CSC 261/461 – Database Systems
Lecture 16

Spring 2018

The IO Model & External Sorting

Today's Lecture

- Chapter 16 (Disk Storage, File Structure and Hashing)
- Chapter 17 (Indexing)

This chapters cover a lot of details and it's not possible to cover everything in class.
So please study as much as you can

Sections to study

Chapter 16

- 16.1
- 16.2 (16.2.1, 16.2.2)
- 16.3
- 16.4
- 16.5
- 16.6
- 16.7
- 16.8 (Skip 16.8.3)

Chapter 17

- 17.1
- 17.2
- 17.3 (17.3.1, 17.3.2)

(Note: You need to study sections marked in red thoroughly. Other sections are important for comprehension. The sections those are not included can be skipped (but not recommended))

Simplified Database System Environment

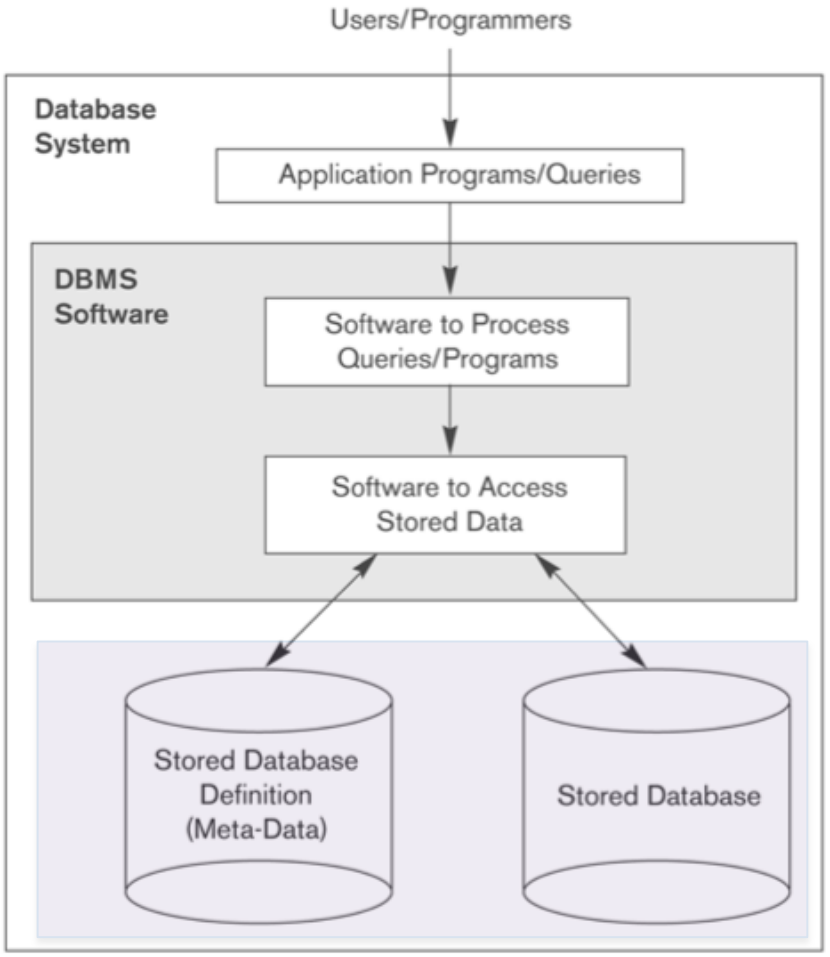


Figure 1.1
A simplified database system environment.

What you will learn about in this section

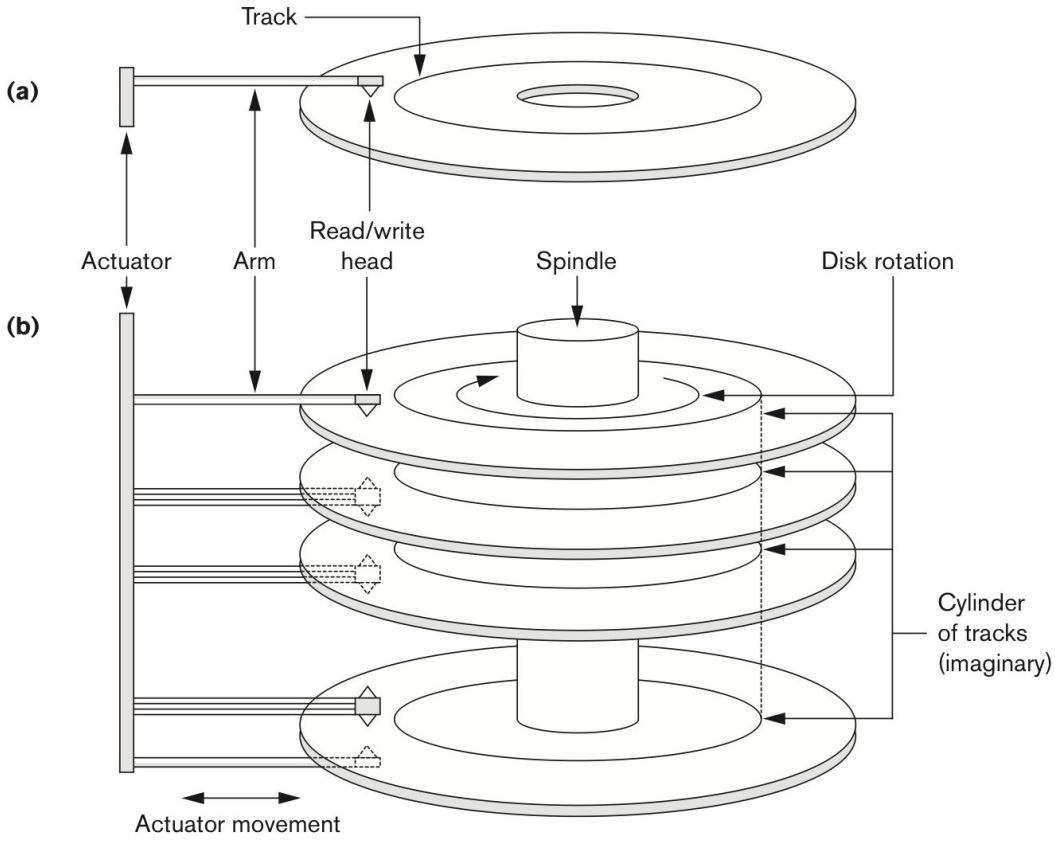
1. Storage and memory model

2. Buffer

1. THE BUFFER

High-level: Disk vs. Main Memory

- **Disk**:
  - Slow
    - Sequential access
      - (although fast sequential reads)
  - Durable
    - We will assume that once on disk, data is safe!
  - Cheap



High-level: Disk vs. Main Memory

- Random Access Memory (RAM) or **Main Memory**:
  - Fast
    - Random access, byte addressable
      - ~10x faster for sequential access
      - ~100,000x faster for random access!
  - *Volatile*
    - Data can be lost if e.g. crash occurs, power goes out, et
  - Expensive
    - For $100, get 16GB of RAM vs. 2TB of disk!

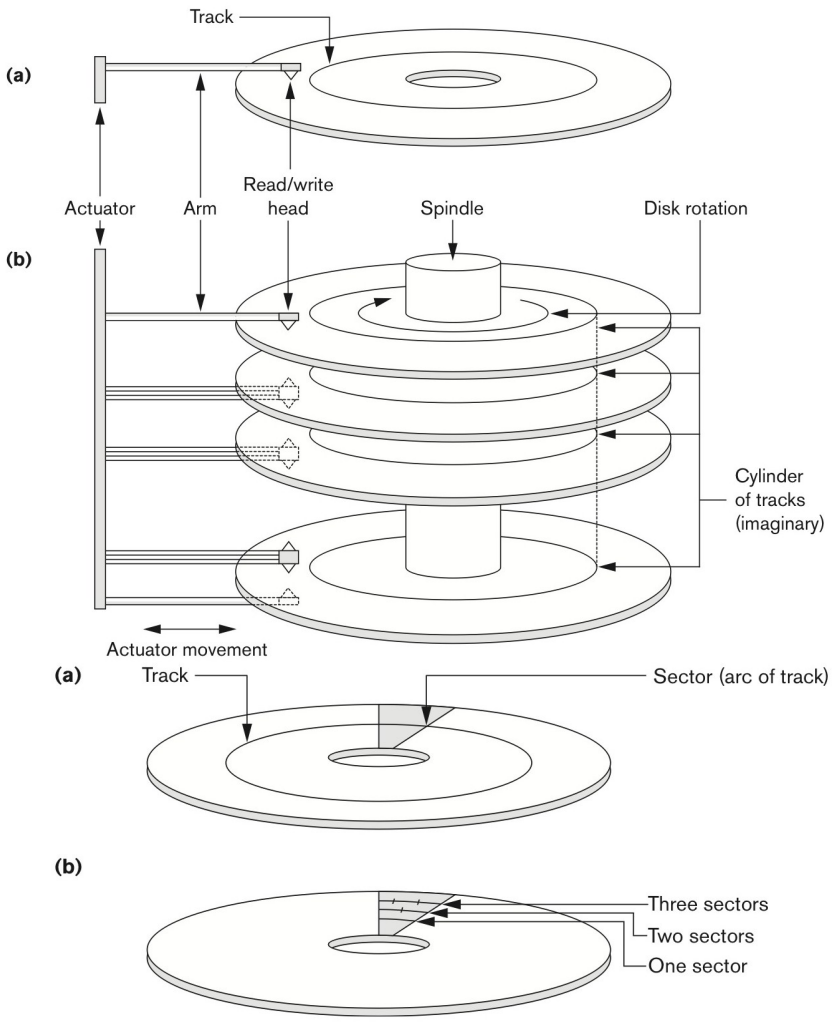
High-level: Disk vs. Main Memory

- Keep in mind the tradeoffs here as motivation for the mechanisms we introduce
  - Main memory: fast but limited capacity, volatile
    - Vs.
  - Disk: slow but large capacity, durable

How do we effectively utilize ***both*** ensuring certain critical guarantees?

Hardware Description of Disk Devices

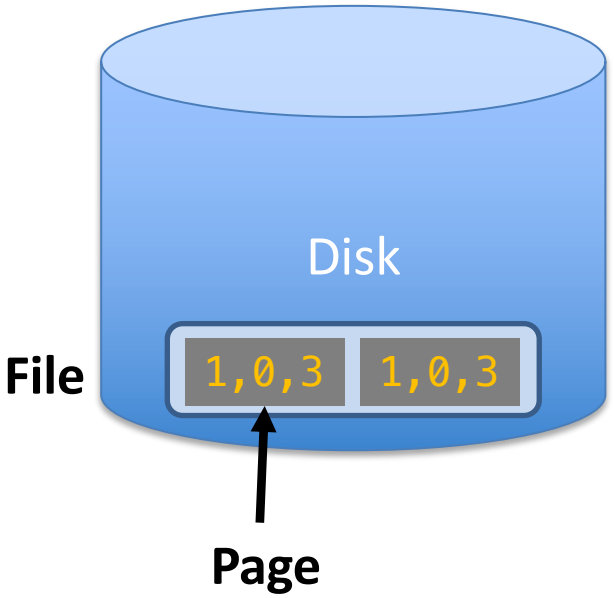
- Information is stored on a disk surface in concentric circles (Track)
- Tracks with same diameter on various surfaces is called cylinder
- Tracks are divided into sectors
- OS divides a track into equal sized disk blocks (pages)
  - One page = one or more sectors



A Simplified Filesystem Model

- For us, a **page** is a ***fixed-sized array*** of memory
  - One (or more) disk block (blocks)
  - Interface:
    - write to an entry (called a **slot**) or set to "None"

- And a **file** is a *variable-length list* of pages
  - Interface: create / open / close; next_page(); etc.



The Buffer

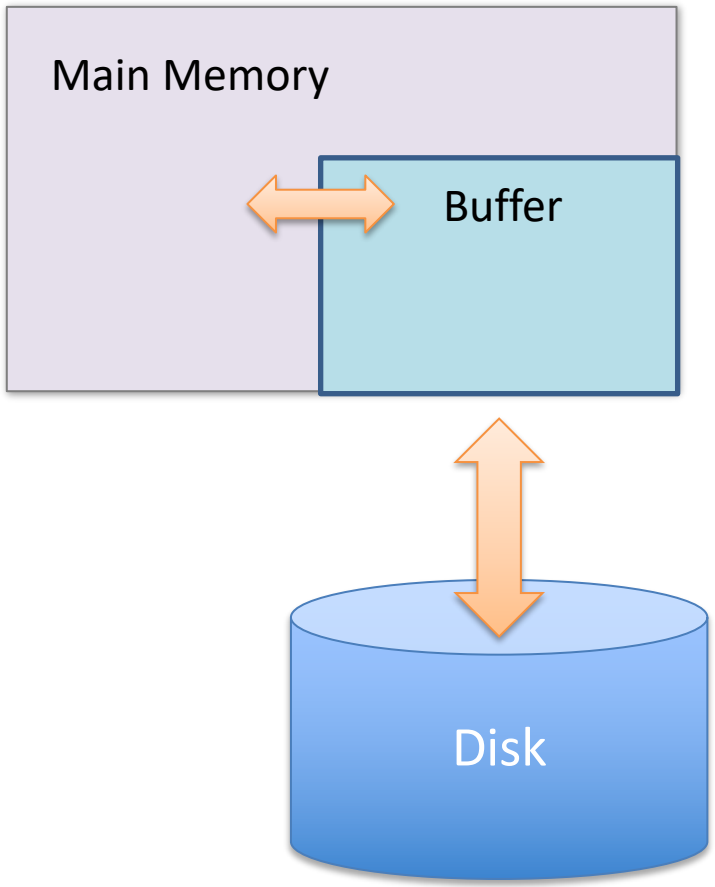
- Transfer of data between main memory and disk takes place in units of disk blocks.
- The hardware address of a block is a combination of a cylinder number, track number, and block number.

- A **buffer** is a region of physical memory used to store a single block.

- Sometimes, several contiguous blocks can be copied into a cluster

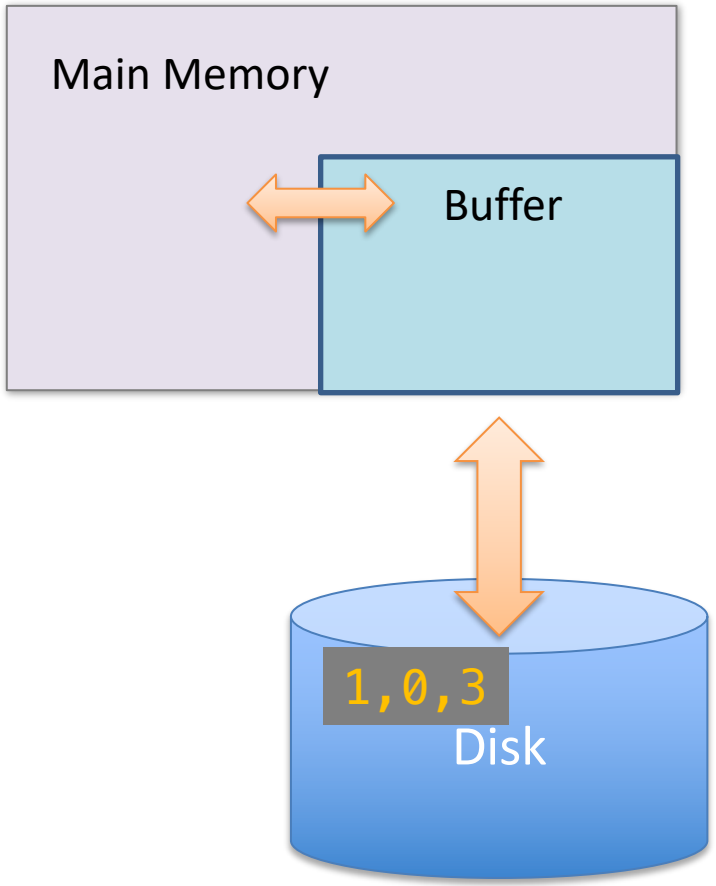
  – In this lecture: We will mostly not distinguish between a buffer and a cluster.

- *Key idea:* Reading / writing to disk is slow- need to cache data!



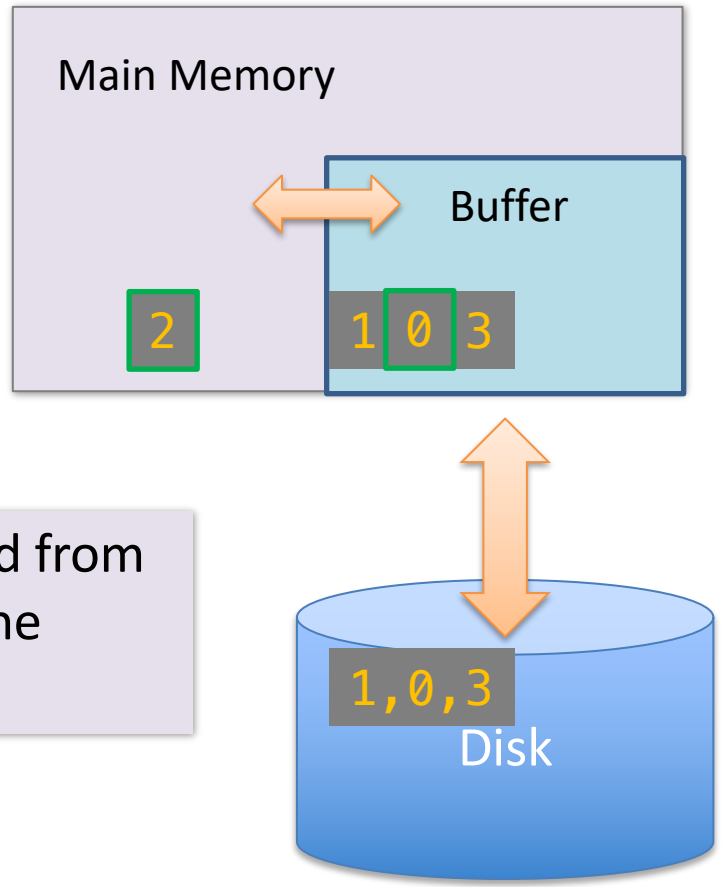
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*



The (Simplified) Buffer

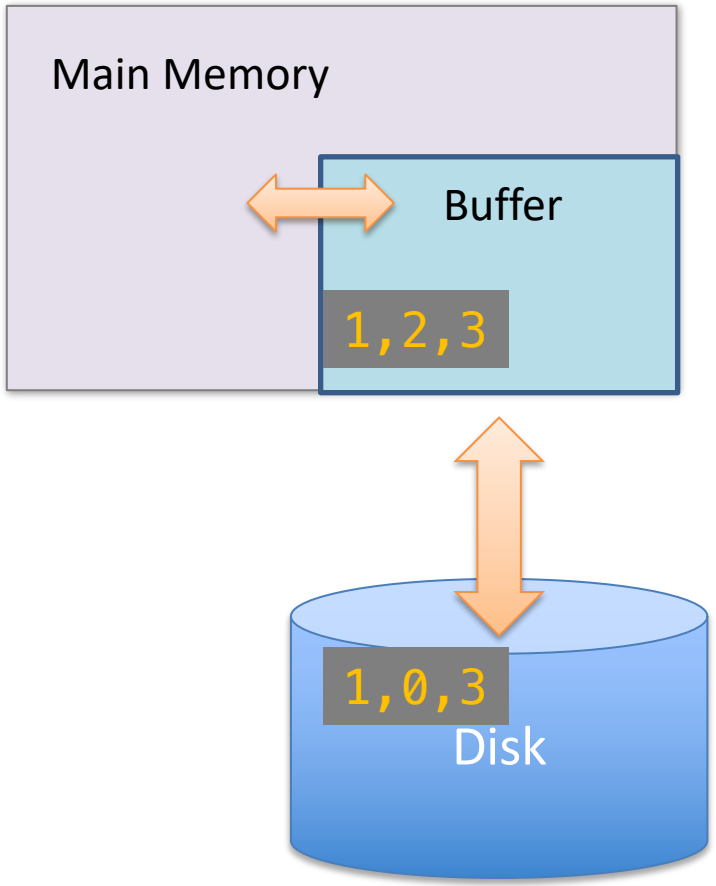
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*



Processes can then read from / write to the page in the buffer

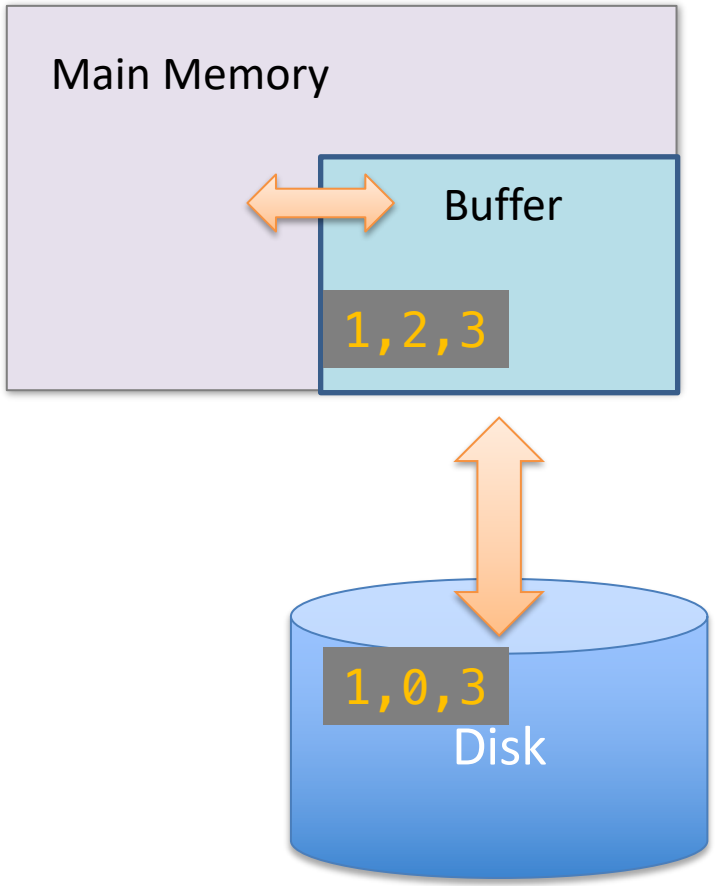
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*
  - **Flush(page):** Evict page from buffer & write to disk



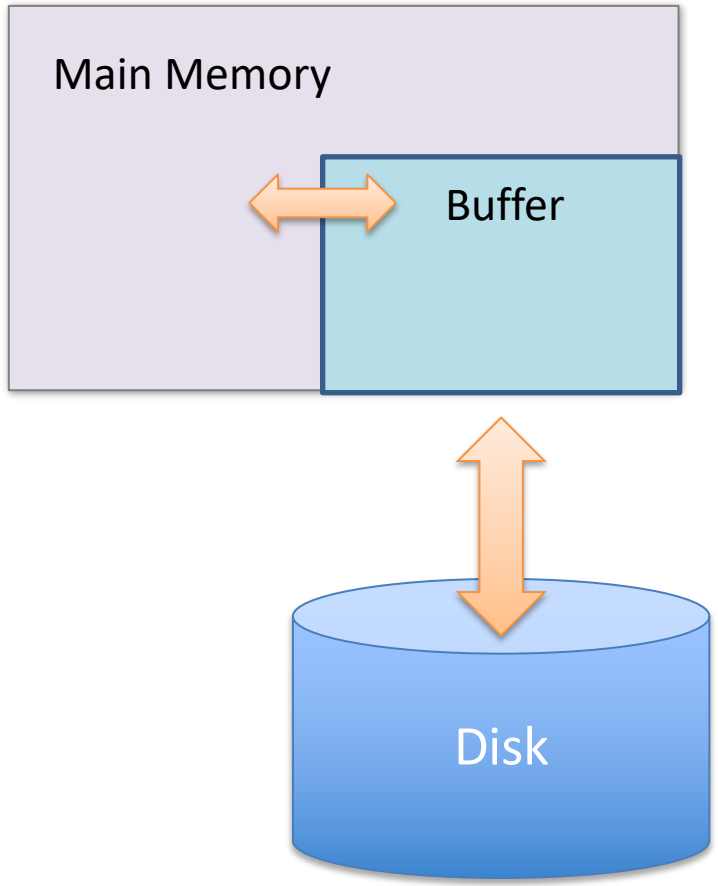
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*
  - **Flush(page):** Evict page from buffer & write to disk
  - **Release(page):** Evict page from buffer *without* writing to disk



Managing Disk: The DBMS Buffer

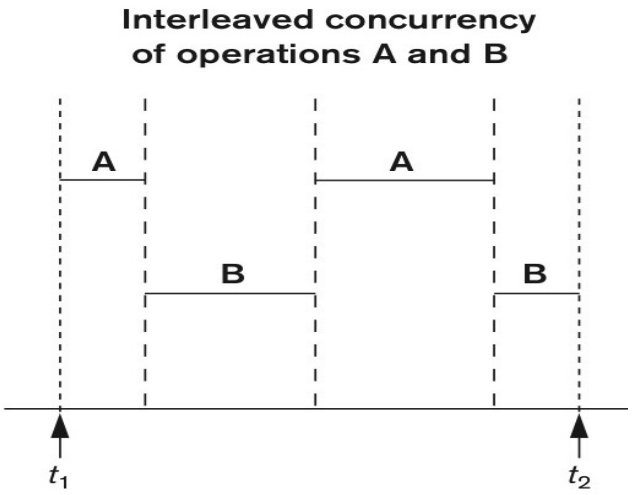
- Database maintains its own buffer
  - Why? The OS already does this…
  - DB knows more about access patterns.
  - Recovery and logging require ability to **flush** to disk.

