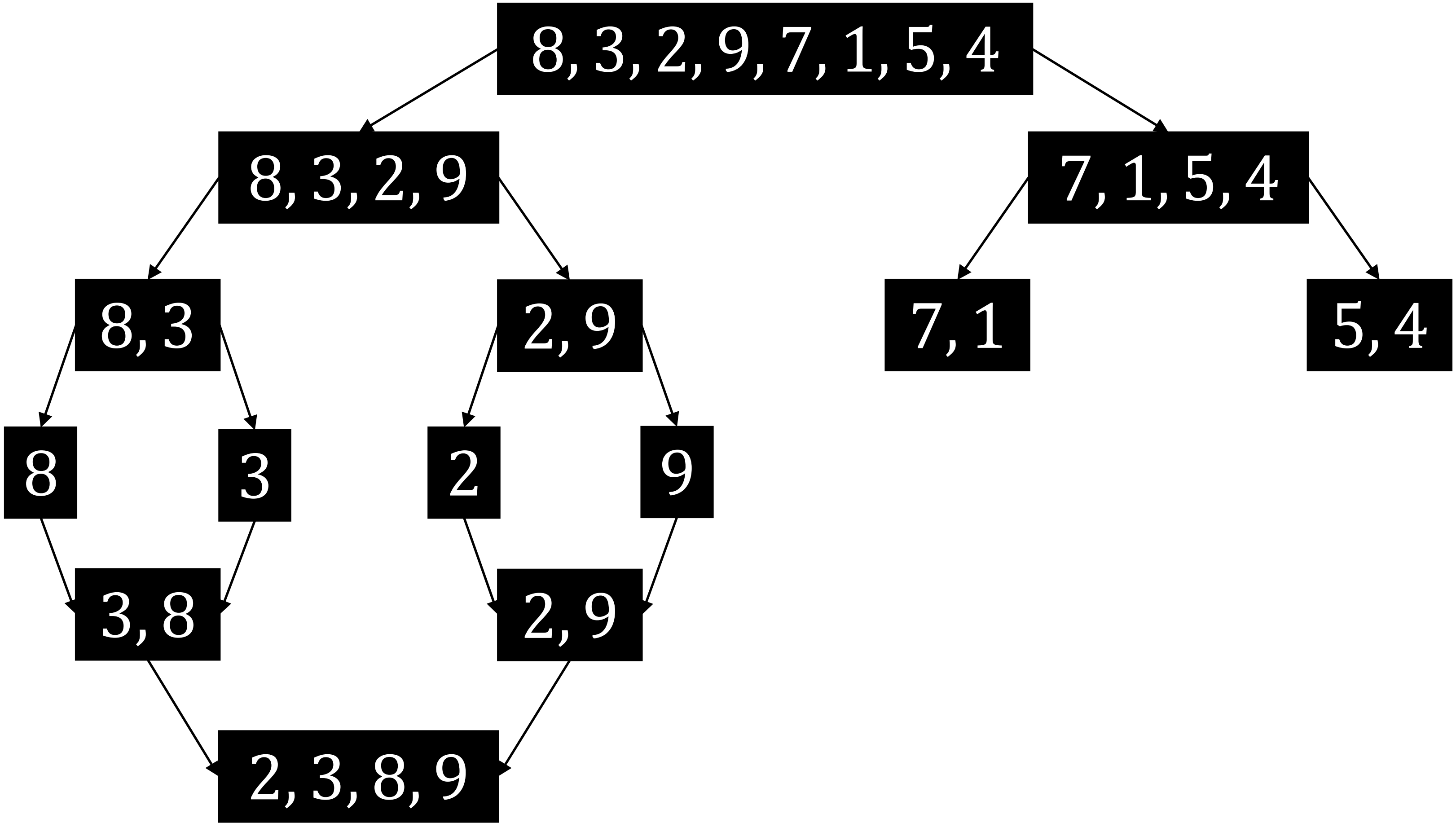
Example: Mergesort



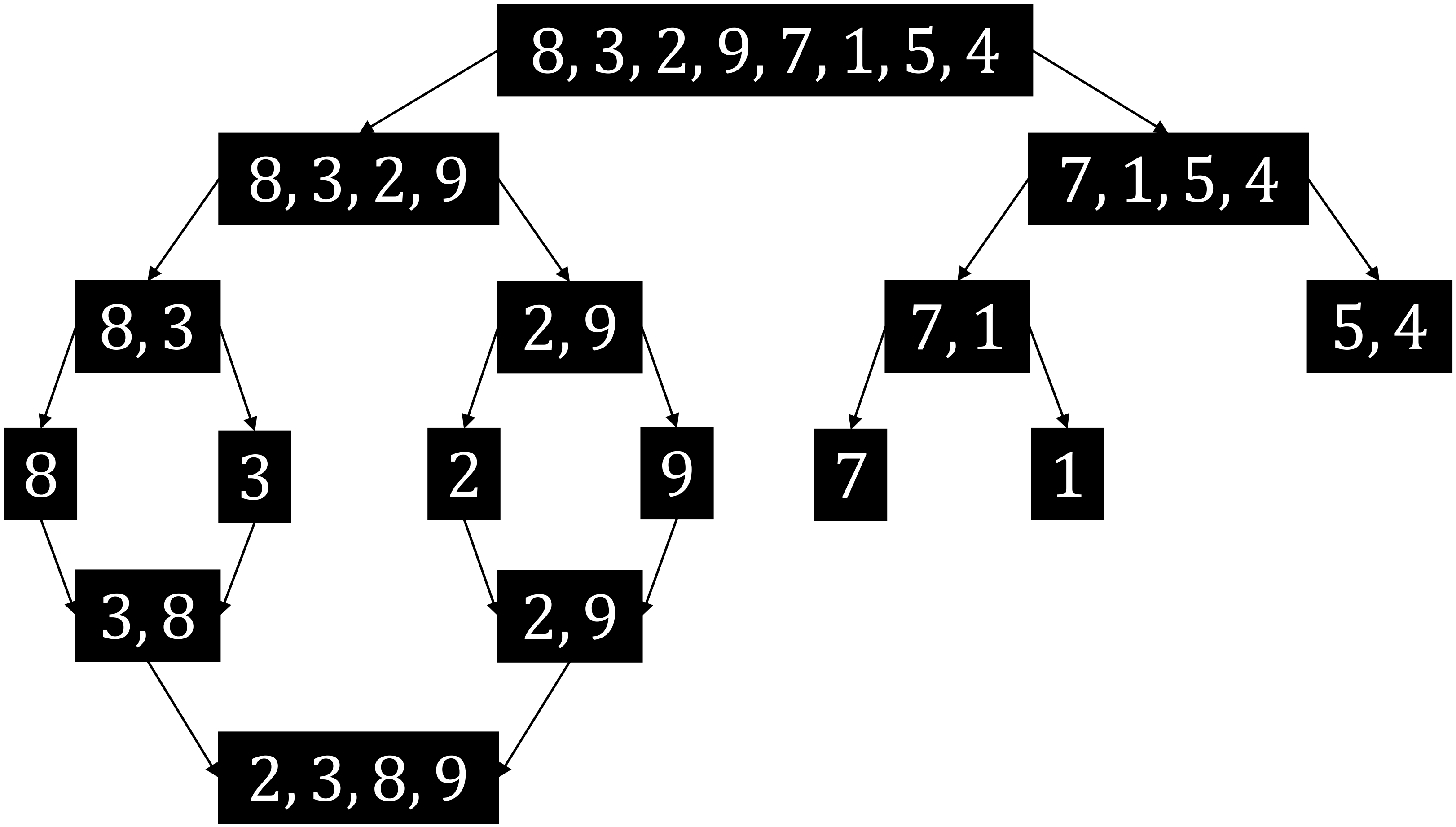
