Sorting with Divide and Conquer Merge Sort

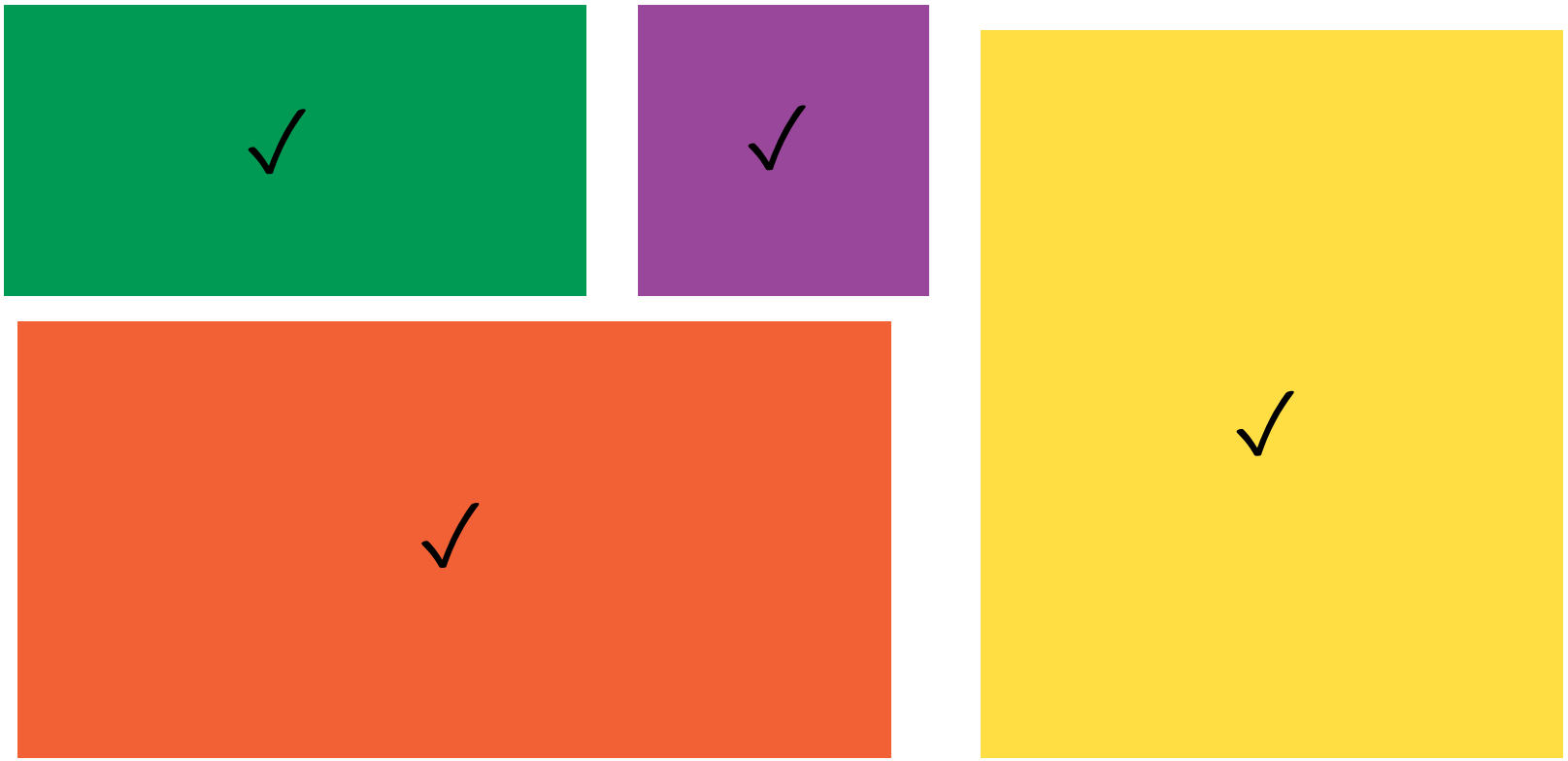
Lecture 18

Divide-and-conquer technique