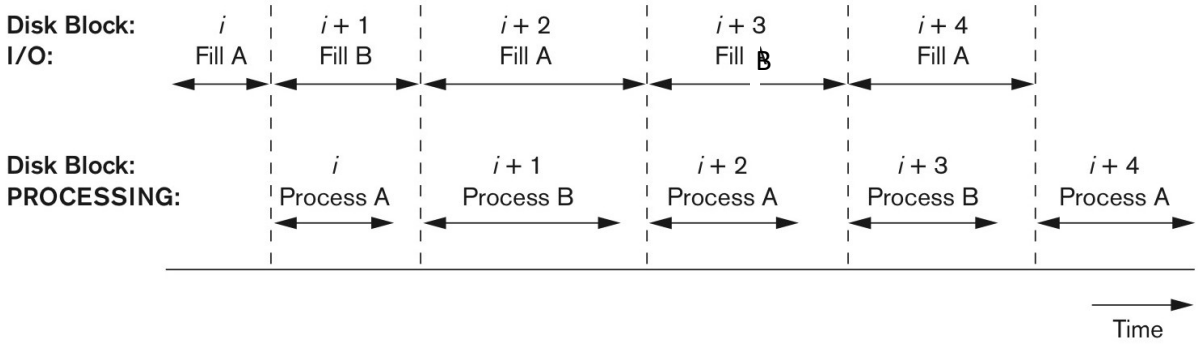


The Buffer Manager

- A **buffer manager** handles supporting operations for the buffer:
  - Primarily, handles & executes the "replacement policy"
    - i.e. finds a page in buffer to flush/release if buffer is full and a new page needs to be read in
  - DBMSs typically implement their own buffer management routines

Use of Two Buffer





Buffer Replacement Strategies

- Least recently used (LRU)
- Clock policy
- First-in-first-out (FIFO)
- Refer 16.3.2 for details

Records and Files

- Data is usually stored in the form of records

- Each record consists of a collection of related data values or items.
  - Record usually describe entities

File Types

- Unordered Records (Heap Files)
- Ordered Records (Sorted Files)

Heap Files

- Insertion (of a record):
  - Very efficient.
  - Last disk block is copied into a buffer
  - New record is added
  - Block is rewritten back to disk
- Searching:
  - Linear search
- Deletion:
  - Rewrite empty block after deleting record. (or)
  - Use deletion marker

Sorted Files

- Physically sort the records of a file
  - Based on the values of one of the fields (ordering fields)
  - Ordered and sequential file

- Searching:
  - Can perform Binary Search.

- Insertion and Deletion:
  - Expensive

Average Access Times for a File of *b* Blocks under Basic File Organizations

Table 16.3 Average Access Times for a File of *b* Blocks under Basic File Organizations

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

2. EXTERNAL MERGE & SORT

Challenge: Merging Big Files with Small Memory

How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?

External Merge Algorithm

- **Input**: 2 sorted lists of length M and N
- **Output:** 1 sorted list of length M + N
- **Required:** At least 3 Buffer Pages
- **IOs**: 2(M+N)

STOP!
Think about the solution before you proceed!
The idea is same as merge step in Merge sort

Recap: Merge Sort

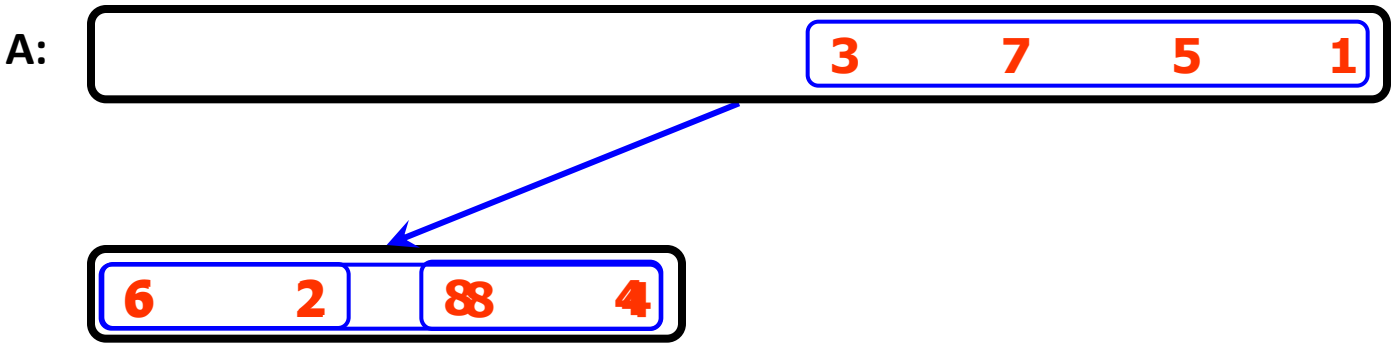
Merge-Sort(A, 0, 7)

Divide

A:

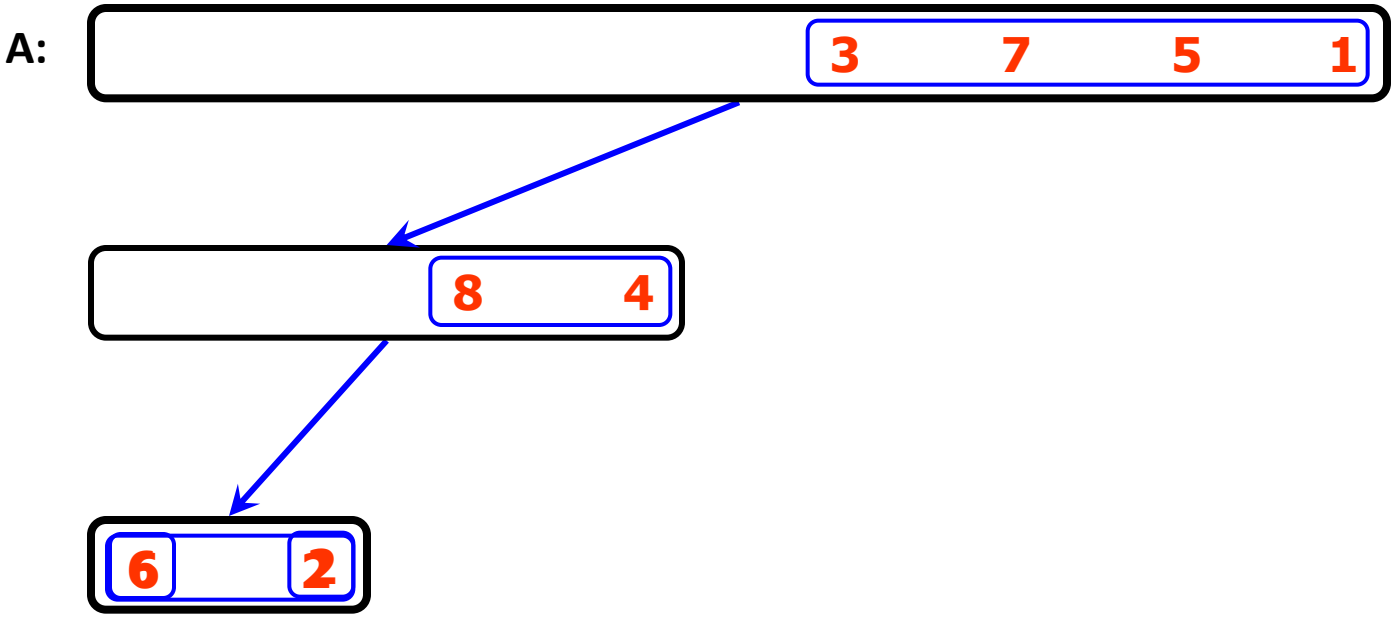
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 3) , divide



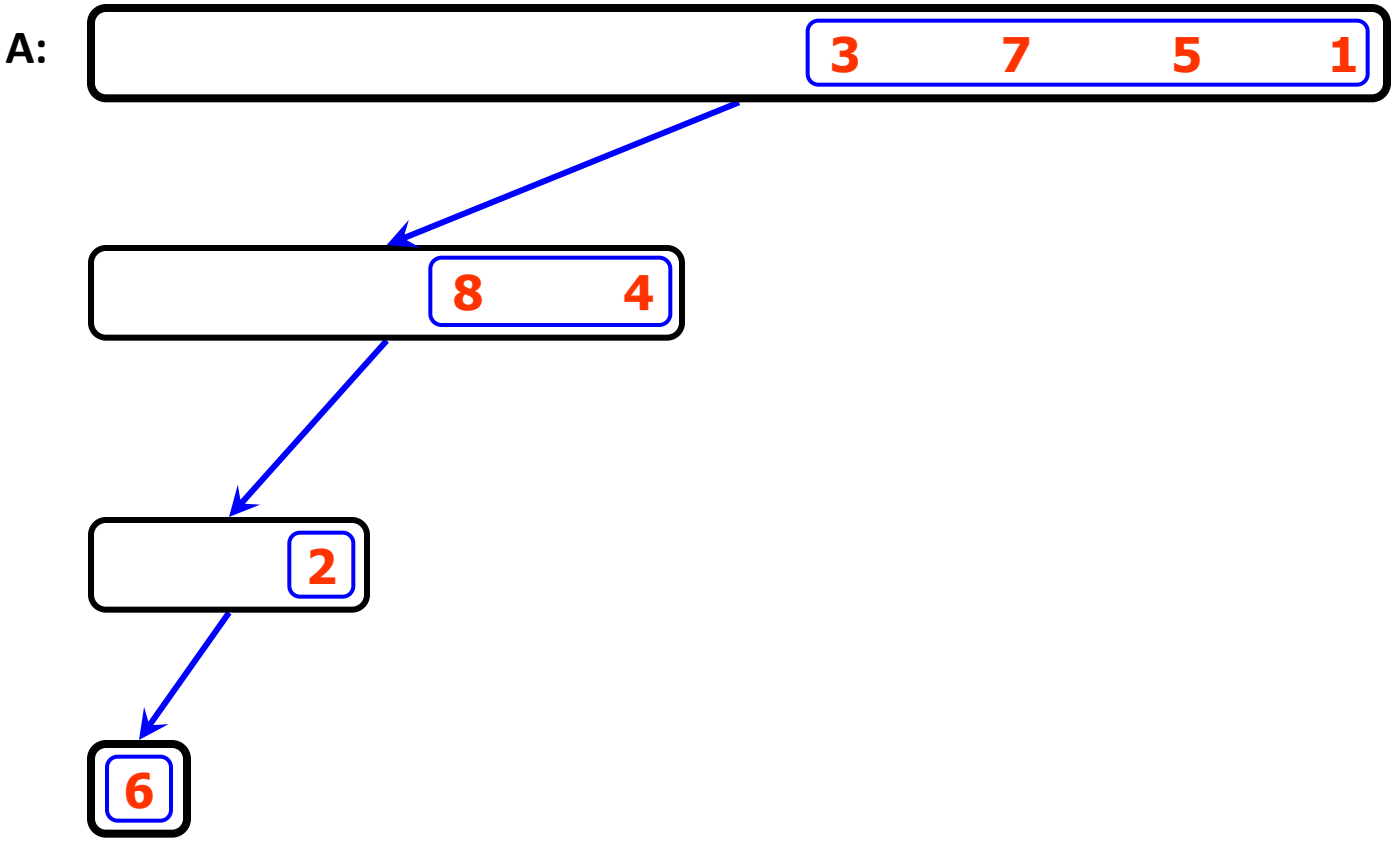
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1) , divide



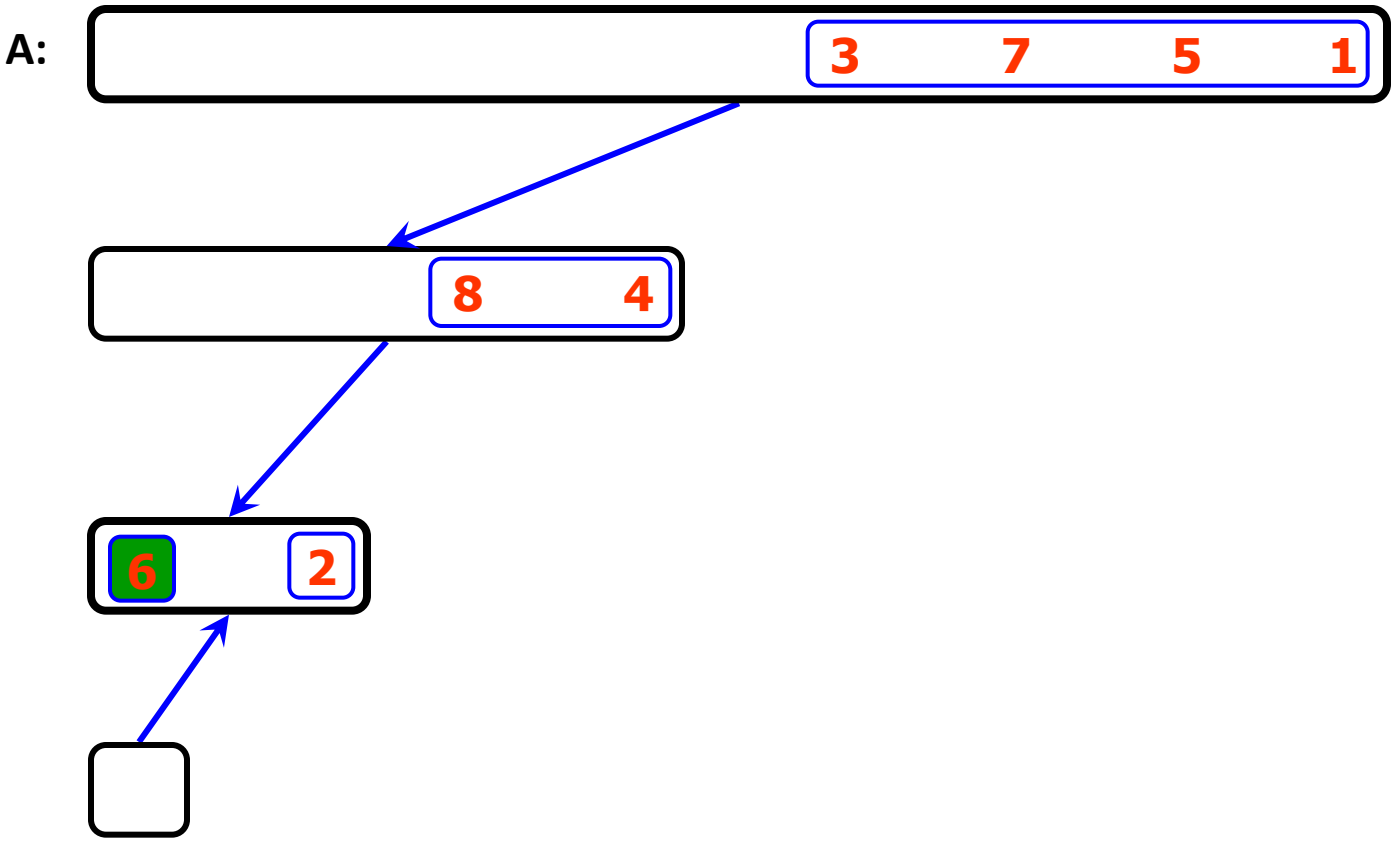
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0) , base case



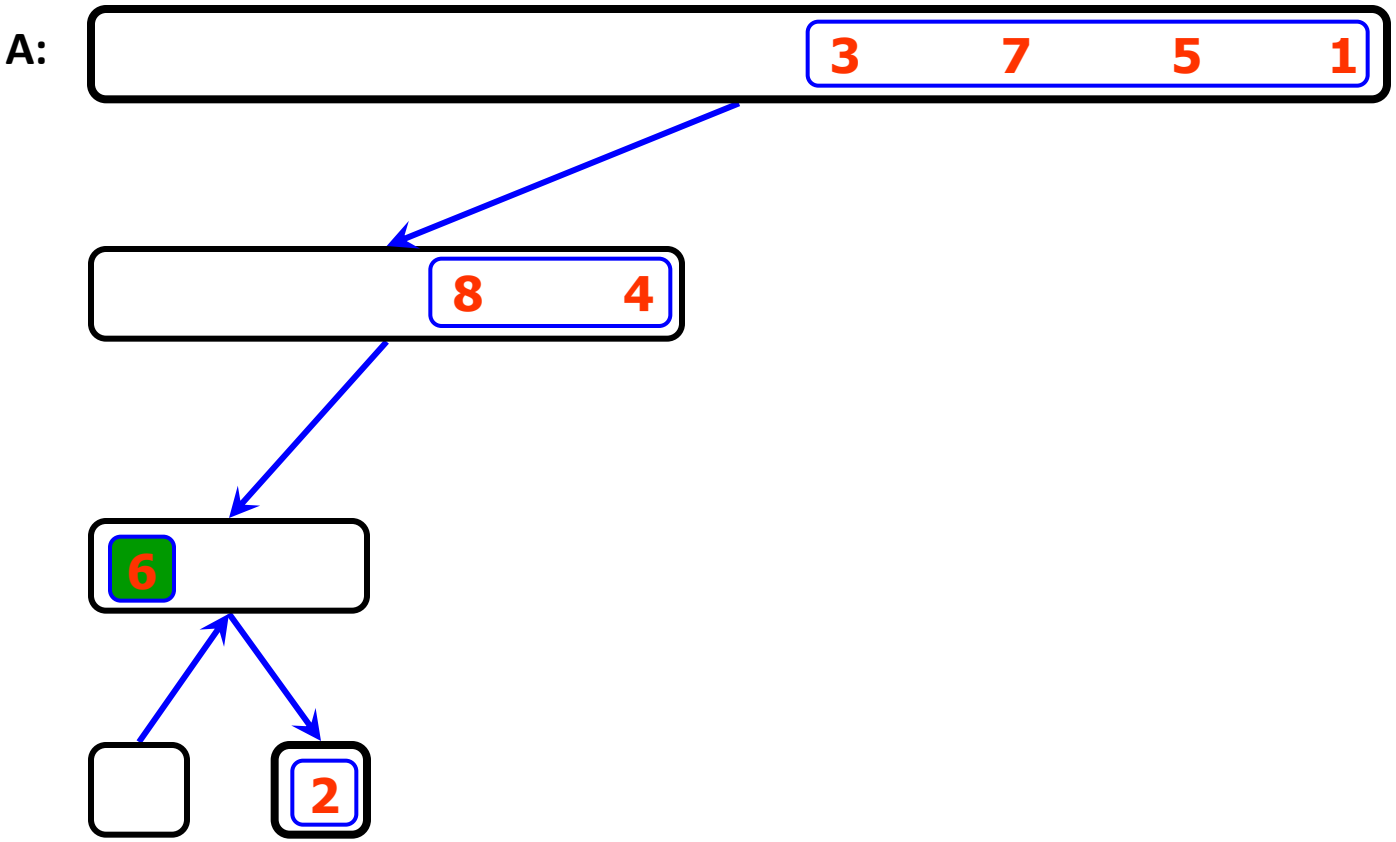
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0), return