Example: Mergesort



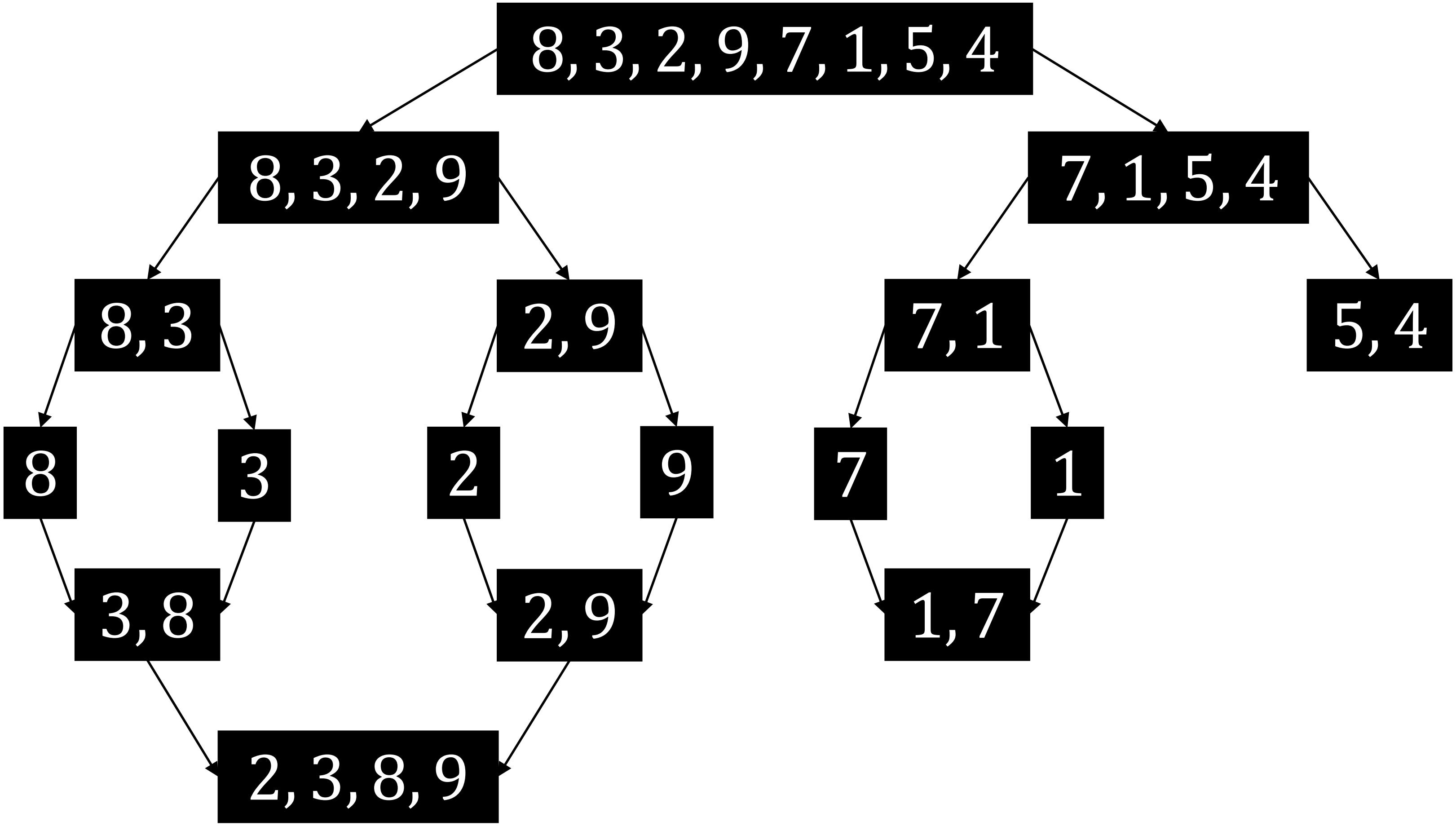
Example: Mergesort