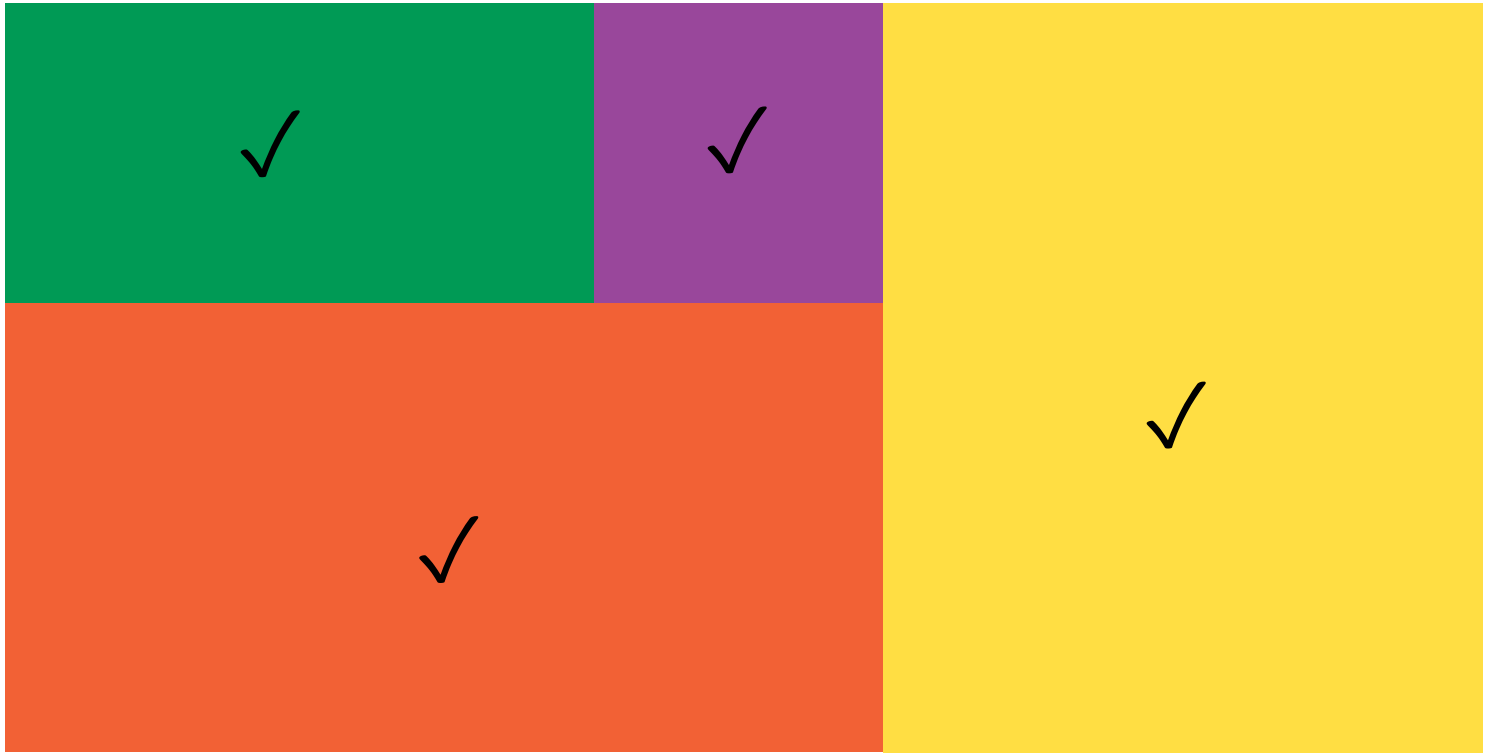
1. Break into *non-overlapping* subproblems *of the same type*
2. Solve subproblems
3. Combine results