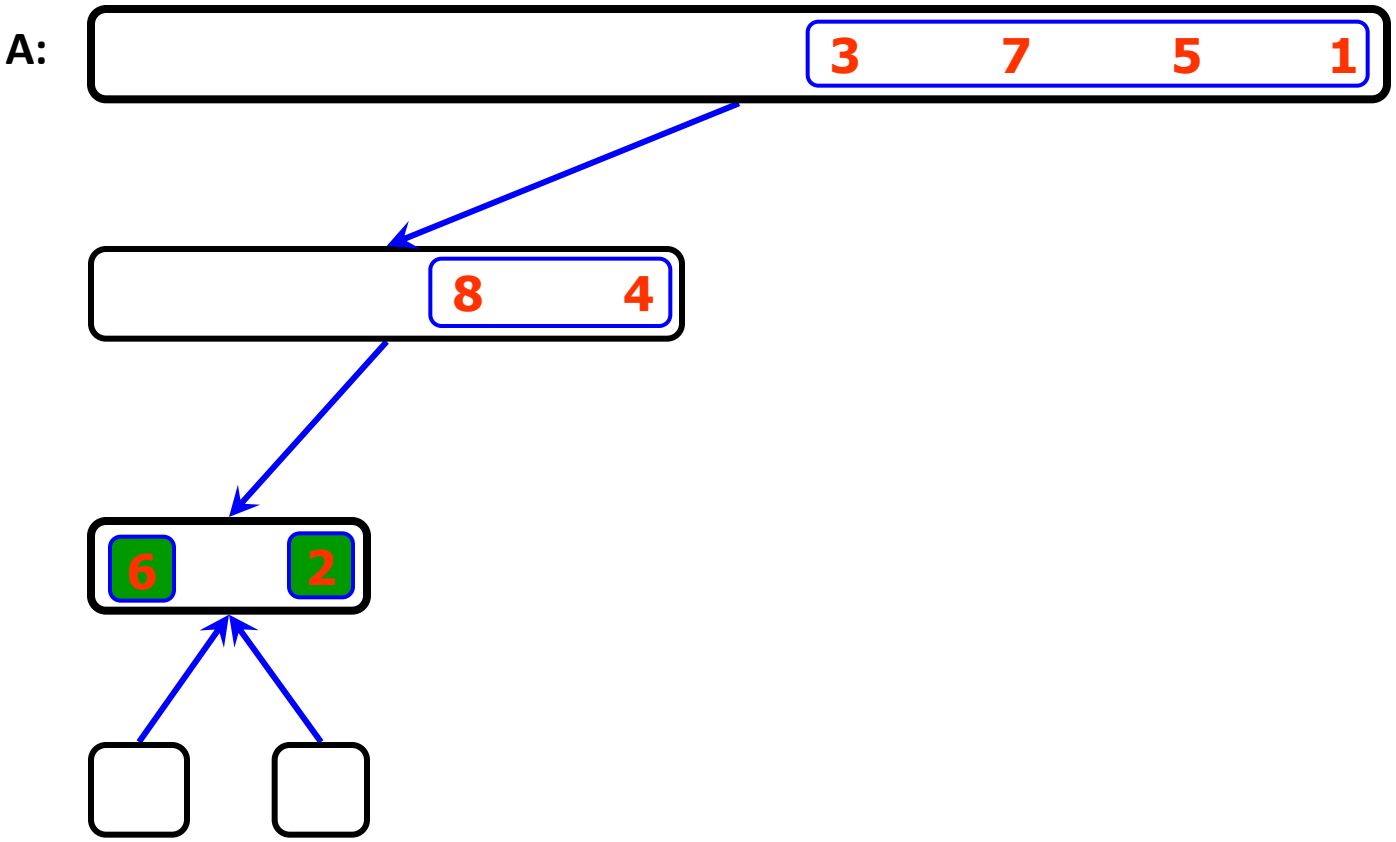
Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1) , base case



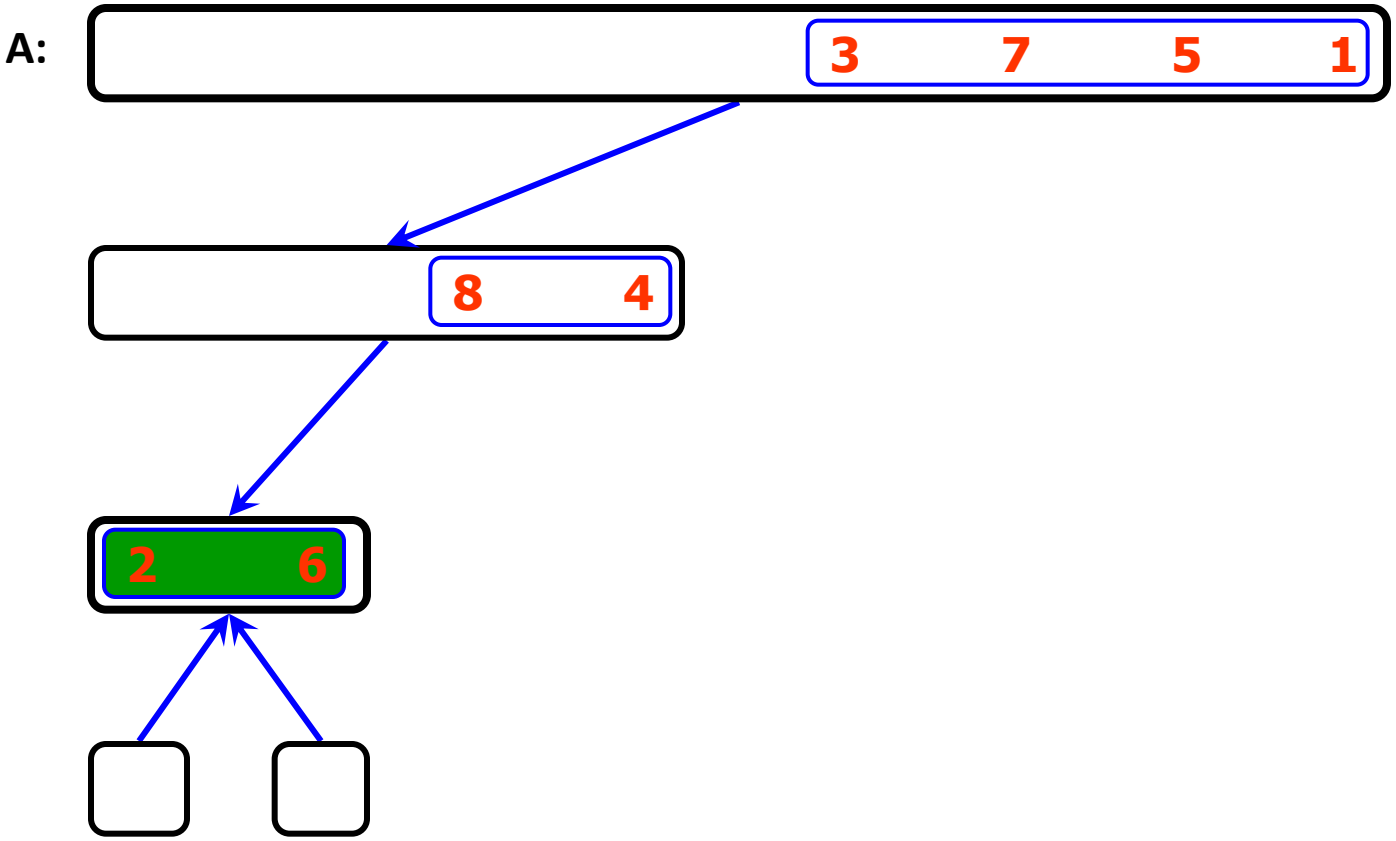
Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1), return



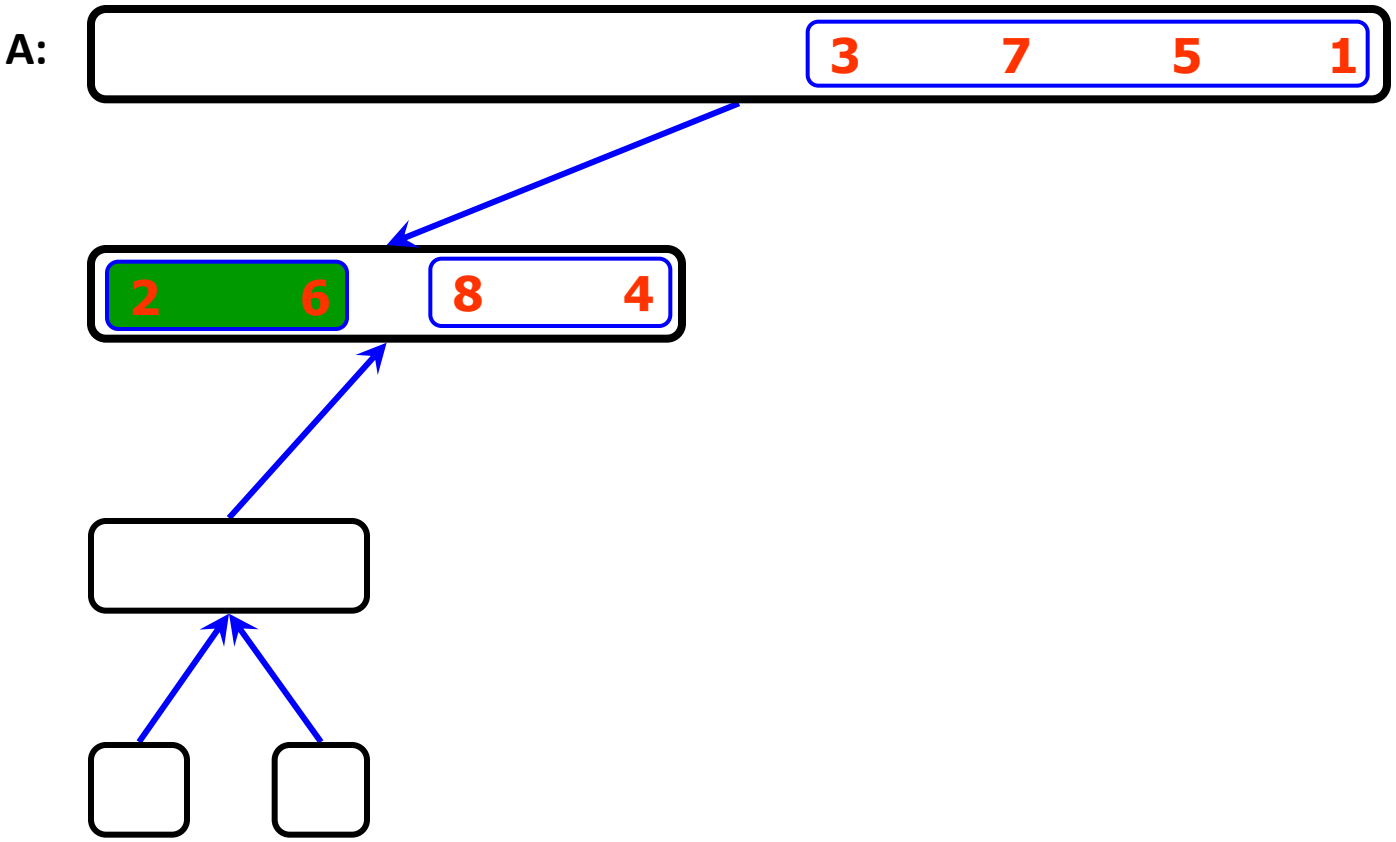
Merge-Sort(A, 0, 7)

Merge(A, 0, 0, 1)



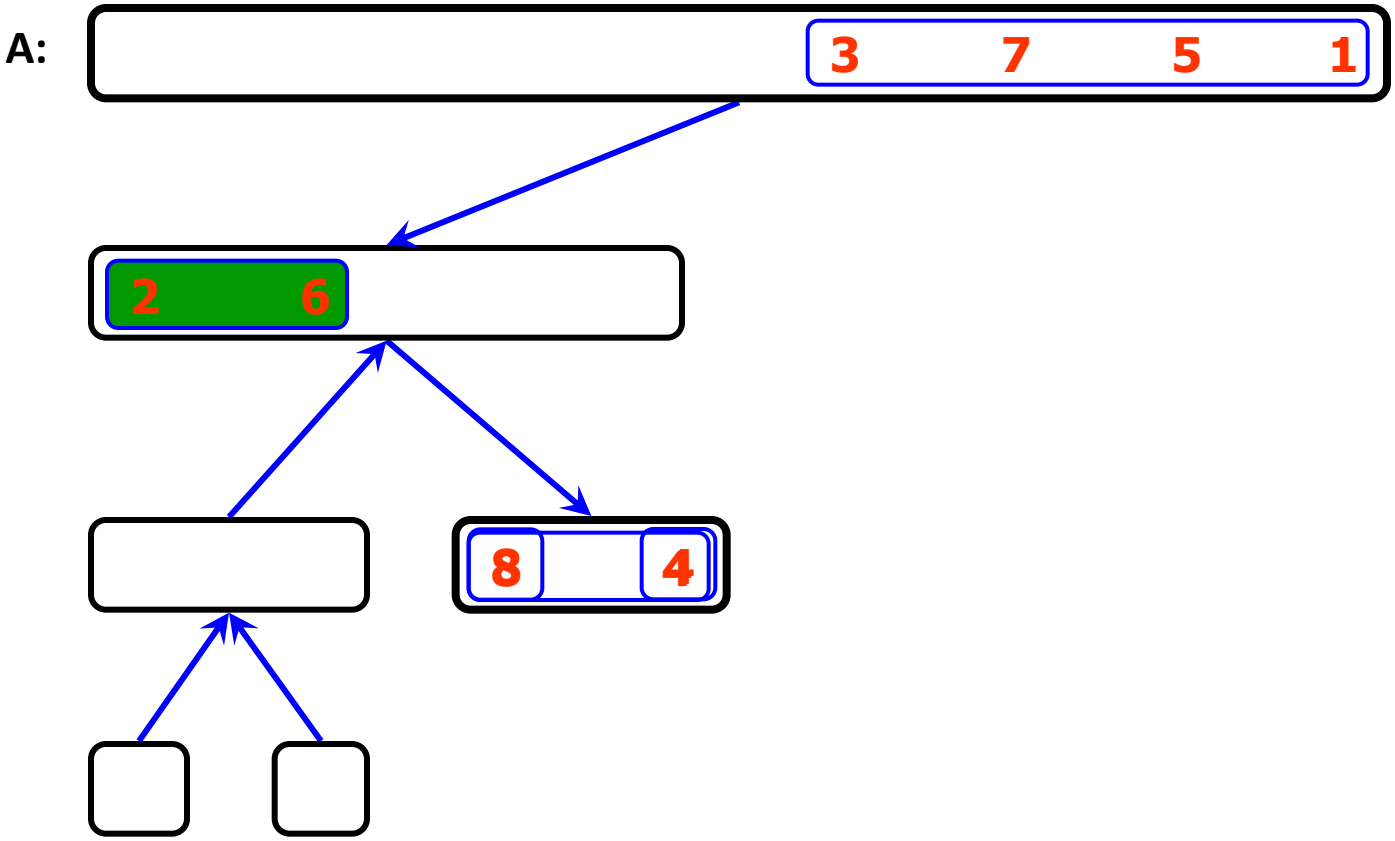
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1), return



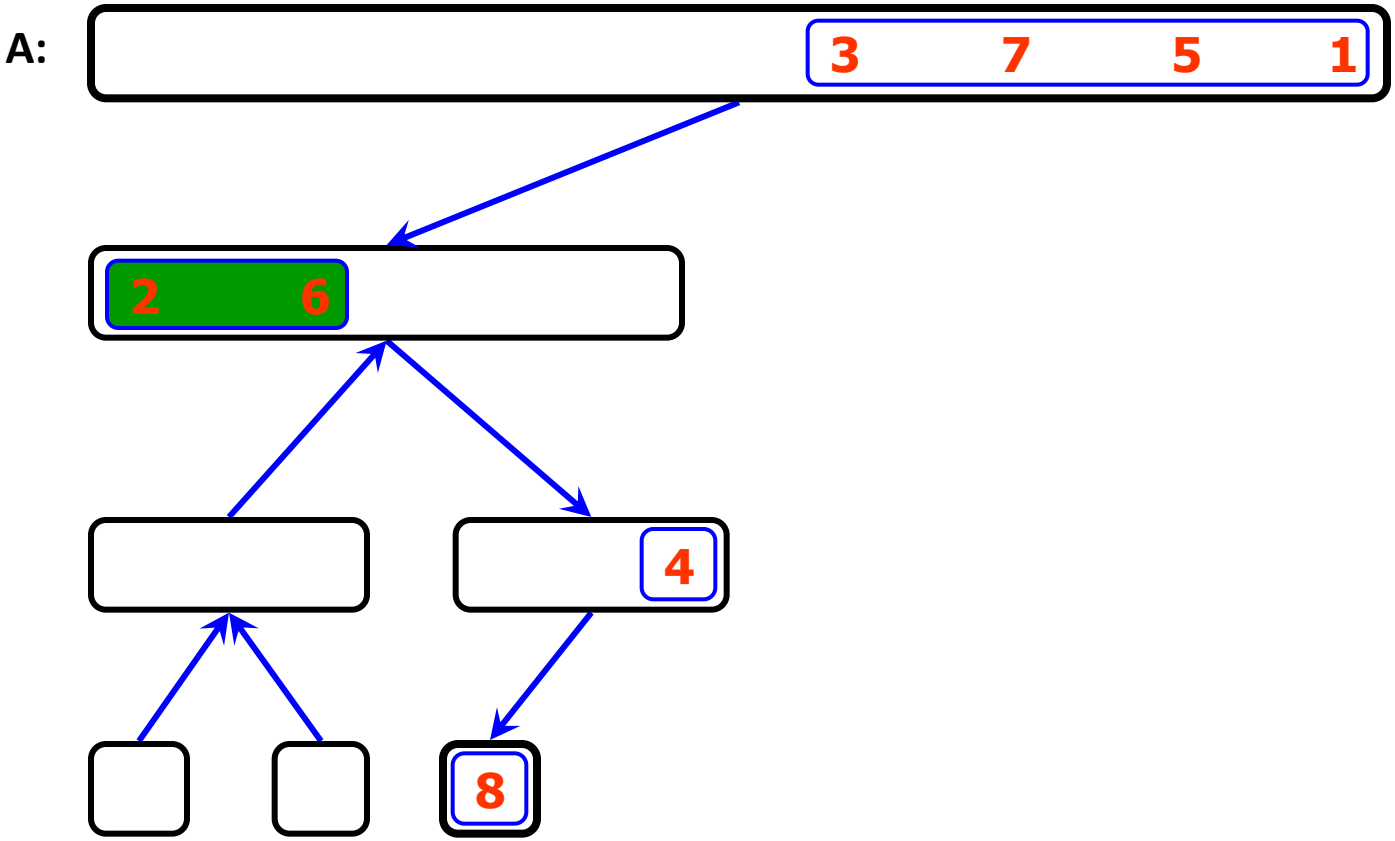
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3) , divide



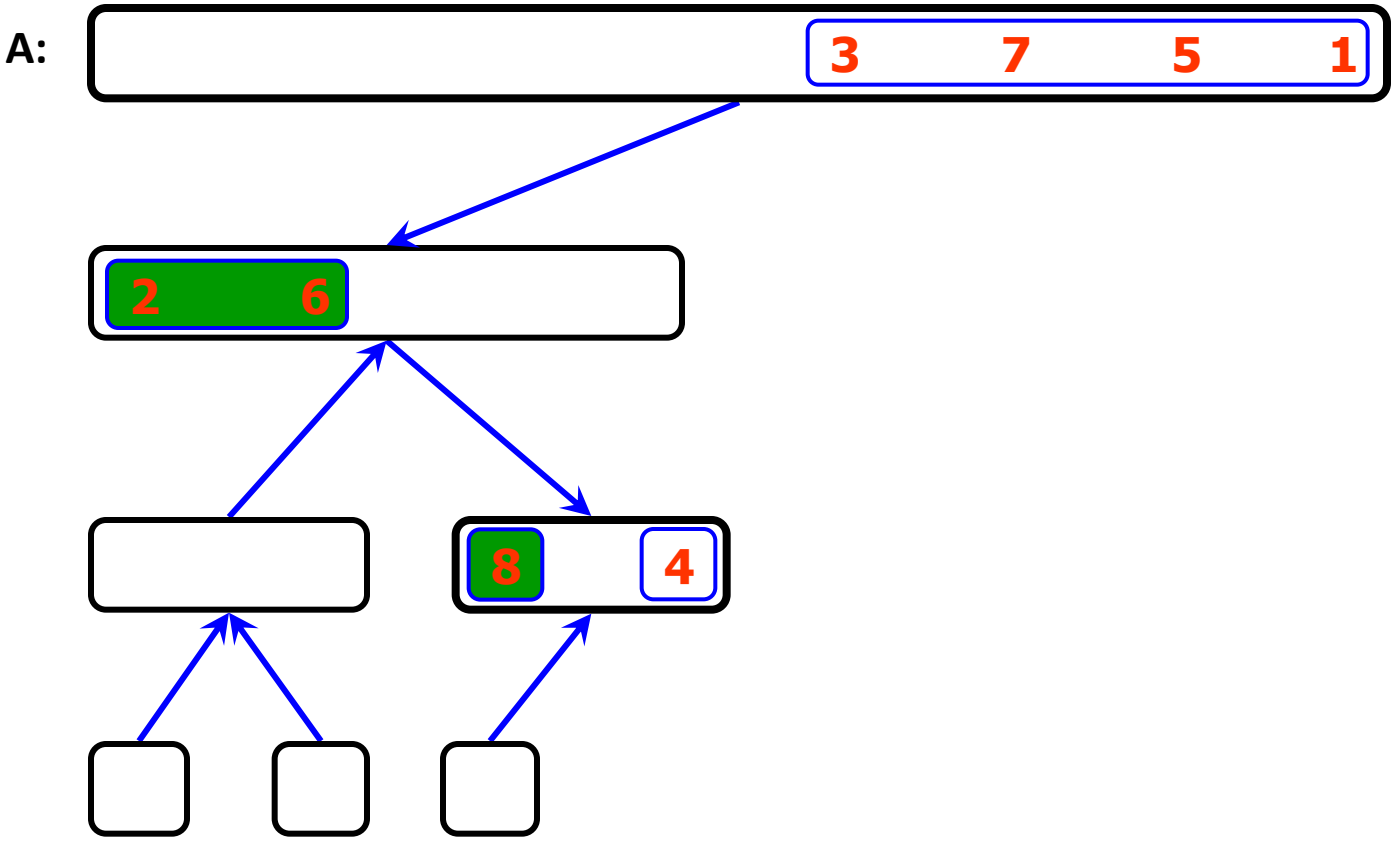
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), base case