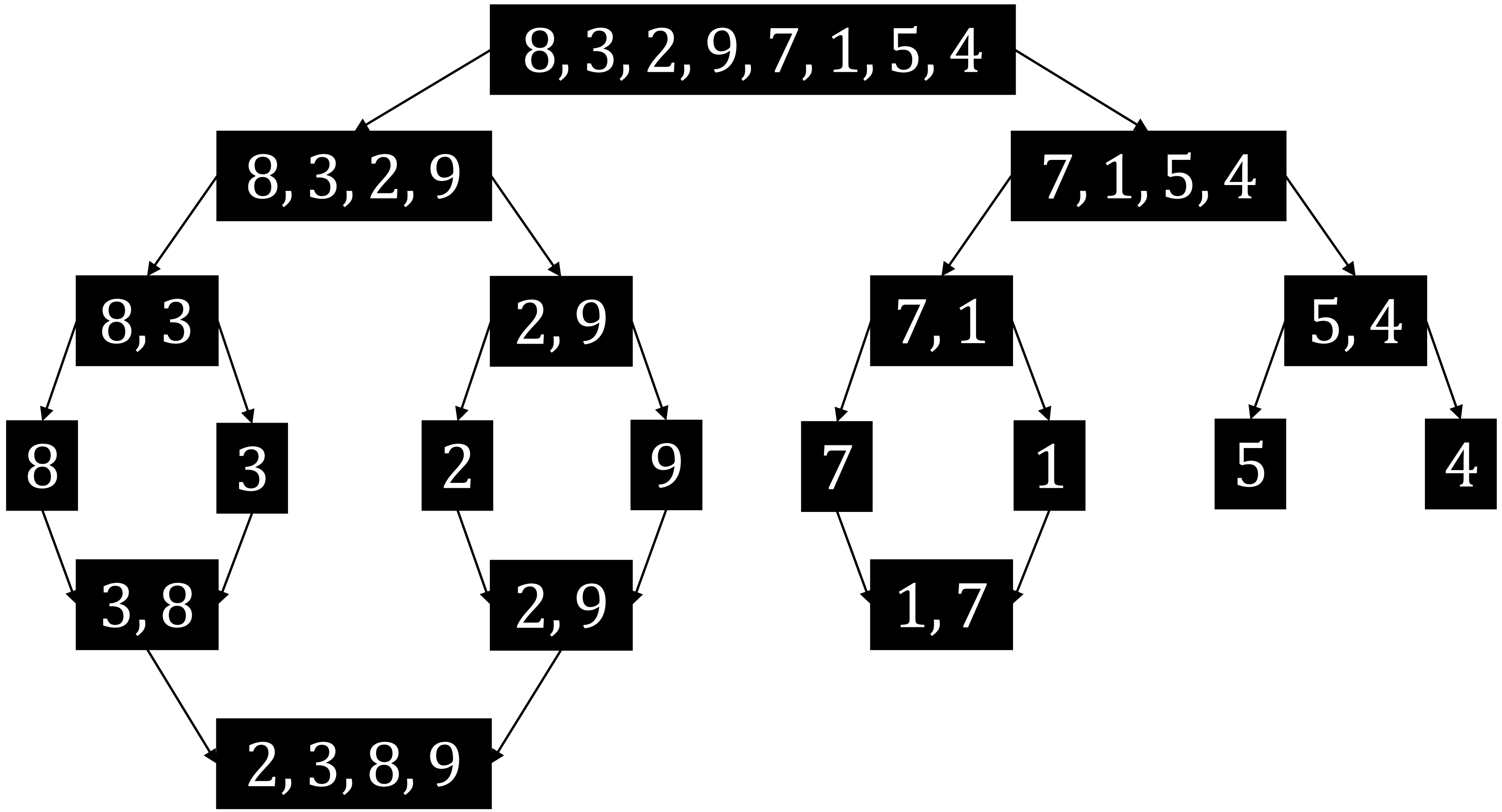
Example: Mergesort



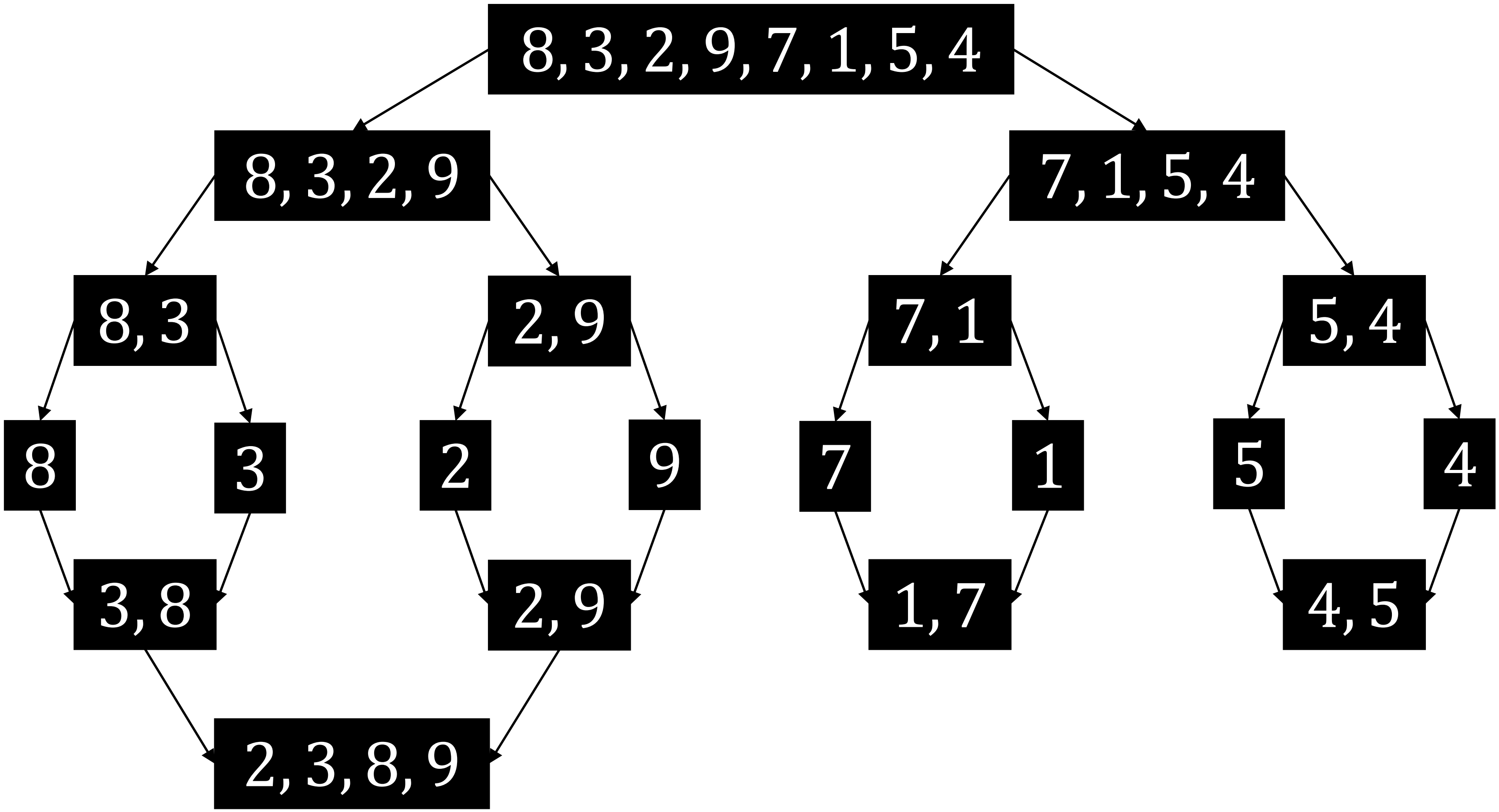