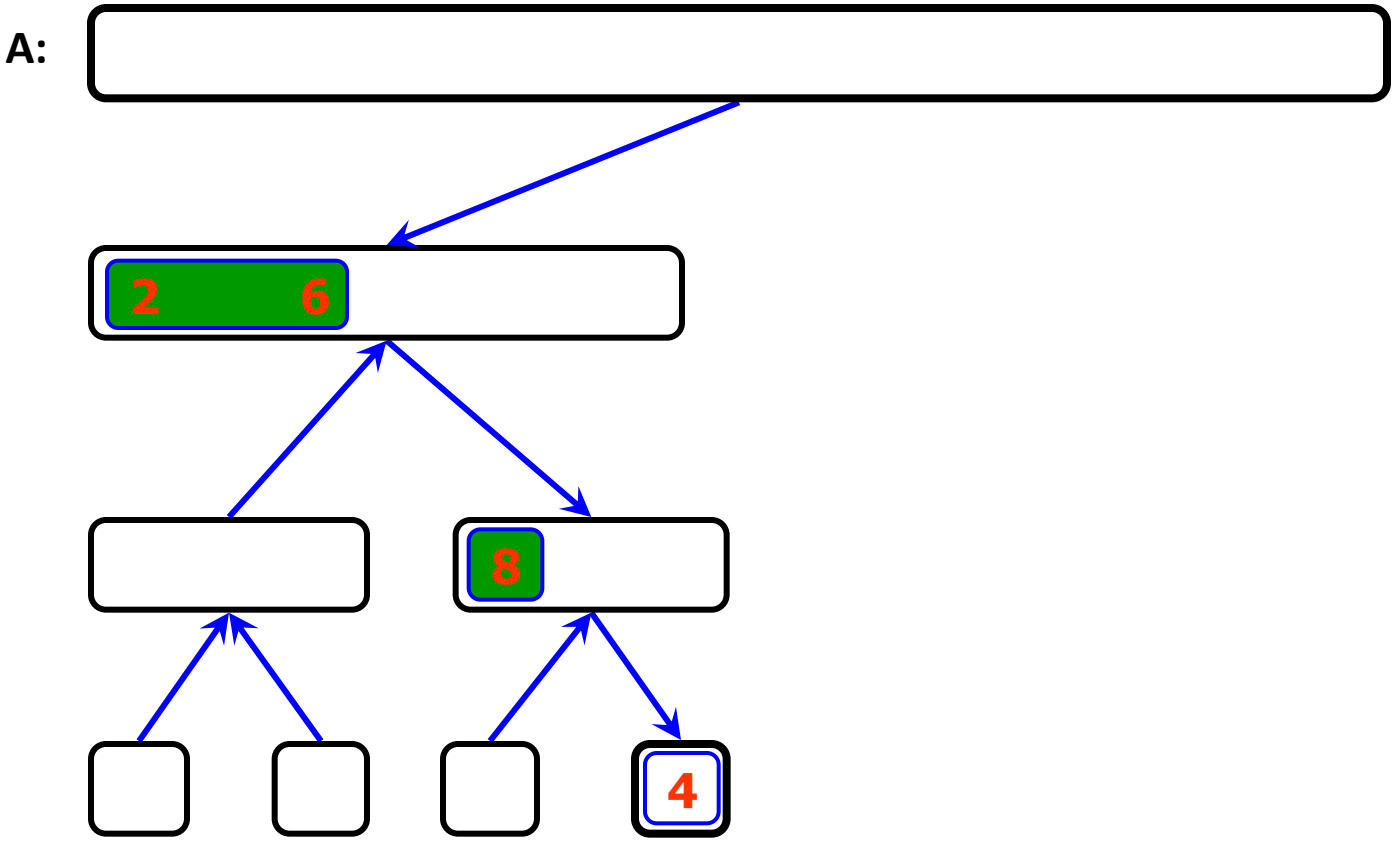
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), return



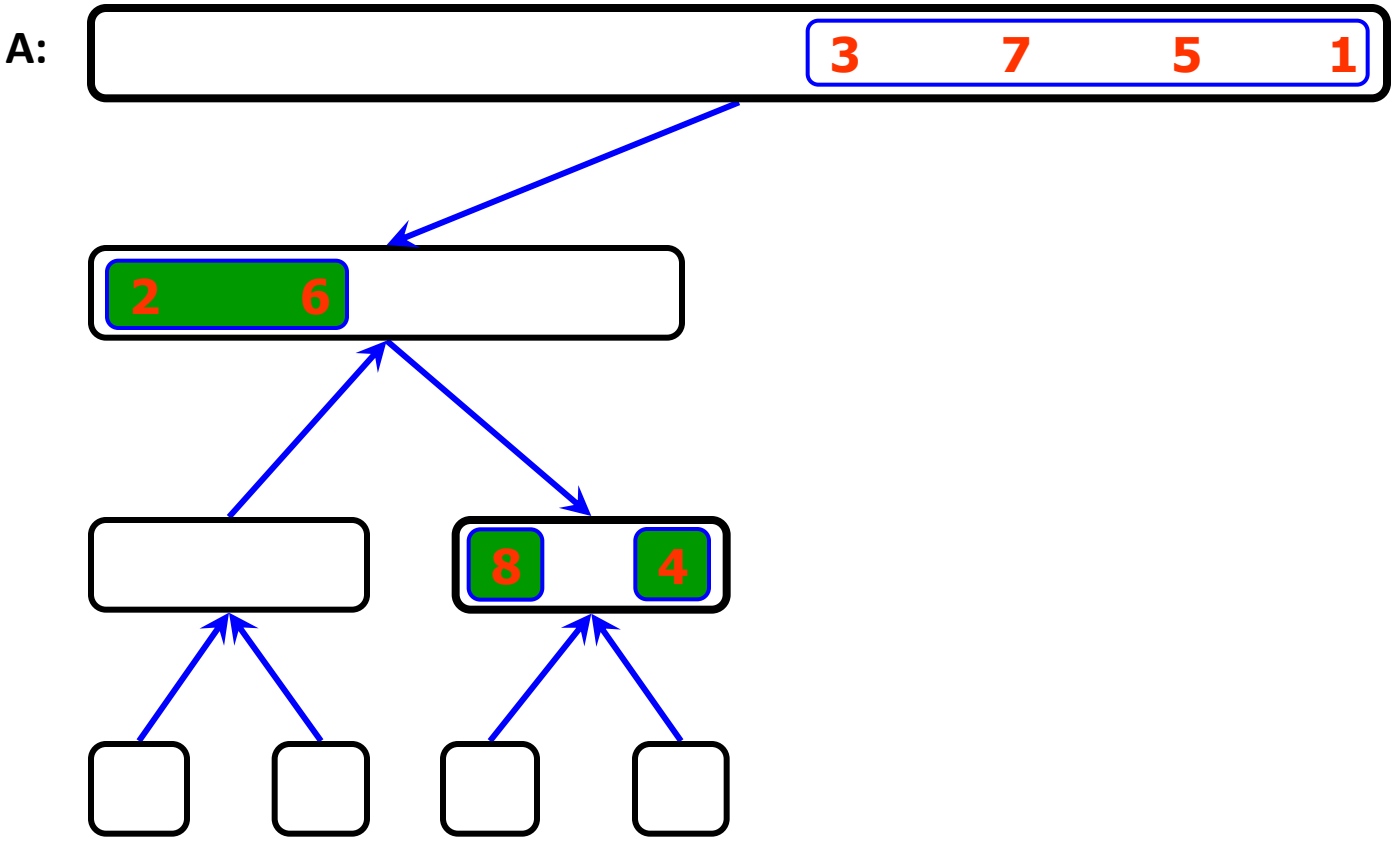
Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), base case



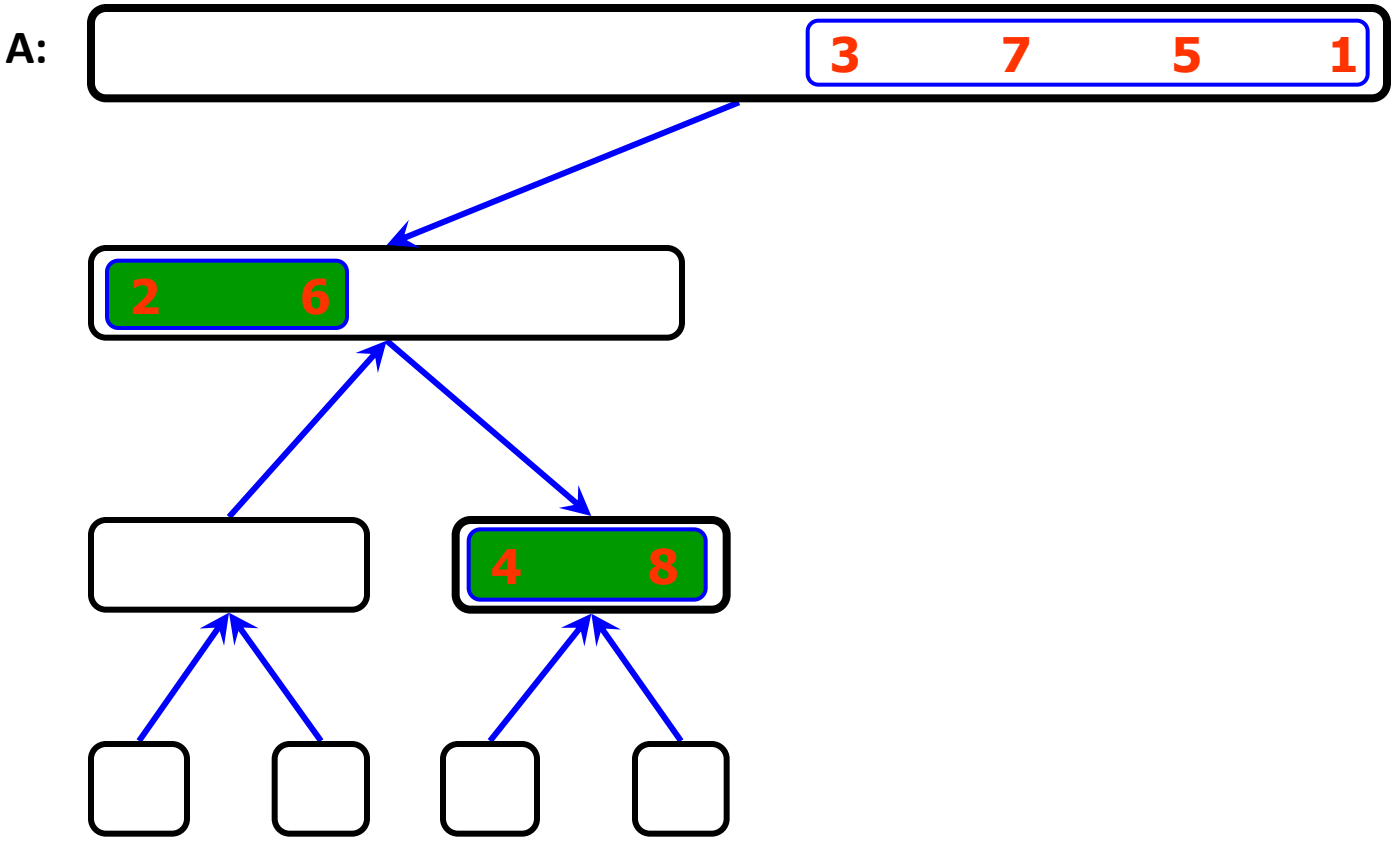
Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), return



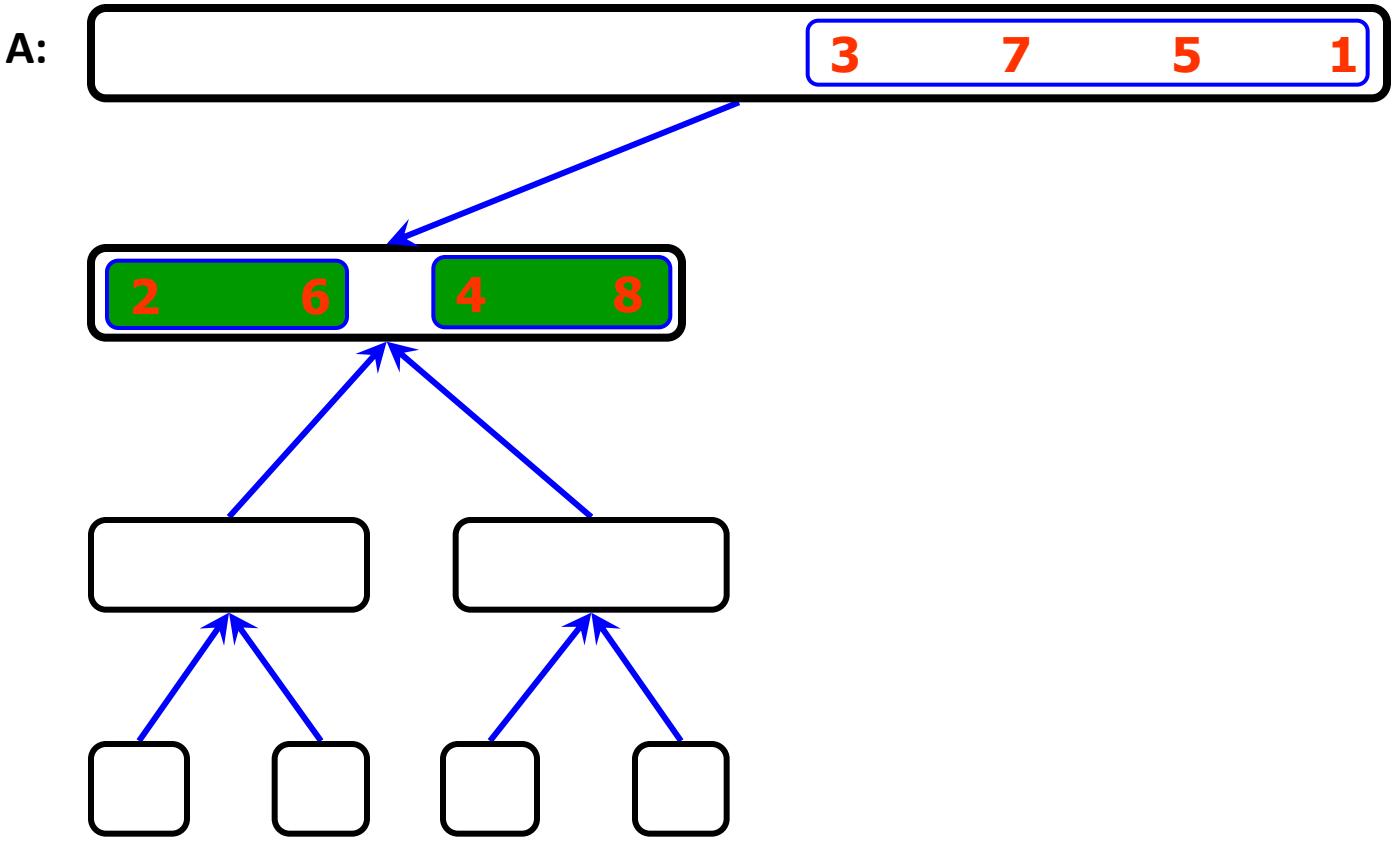
Merge-Sort(A, 0, 7)

Merge(A, 2, 2, 3)



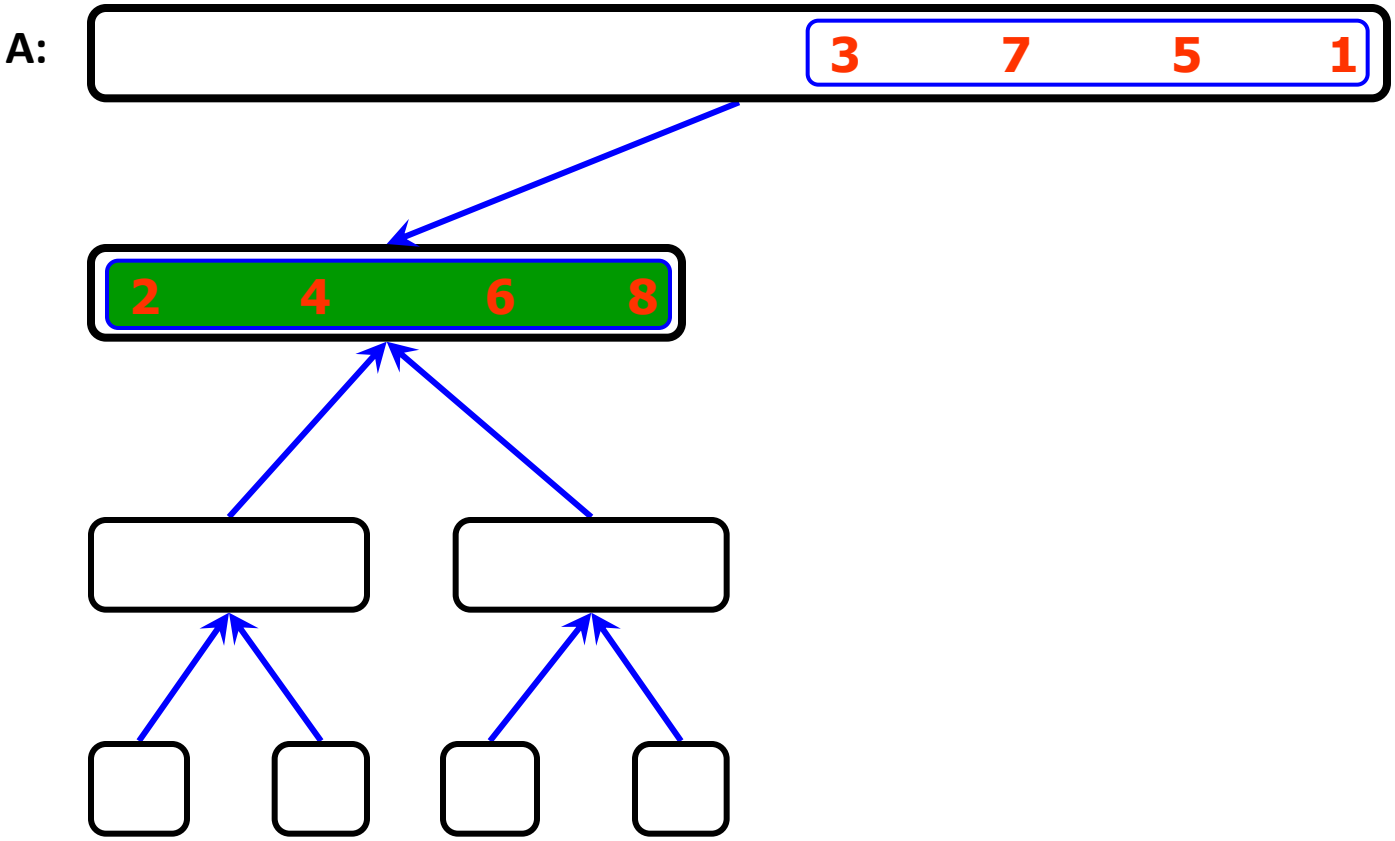
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3), return



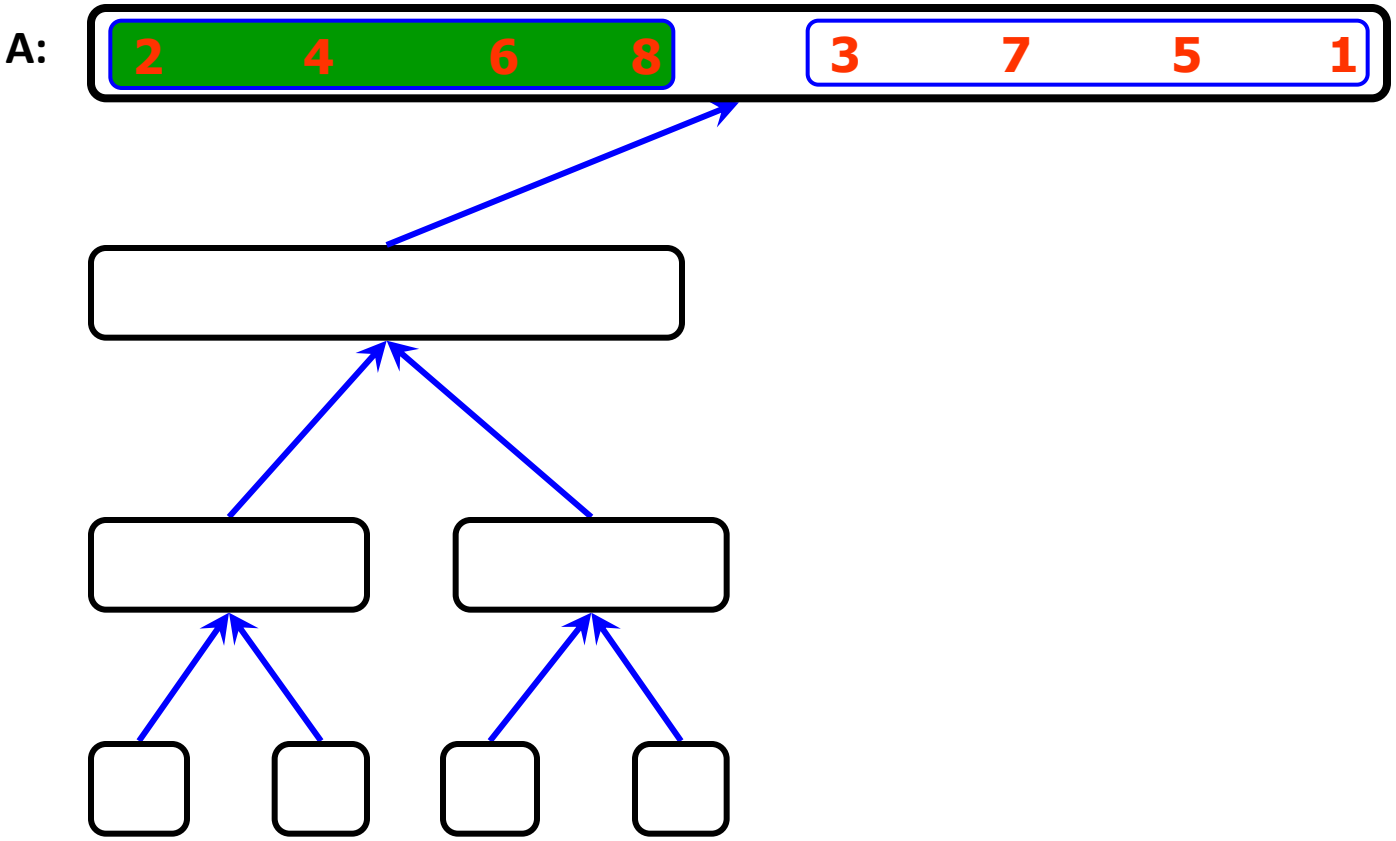
Merge-Sort(A, 0, 7)

Merge(A, 0, 1, 3)



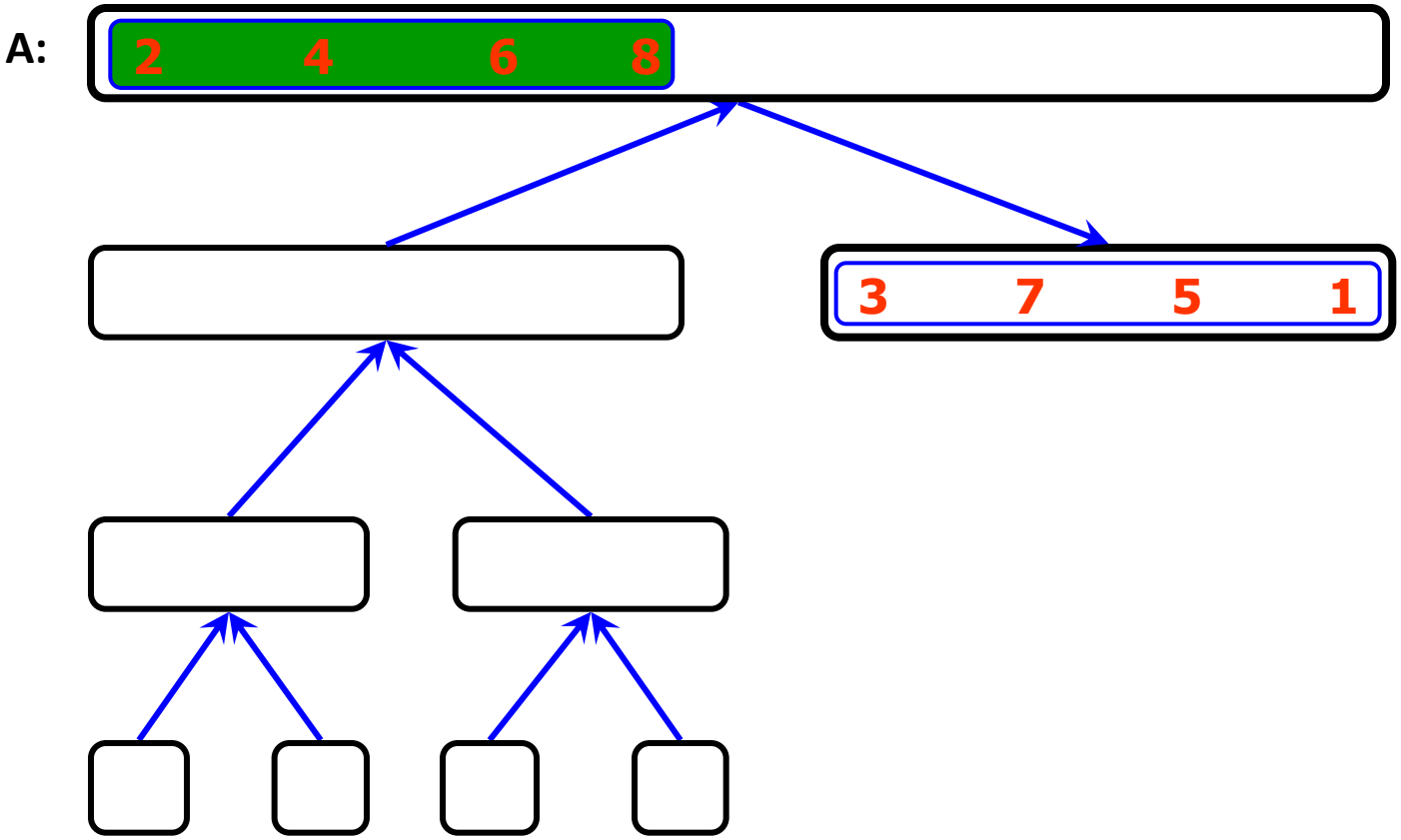
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 3), return



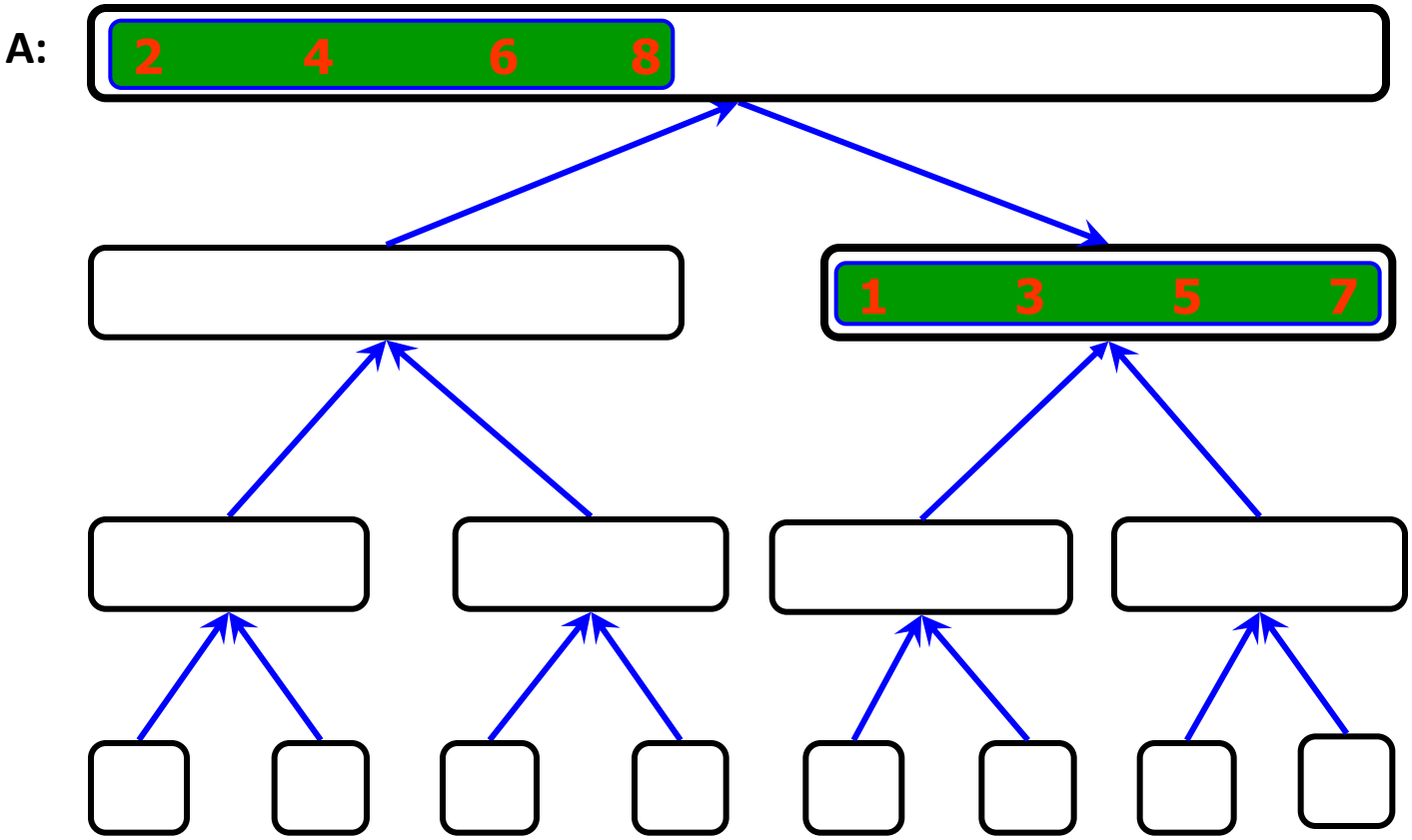
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7)



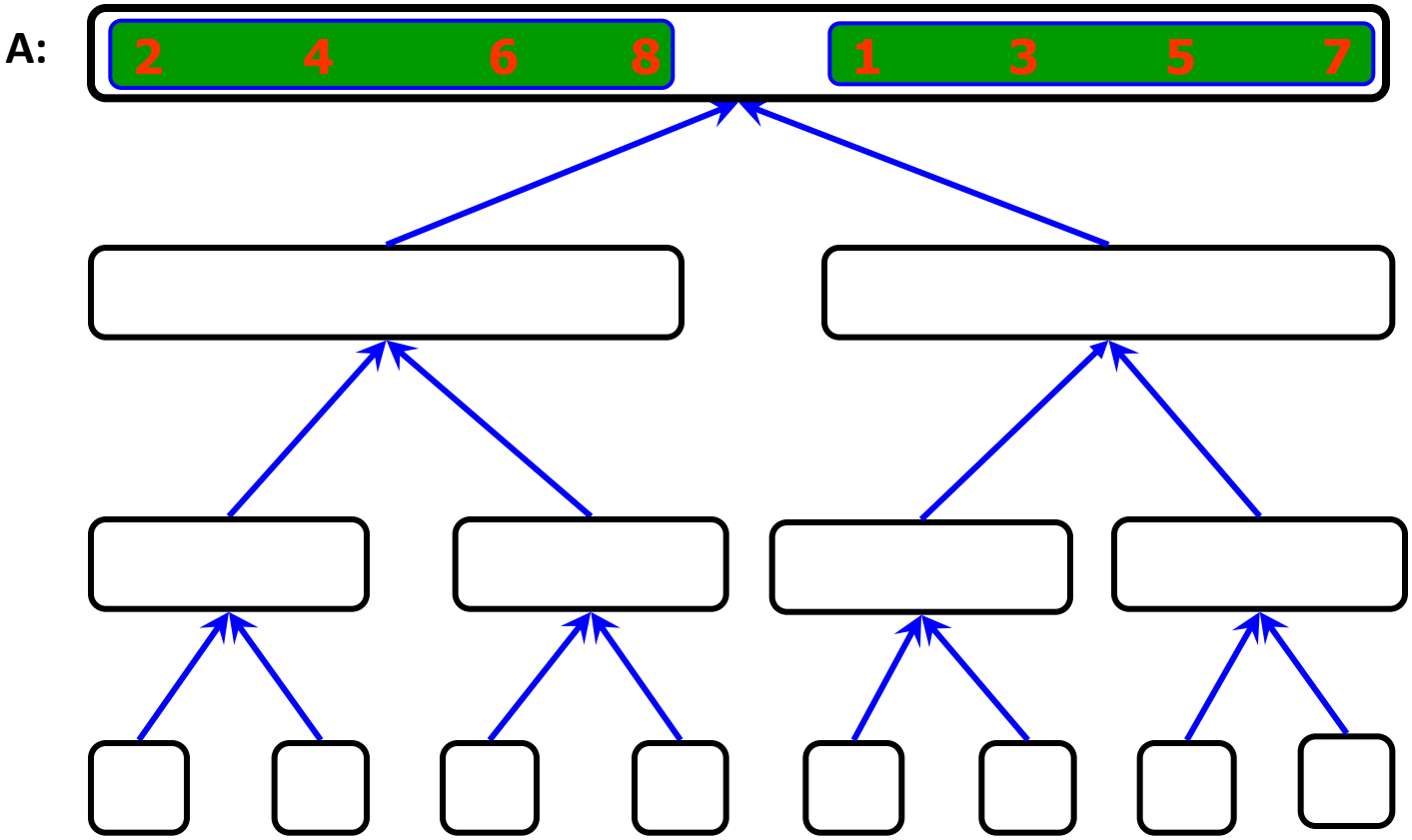
Merge-Sort(A, 0, 7)

Merge (A, 4, 5, 7)



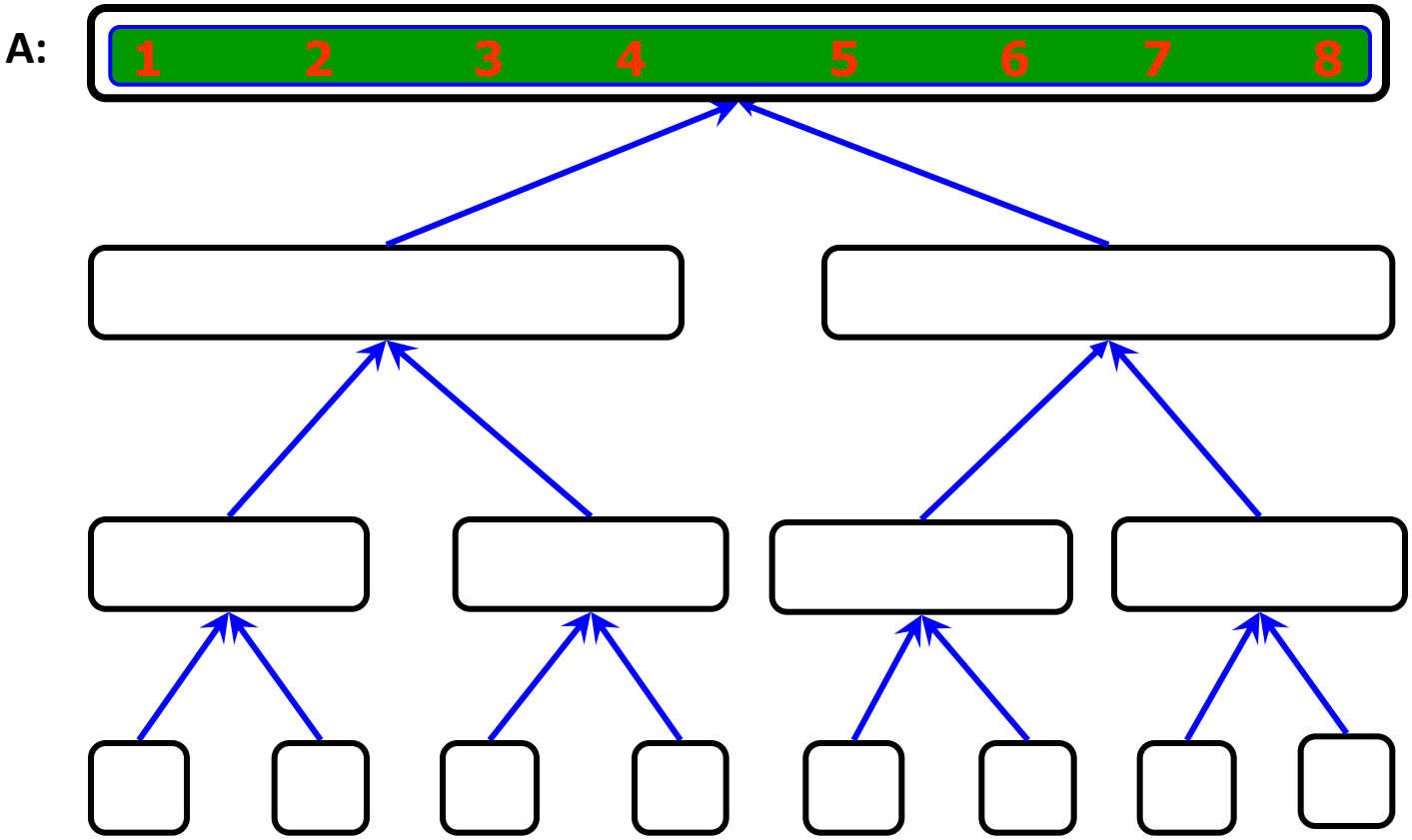
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7), return

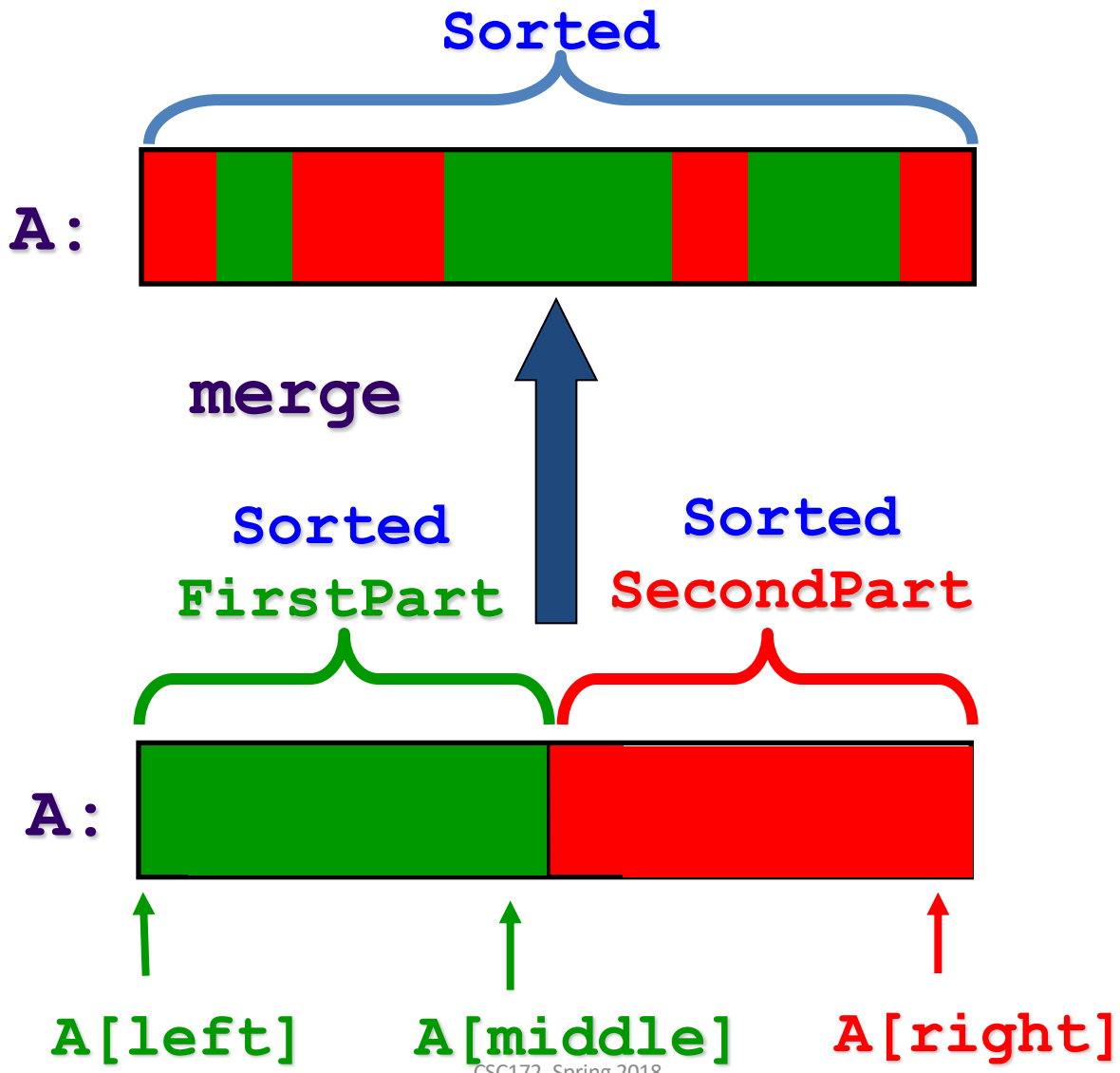


Merge-Sort(A, 0, 7)

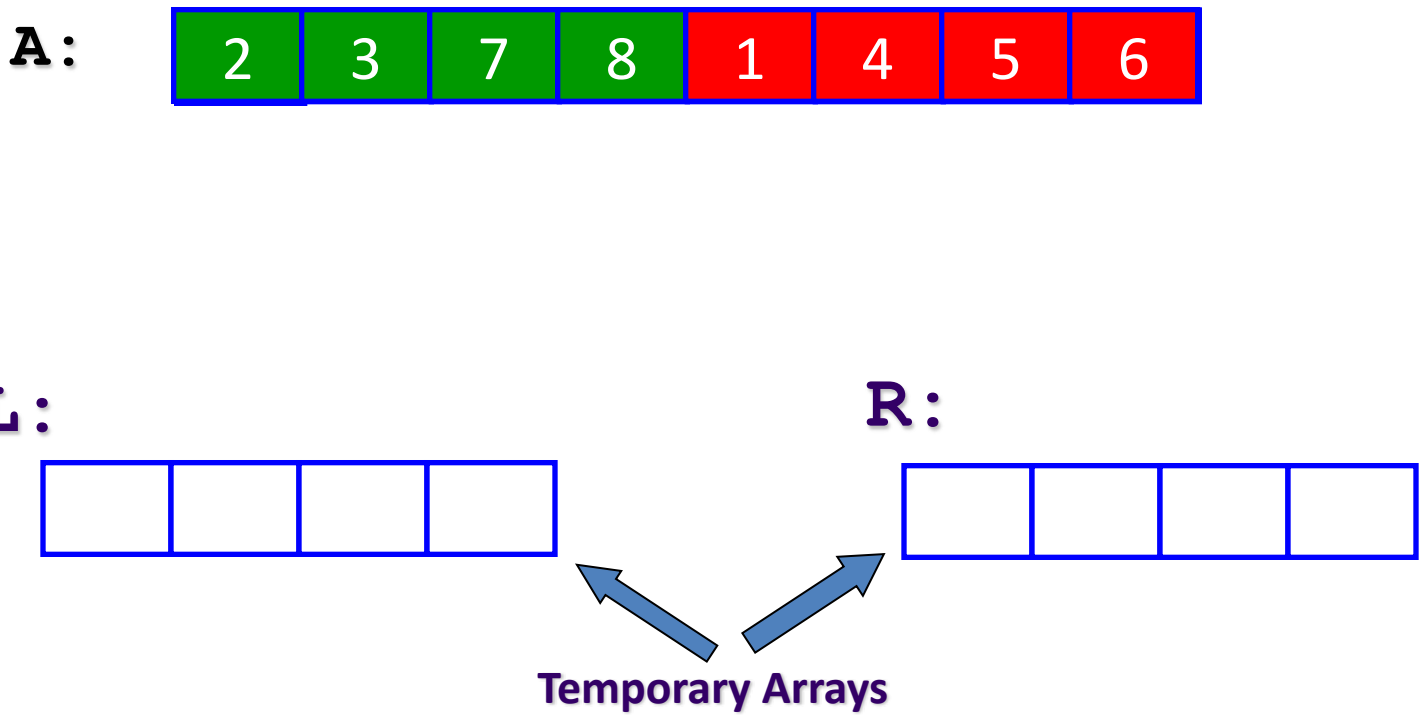
Merge-Sort(A, 0, 7), done!