Example: Mergesort



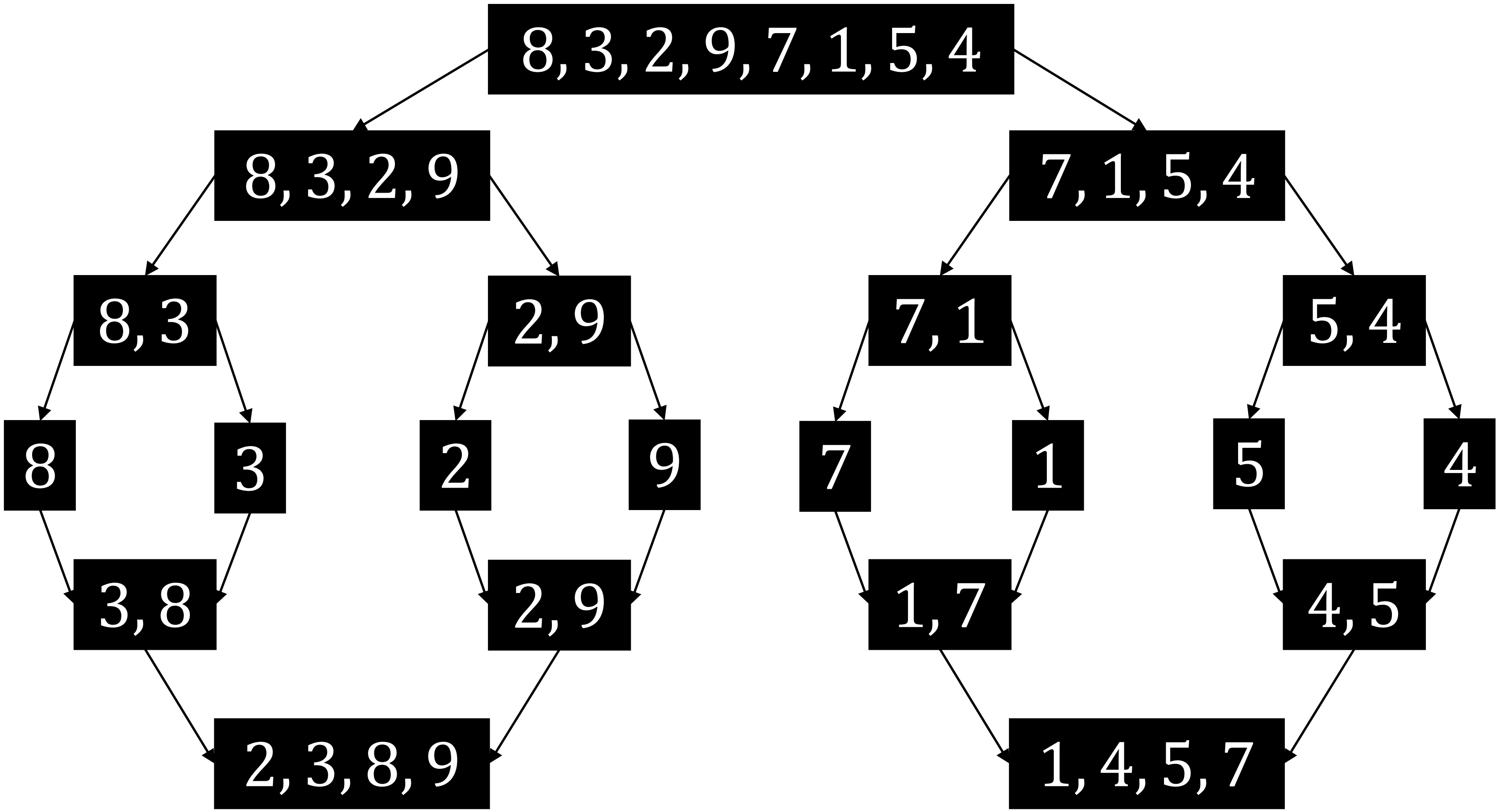
Example: Mergesort