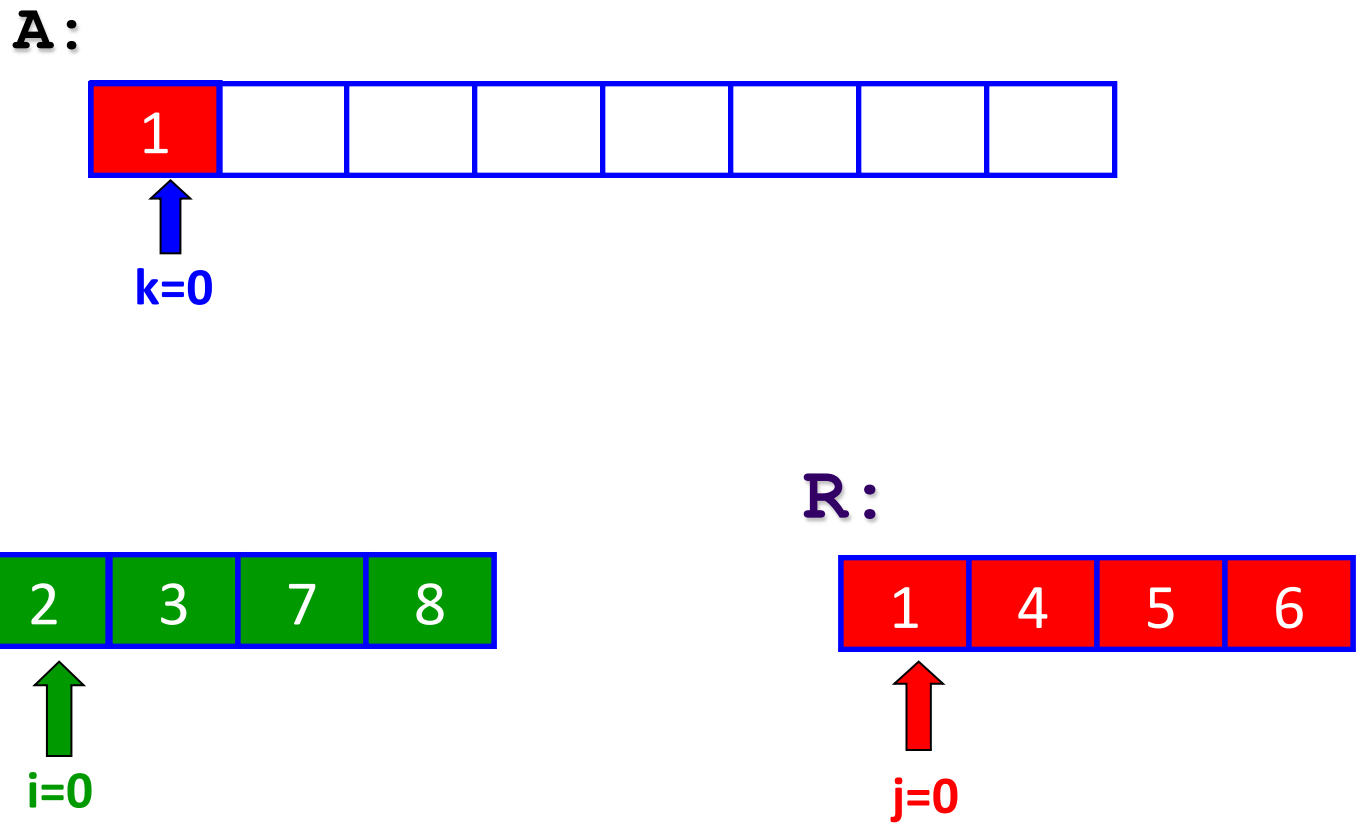
Merge-Sort: Merge



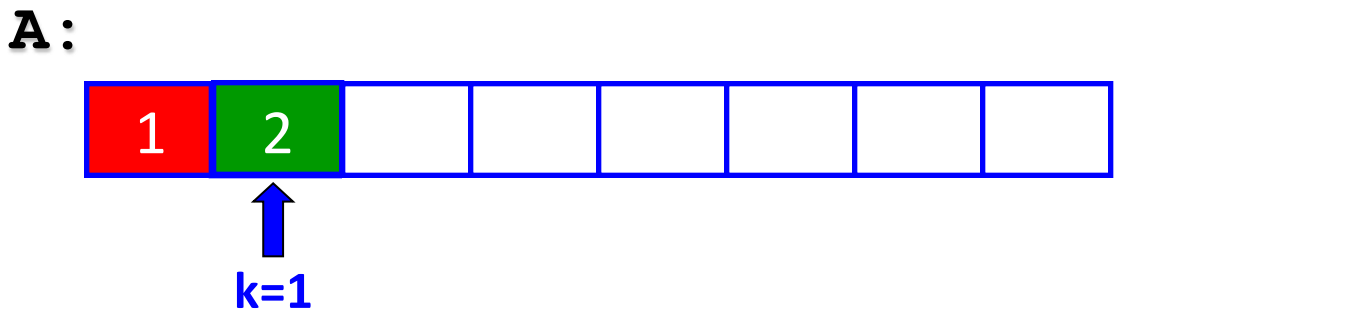
Merge-Sort: Merge Example

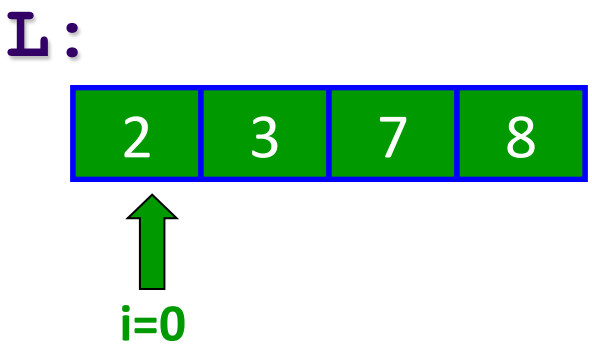


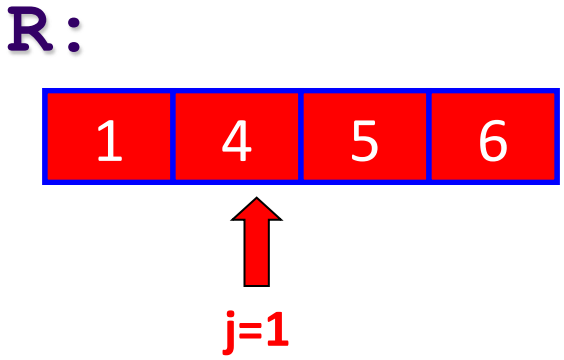
Merge-Sort: Merge Example



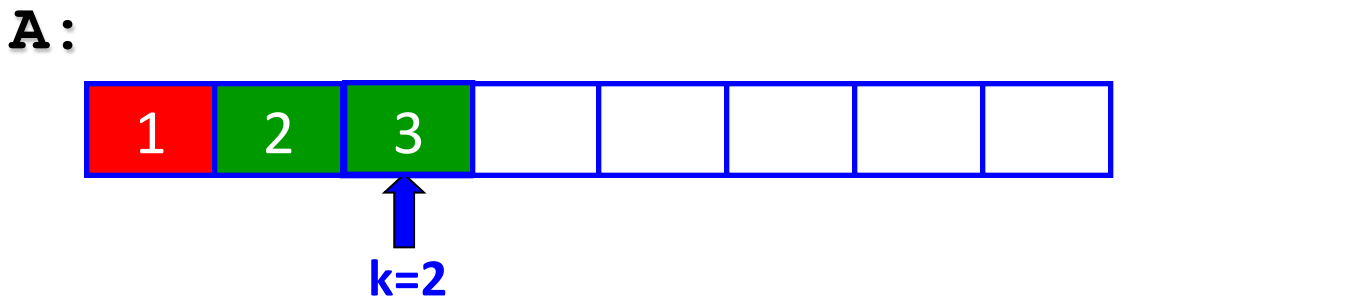
Merge-Sort: Merge Example

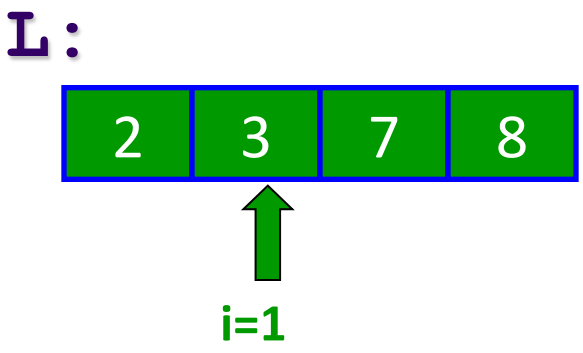


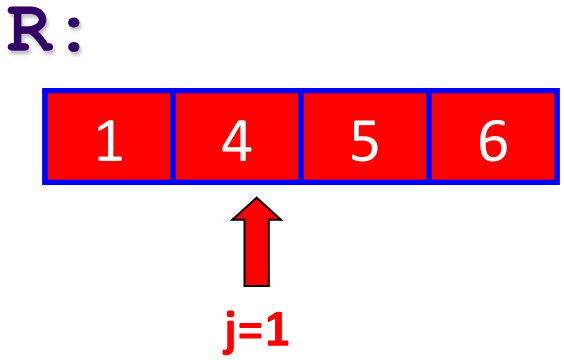




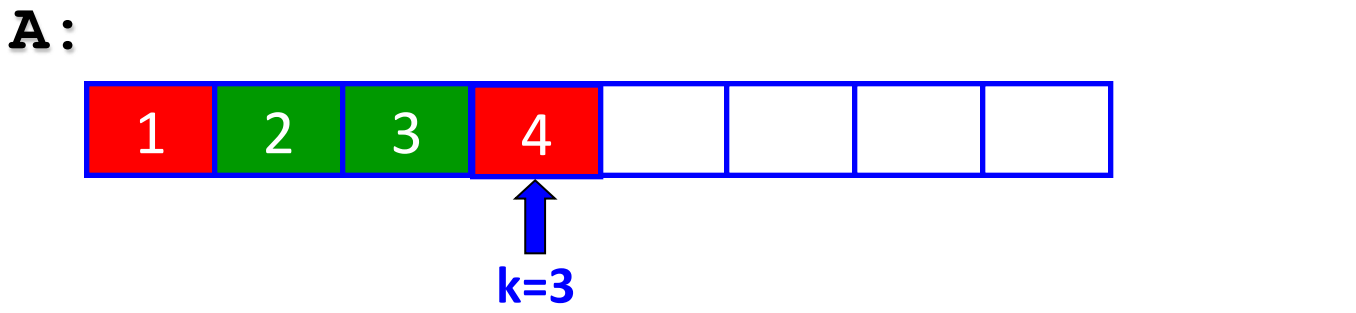
Merge-Sort: Merge Example

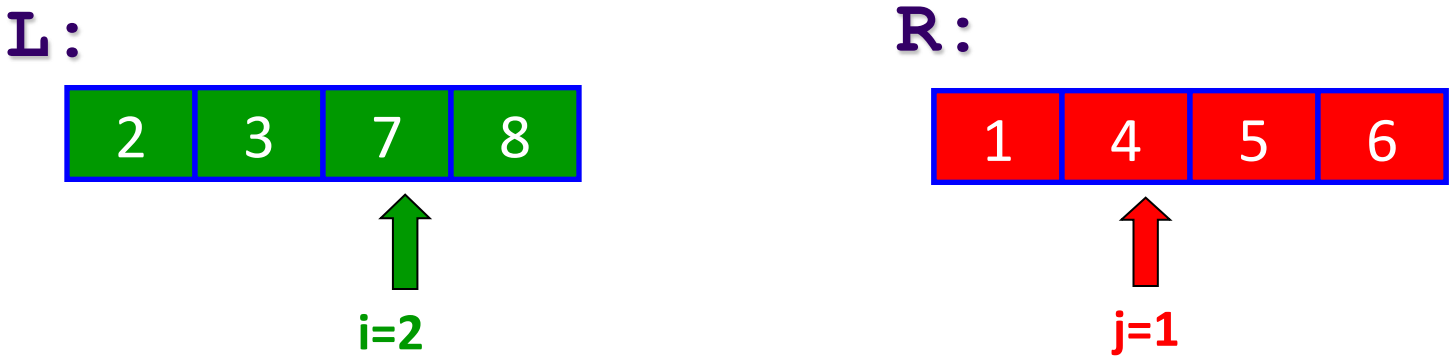




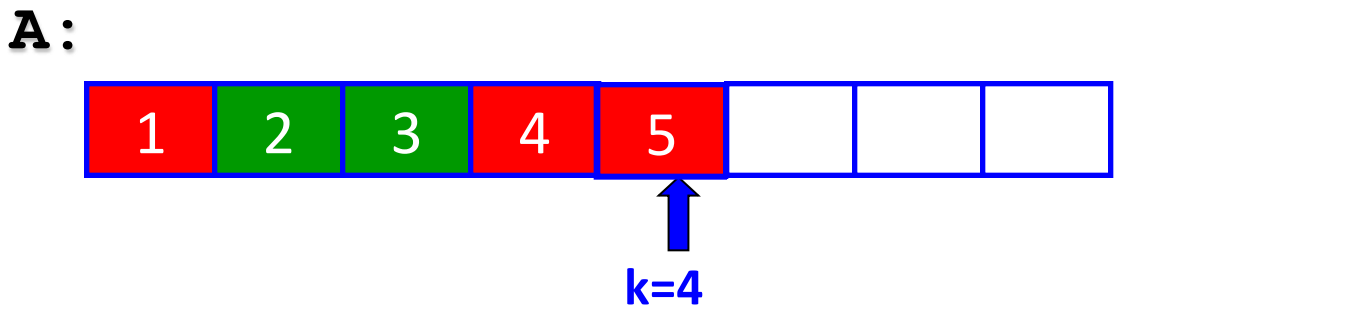


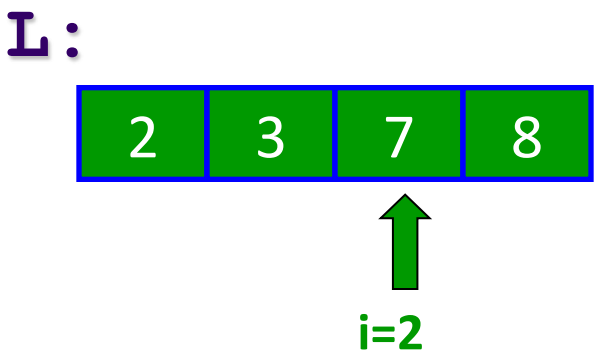
Merge-Sort: Merge Example