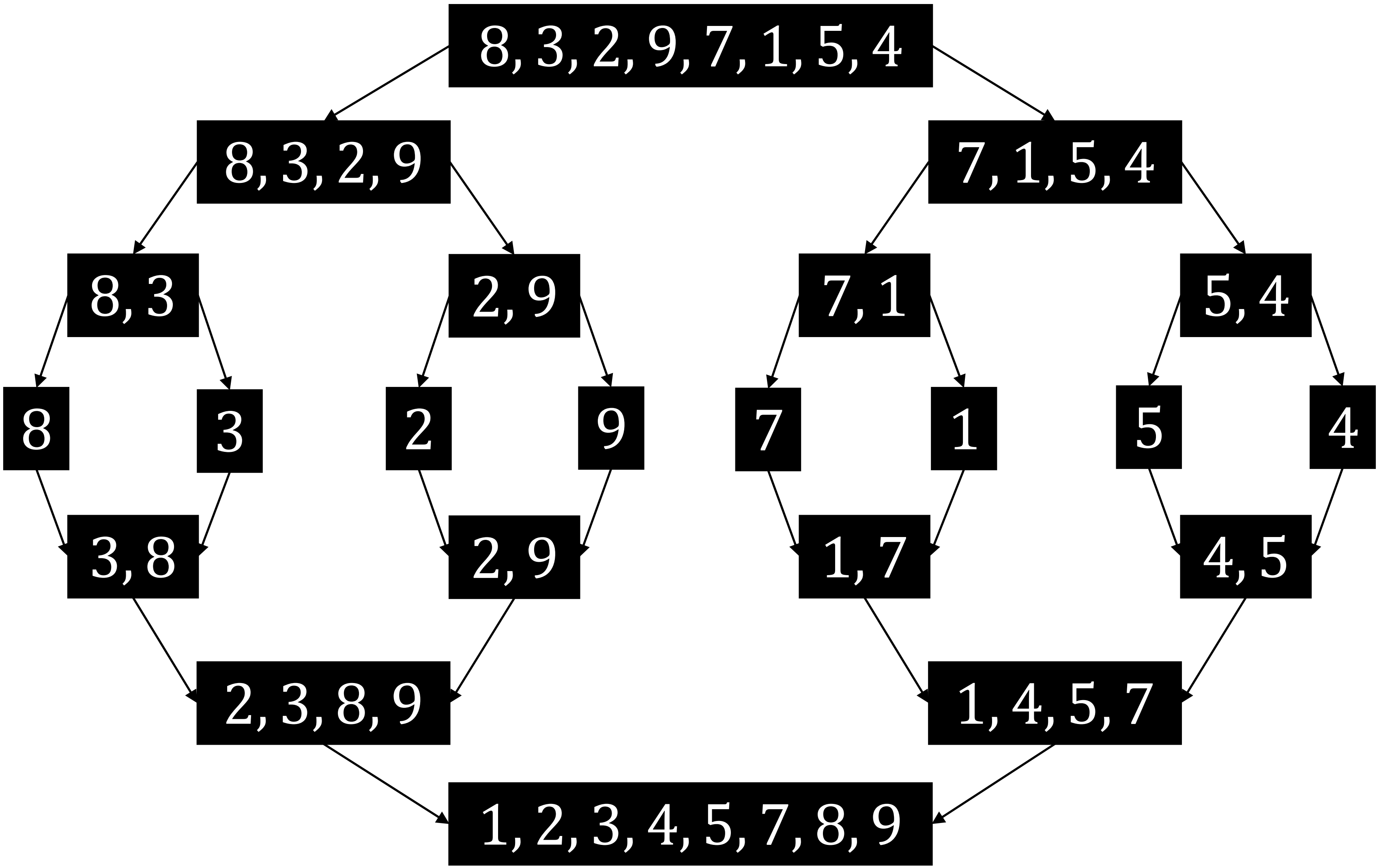
Example: Mergesort



Example: Mergesort



Example: Mergesort



The Efficiency of Mergesort

- The **recurrence relation** for the number of key comparisons:

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for} \quad n > 1$$

$$C(1) = 0$$

- $C_{merge}(n)$ is the number of key comparisons performed during the merge stage

- In the **worst case**: $C_{merge}(n) = n - 1$

$$C_w(n) = 2C_w(n/2) + n - 1 \quad \text{for} \quad n > 1$$

$$C(1) = 0$$

The Efficiency of Mergesort

$a = 2,\ b = 2$ and $d = 1$

$$d = \log_b a$$

$$C_w(n) = \Theta(n^1 \log_2 n) = \Theta(n \log n)$$

Pros: stable (quicksort, heapsort are not stable)

Cons: the algorithm requires the linear time amount of extra memory

Multiplication of Large Integers

Example: Compute $23 \cdot 14$

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0$$

$$14 = 1 \cdot 10^1 + 4 \cdot 10^0$$

Pen-&-pencil algorithm needs n^2 digit multiplications:

$$23 \cdot 14 = (2 \cdot 10^1 + 3 \cdot 10^0) \cdot (1 \cdot 10^1 + 4 \cdot 10^0)$$

$$= (2 \cdot 1) \cdot 10^2 + (2 \cdot 4 + 3 \cdot 1) \cdot 10^1$$

$$+(3 \cdot 4) \cdot 10^0$$

Multiplication of Large Integers

But

$$2 \cdot 4 + 3 \cdot 1 = (2 + 3) \cdot (1 + 4) - 2 \cdot 1 - 3 \cdot 4$$

The number of multiplications are 3 now!!!

$$23 \cdot 14 = (2 \cdot 10^1 + 3 \cdot 10^0) \cdot (1 \cdot 10^1 + 4 \cdot 10^0)$$

$$= (\mathrm{m}_1) \cdot 10^2 + (\mathrm{m}_3) \cdot 10^1 + (\mathrm{m}_2) \cdot 10^0$$

- $\mathrm{m}_1 = 2 \cdot 1$
- $\mathrm{m}_2 = 3 \cdot 4$
- $\mathrm{m}_3 = (2 + 3) \cdot (1 + 4) - 2 \cdot 1 - 3 \cdot 4$

Multiplication of Large Integers

In general: $a = a_1 a_0$ and $b = b_1 b_0$, then

$$c = a \cdot b = c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0$$

where

- $c_2 = a_1 \cdot b_1$
- $c_0 = a_0 \cdot b_0$
- $c_1 = (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0)$

Multiplication of Large Integers

- Let a and b are n-digit integers where n is an **even** number
- Let $a = a_1 a_0$ and $b = b_1 b_0$, where the first halves of a and b is are denoted by a_1 and b_1, and the second halves are denoted by a_0 and b_0, i.e.,

$$a = a_1 a_0 = a_1 \cdot 10^{n/2} + a_0$$

$$b = b_1 b_0 = b_1 \cdot 10^{n/2} + b_0$$

Multiplication of Large Integers

$$c = a \cdot b = \left(a_1 \cdot 10^{n/2} + a_0\right)\left(b_1 \cdot 10^{n/2} + b_0\right)$$

$$= (a_1 b_1) \cdot 10^{n} + (a_1 b_0 + a_0 b_1) \cdot 10^{n/2} + (a_0 b_0)$$

$$= m_2 \cdot 10^{n} + m_1 \cdot 10^{n/2} + m_0$$

where

$$m_2 = a_1 \cdot b_1$$

$$m_0 = a_0 \cdot b_0$$

$$m_1 = (a_1 + a_0) \cdot (b_1 + b_0) - (m_2 + m_0)$$

Multiplication of Large Integers

Time efficiency: $M(n)$ - the total # of multiplications

$$M(n) = 3M(n/2), n > 1$$

$$M(1) = 1$$

Multiplication of Large Integers

Time efficiency: $M(n)$ - the total # of multiplications

$$M(n) = 3M(n/2), n > 1$$

$$M(1) = 1$$

Backward substitutions: $n = 2^k,\ (k = \log_2 n)$

$$M(2^k) = 3M(2^{k-1}) = 3\left(3M(2^{k-2})\right) = 3^2M(2^{k-2})$$
$$= \cdots = 3^iM(2^{k-i}) = \cdots = 3^kM(2^{k-k})$$
$$= 3^kM(1) = 3^k$$

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$$

Strassen's Matrix Multiplication

- Let $\mathbf{A}$ and $\mathbf{B}$ be (2×2)-matrices. Then $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ requires 8 multiplications (brute-force approach).