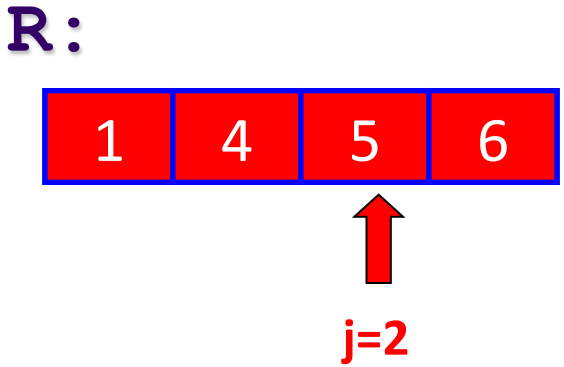




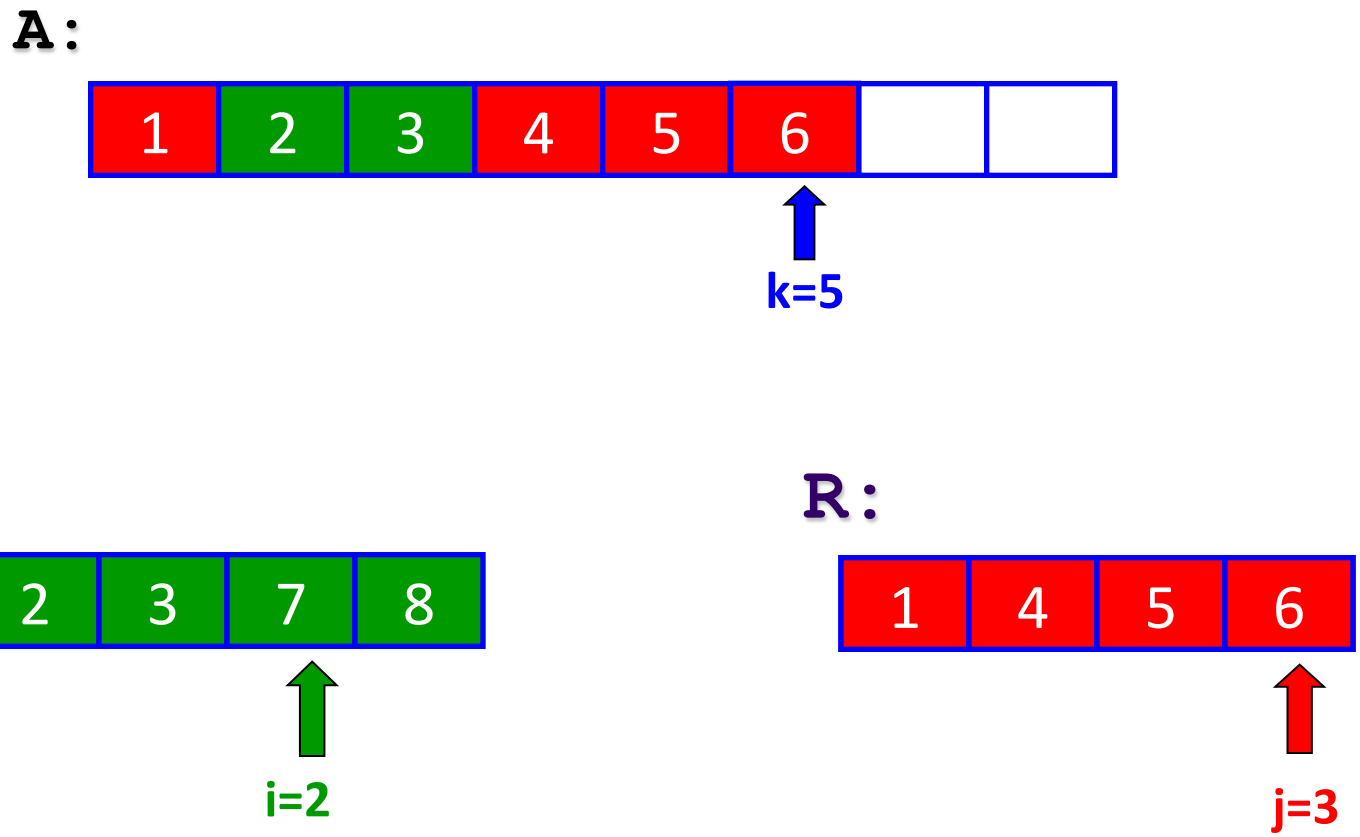
Merge-Sort: Merge Example



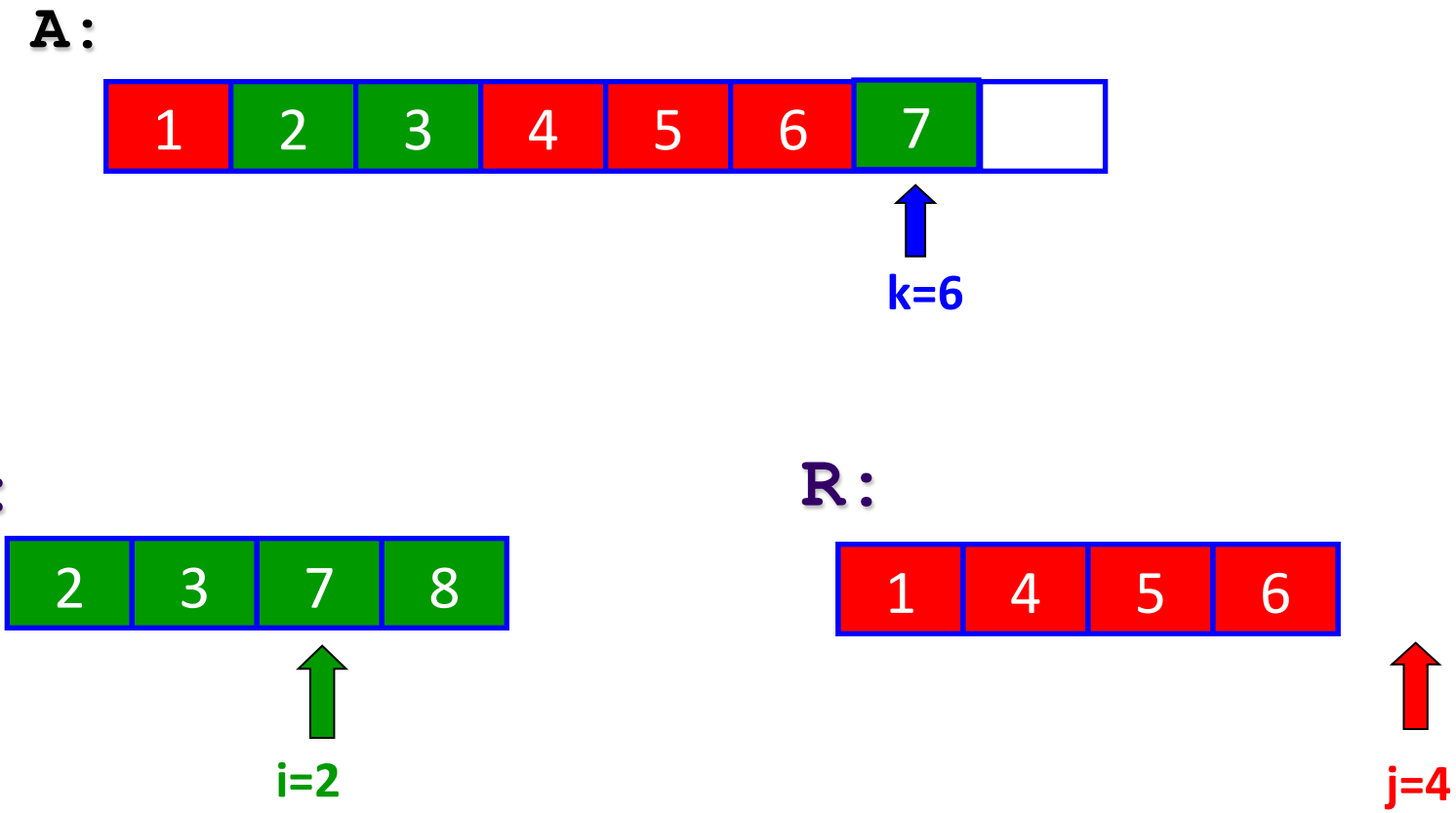




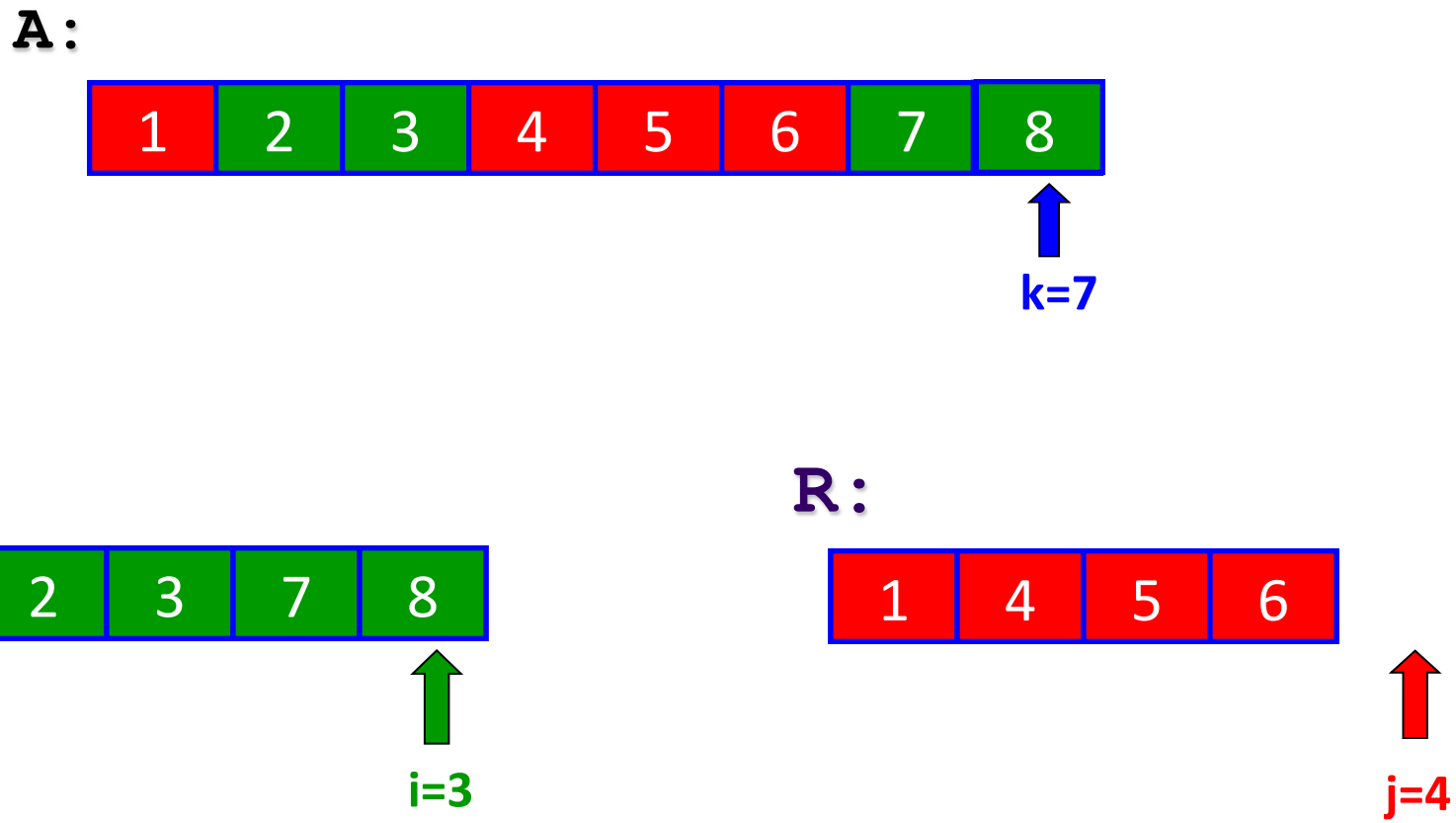
Merge-Sort: Merge Example



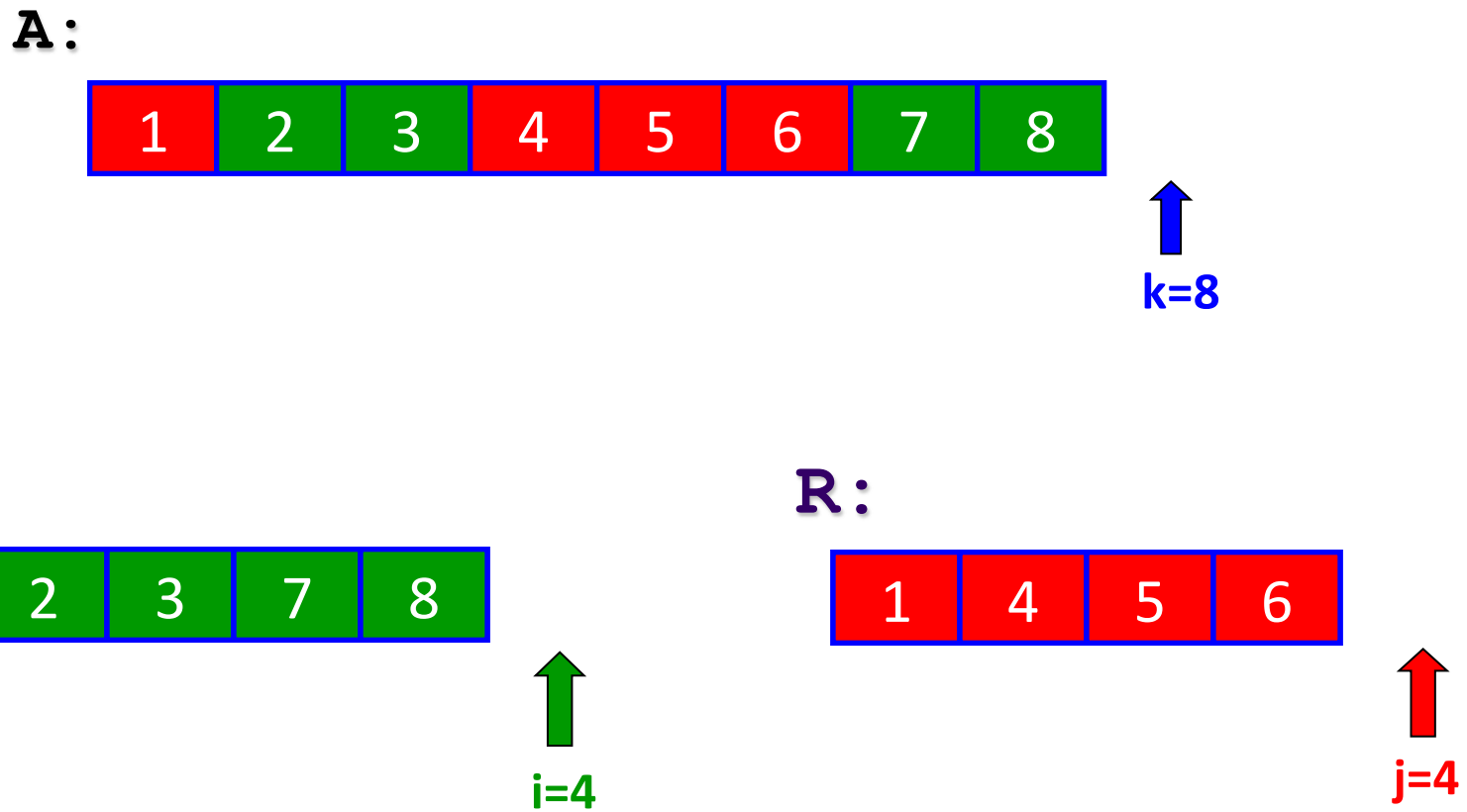
Merge-Sort: Merge Example



Merge-Sort: Merge Example



Merge-Sort: Merge Example



Key (Simple) Idea

To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

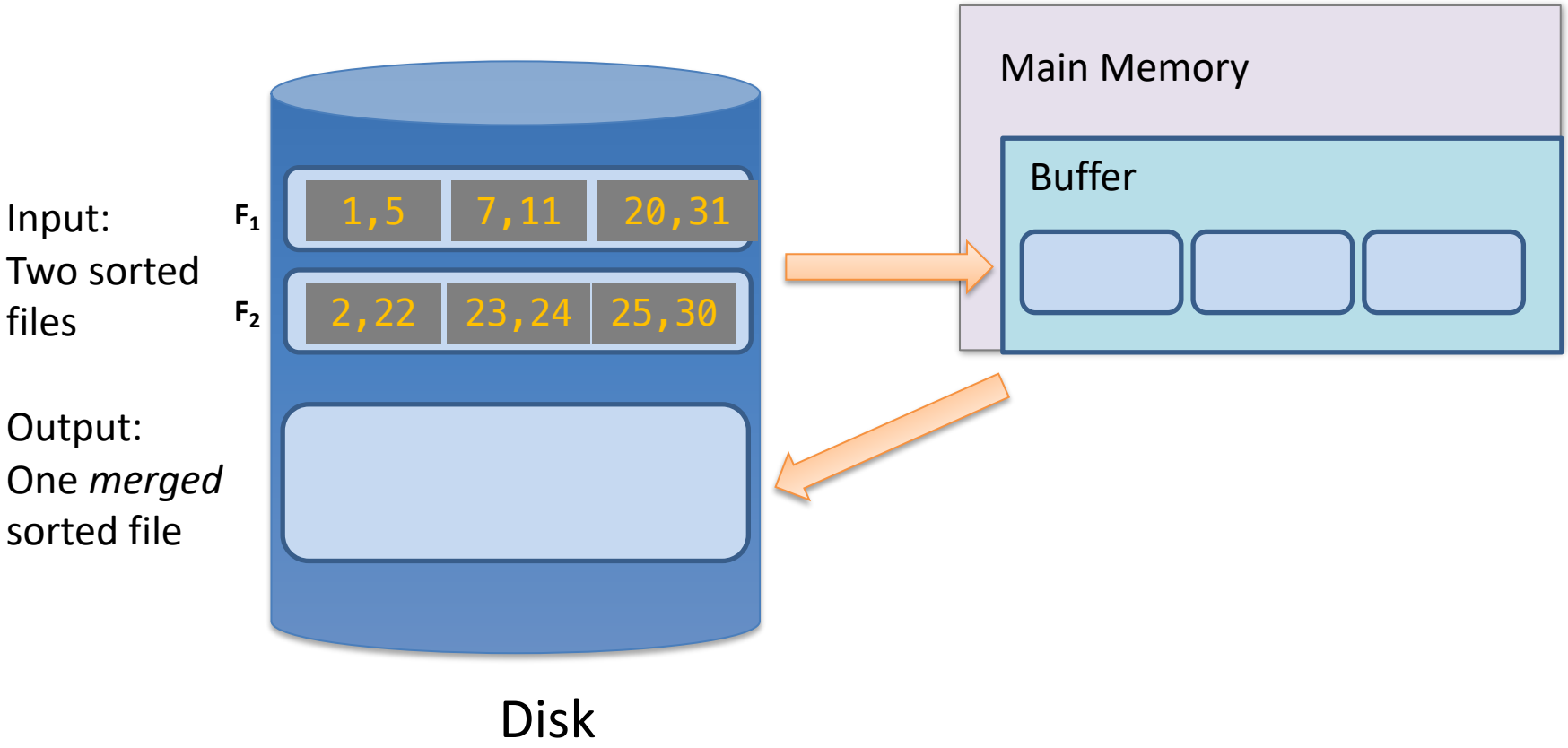
$$A_1 \leq A_2 \leq \cdots \leq A_N$$
$$B_1 \leq B_2 \leq \cdots \leq B_M$$

Then:

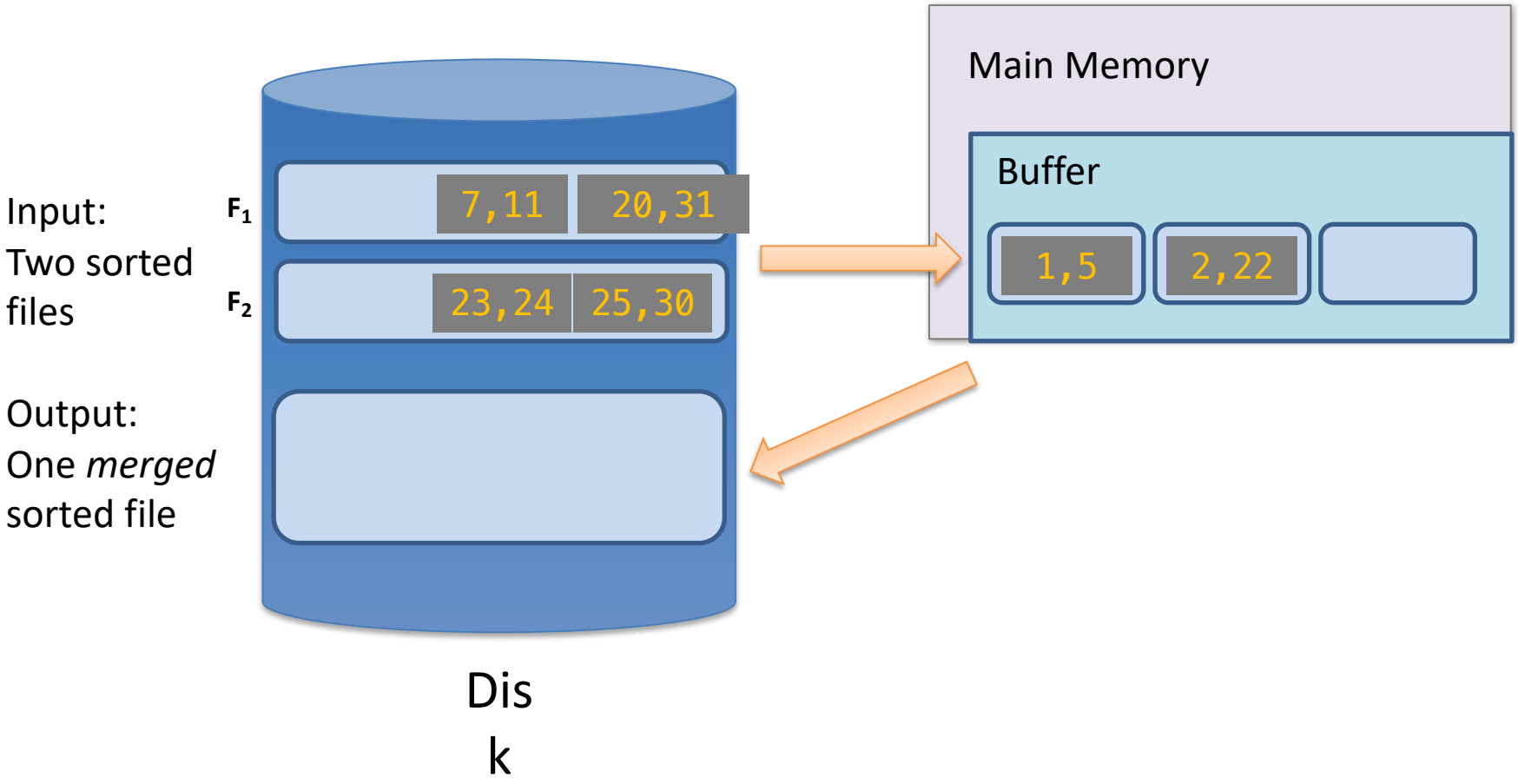
$$Min(A_1, B_1) \leq A_i$$
$$Min(A_1, B_1) \leq B_j$$

for i=1….N and j=1….M

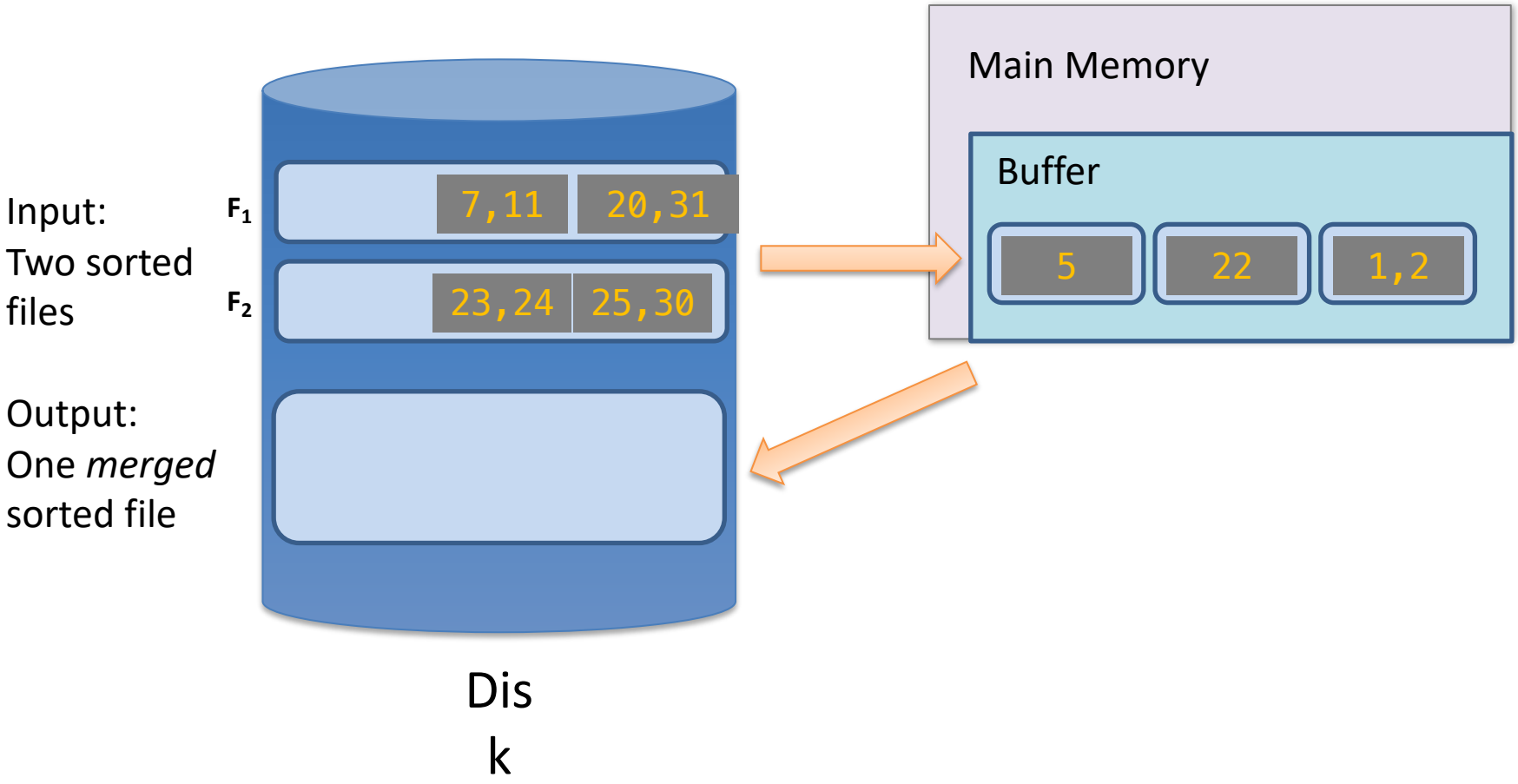
External Merge Algorithm



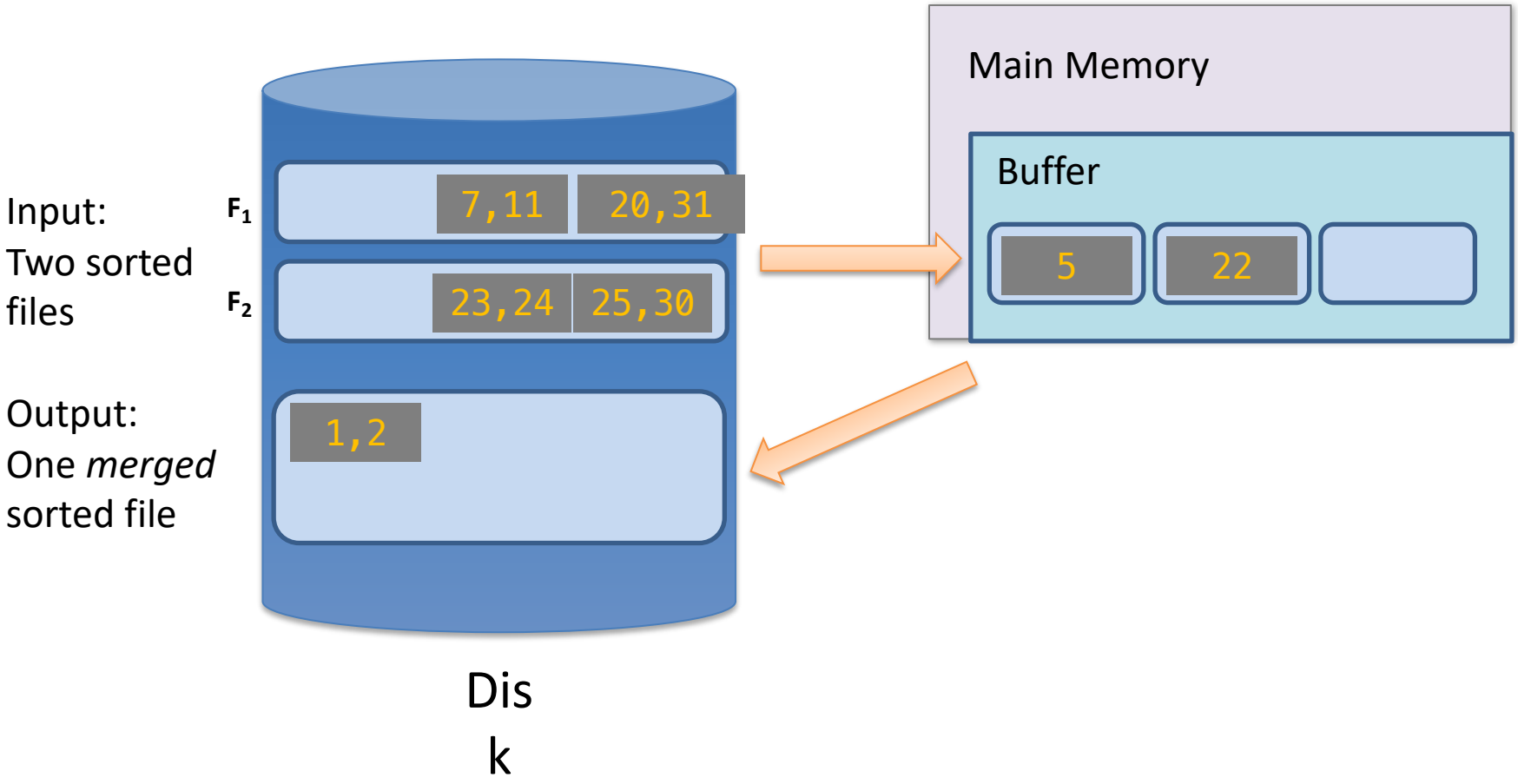
External Merge Algorithm



External Merge Algorithm



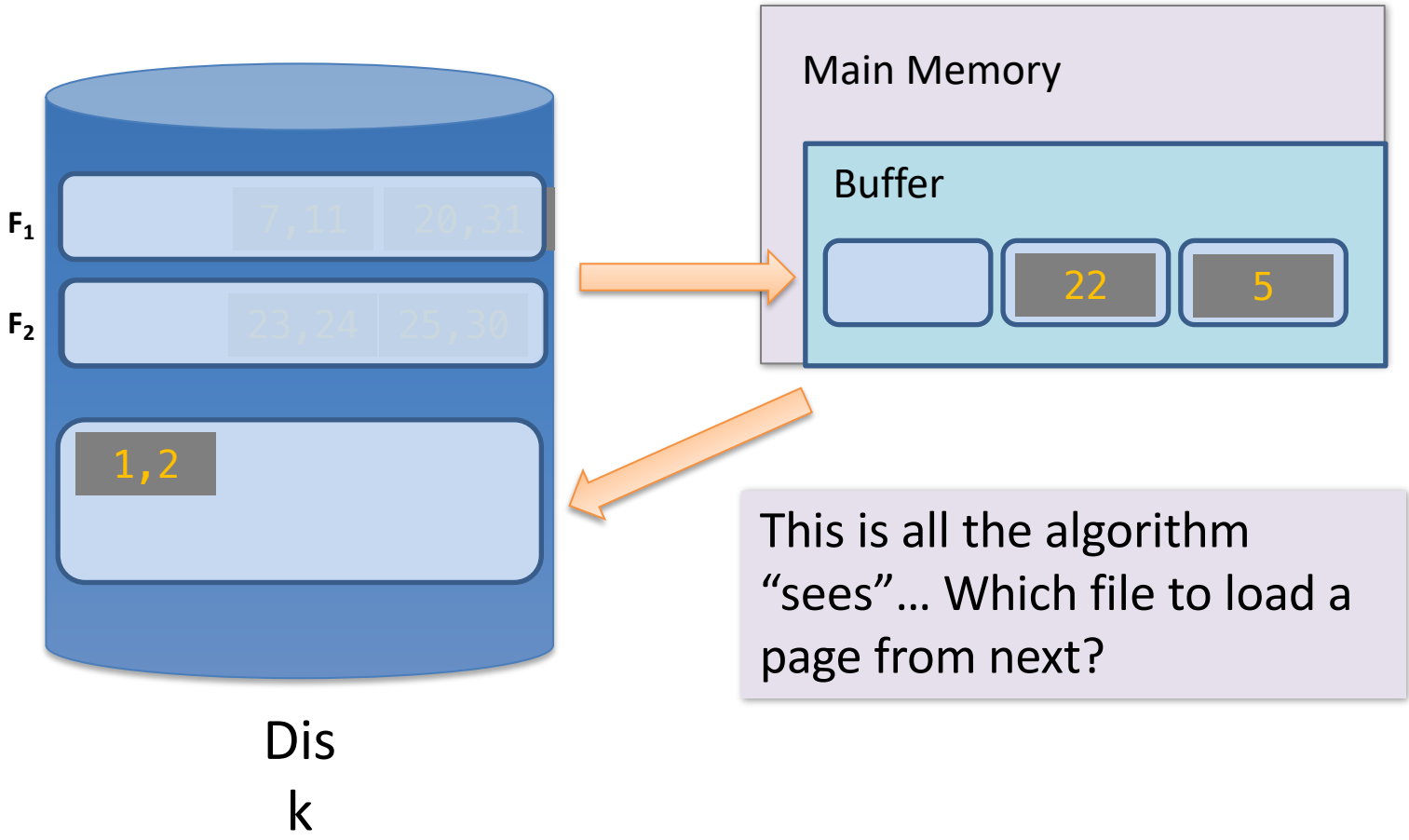
External Merge Algorithm



External Merge Algorithm

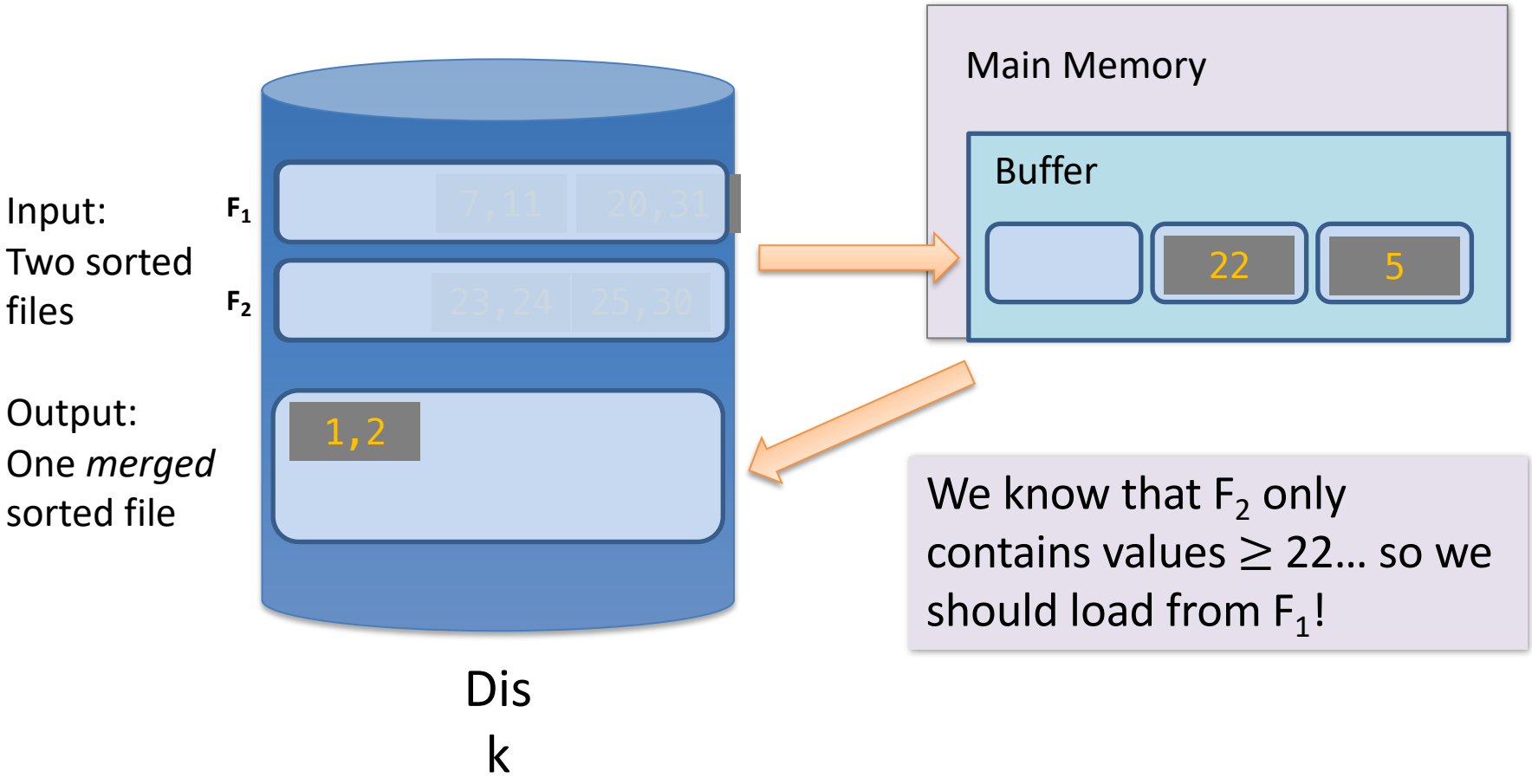
Input:
Two sorted files

Output:
One *merged* sorted file

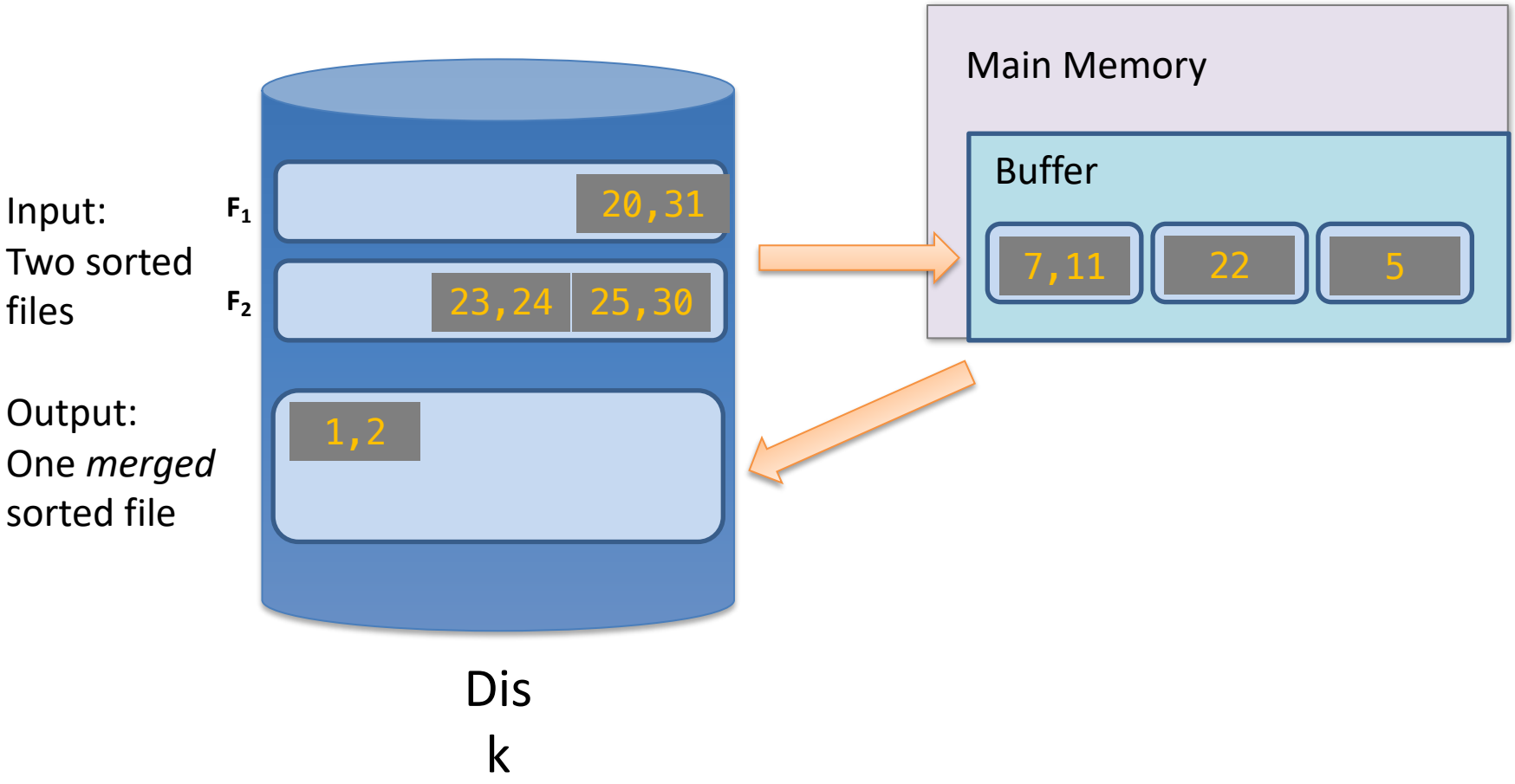


This is all the algorithm "sees"… Which file to load a page from next?

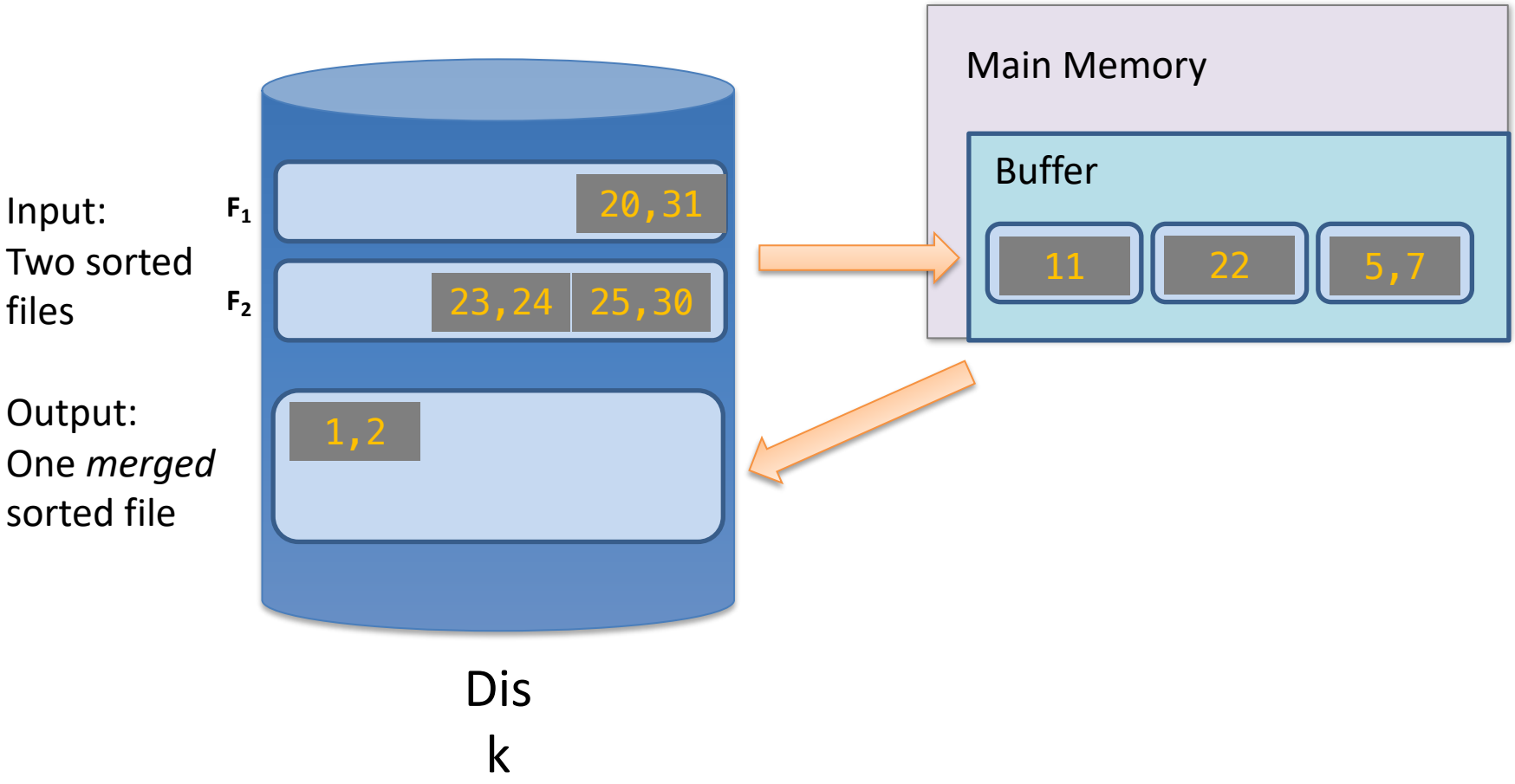
External Merge Algorithm



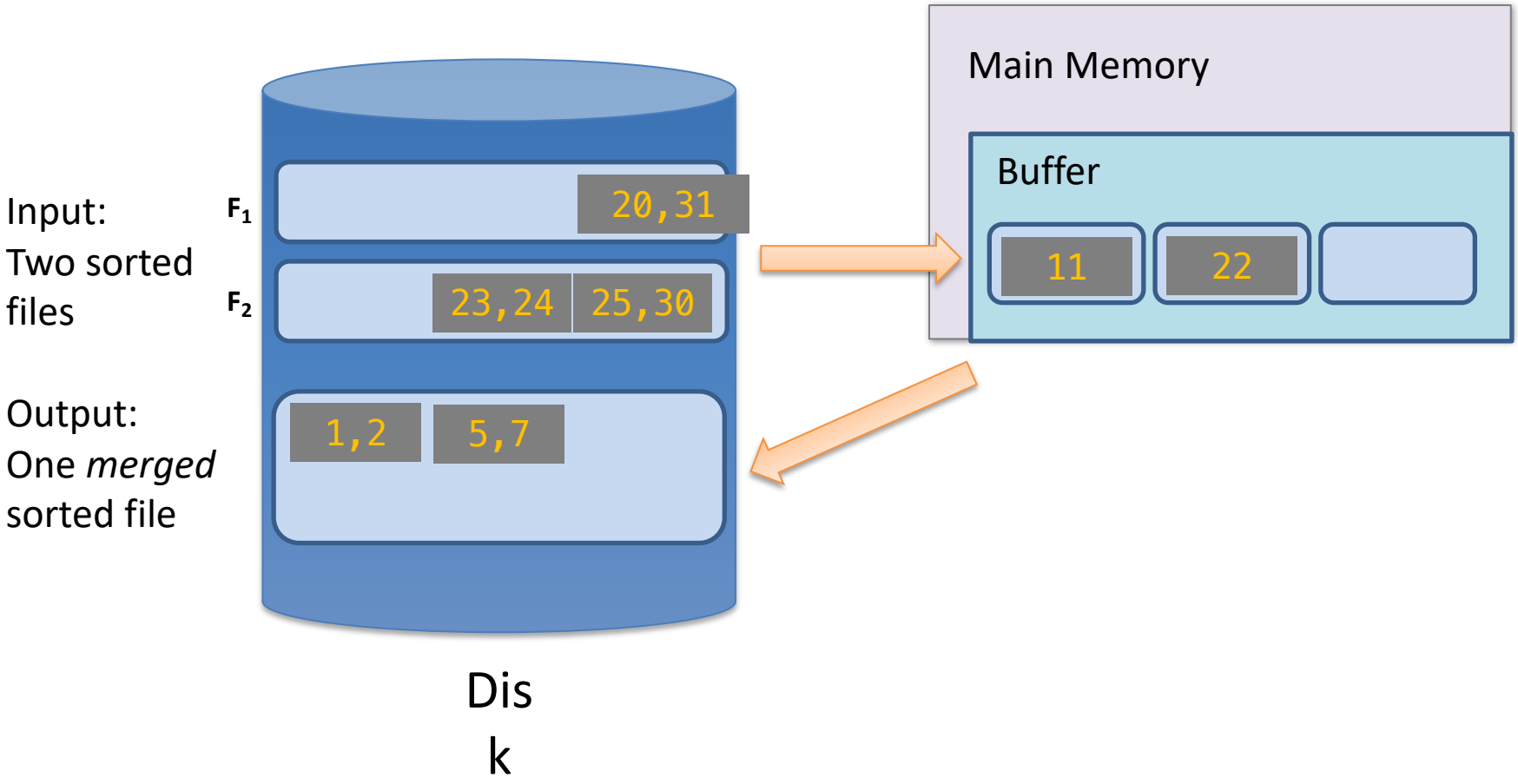
External Merge Algorithm



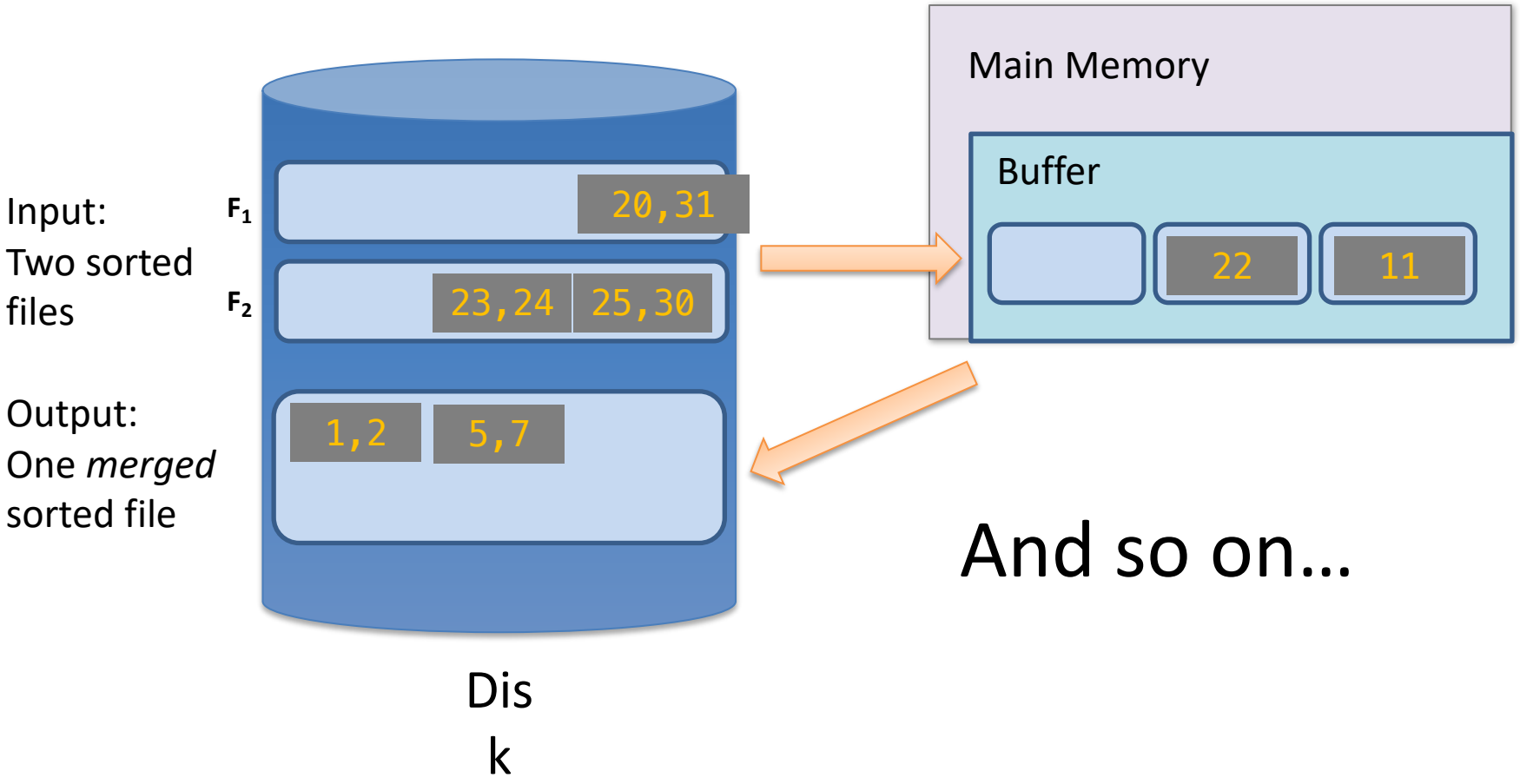
External Merge Algorithm



External Merge Algorithm



External Merge Algorithm



We can merge lists of **arbitrary length** with *only* 3 buffer pages.

If lists of size M and N, then

Cost: 2(M+N) IOs

Each page is read once, written once

Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.