- Strassen's algorithm reduces the number of multiplications to 7.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Strassen's Matrix Multiplication

$$m_1 = (a_{00} + a_{11}) \cdot (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) \cdot b_{00}$$

$$m_3 = a_{00} \cdot (b_{01} - b_{11})$$

$$m_4 = a_{11} \cdot (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) \cdot b_{11}$$

$$m_6 = (a_{10} - a_{00}) \cdot (b_{00} + b_{01})$$

$$m_7 = (a_{01} - a_{11}) \cdot (b_{10} + b_{11})$$

Strassen's Matrix Multiplication

- Let $\mathbf{A}$ and $\mathbf{B}$ be $(\mathrm{n} \times \mathrm{n})$-matrices. Let $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$

$$\begin{bmatrix} \mathbf{C}_{00} & \mathbf{C}_{01} \\ \mathbf{C}_{10} & \mathbf{C}_{11} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{00} & \mathbf{A}_{01} \\ \mathbf{A}_{10} & \mathbf{A}_{11} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B}_{00} & \mathbf{B}_{01} \\ \mathbf{B}_{10} & \mathbf{B}_{11} \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 & \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{M}_2 + \mathbf{M}_4 & \mathbf{M}_1 + \mathbf{M}_3 - \mathbf{M}_2 + \mathbf{M}_6 \end{bmatrix}$$

Strassen's Matrix Multiplication

$$\mathbf{M}_1 = (\mathbf{A}_{00} + \mathbf{A}_{11}) \cdot (\mathbf{B}_{00} + \mathbf{B}_{11})$$

$$\mathbf{M}_2 = (\mathbf{A}_{10} + \mathbf{A}_{11}) \cdot \mathbf{B}_{00}$$

$$\mathbf{M}_3 = \mathbf{A}_{00} \cdot (\mathbf{B}_{01} - \mathbf{B}_{11})$$

$$\mathbf{M}_4 = \mathbf{A}_{11} \cdot (\mathbf{B}_{10} - \mathbf{B}_{00})$$

$$\mathbf{M}_5 = (\mathbf{A}_{00} + \mathbf{A}_{01}) \cdot \mathbf{B}_{11}$$

$$\mathbf{M}_6 = (\mathbf{A}_{10} - \mathbf{A}_{00}) \cdot (\mathbf{B}_{00} + \mathbf{B}_{01})$$

$$\mathbf{M}_7 = (\mathbf{A}_{01} - \mathbf{A}_{11}) \cdot (\mathbf{B}_{10} + \mathbf{B}_{11})$$

Strassen's Matrix Multiplication

Time efficiency: $\mathrm{M(n)}$ is the total # of multiplications made by Strassen's algorithm

$$\mathrm{M(n)} = 7\mathrm{M(n/2)}, \mathrm{n} > 1$$

$$\mathrm{M(1)} = 1$$

Strassen's Matrix Multiplication

Time efficiency: $\mathrm{M(n)}$ is the total # of multiplications made by Strassen's algorithm

$$\mathrm{M(n) = 7M(n/2), n > 1}$$

$$\mathrm{M(1) = 1}$$

Backward substitutions: $\mathrm{n = 2^k,\ (k = \log_2 n)}$

$$\mathrm{M}\left(2^{\mathrm{k}}\right) = 7\mathrm{M}\left(2^{\mathrm{k-1}}\right) = 7\left(7\mathrm{M}\left(2^{\mathrm{k-2}}\right)\right) = 7^2\mathrm{M}\left(2^{\mathrm{k-2}}\right)$$

$$= \cdots = 7^{\mathrm{i}}\mathrm{M}\left(2^{\mathrm{k-i}}\right) = \cdots = 7^{\mathrm{k}}\mathrm{M}\left(2^{\mathrm{k-k}}\right)$$

$$= 7^{\mathrm{k}}\mathrm{M(1)} = 3^{\mathrm{k}}$$

$$\mathrm{M(n)} = 7^{\log_2 \mathrm{n}} = \mathrm{n}^{\log_2 7} \approx \mathrm{n}^{2.807}$$