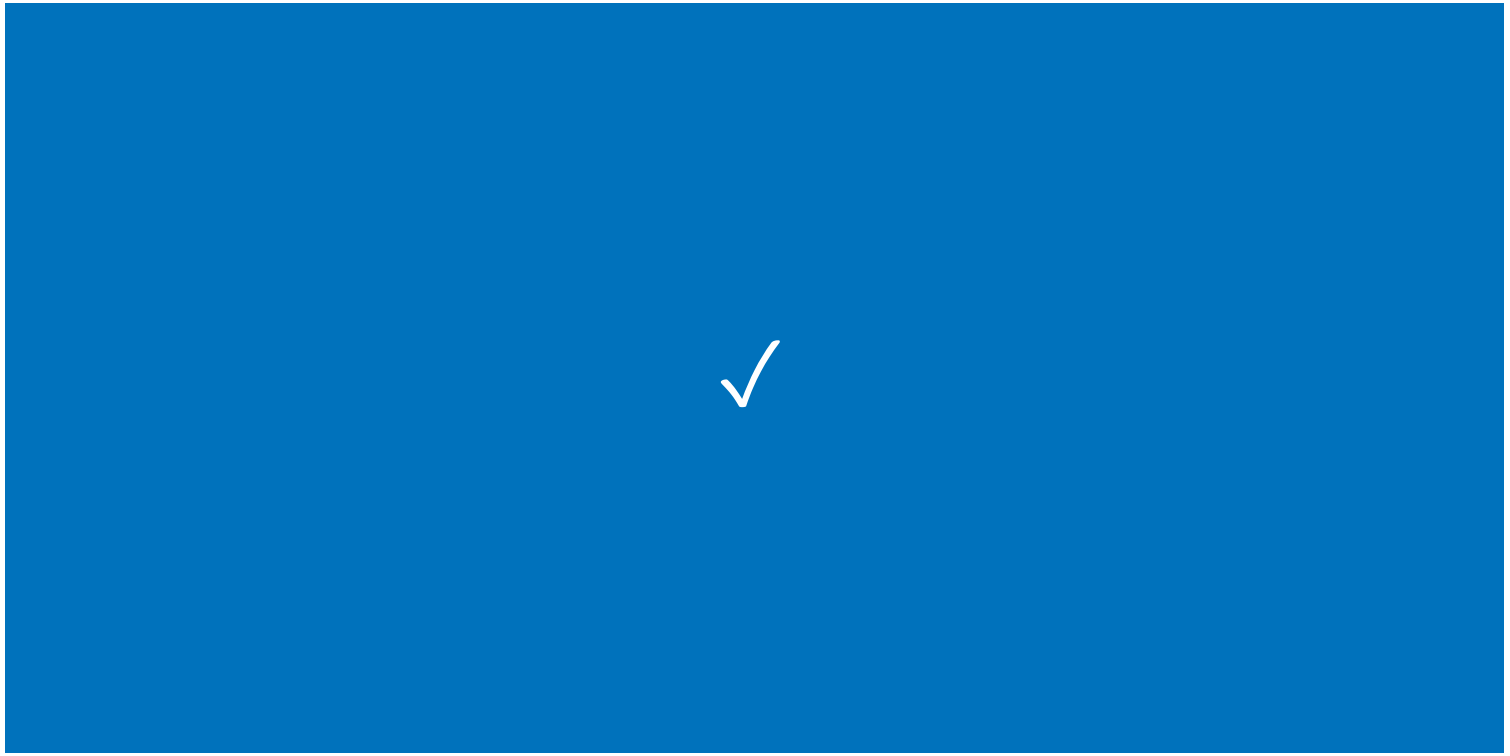
Divide: break

Conquer: solve

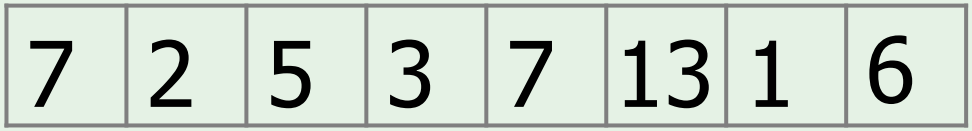


Combine

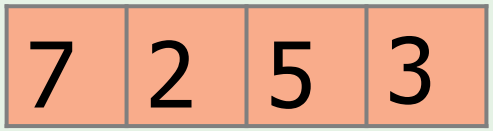




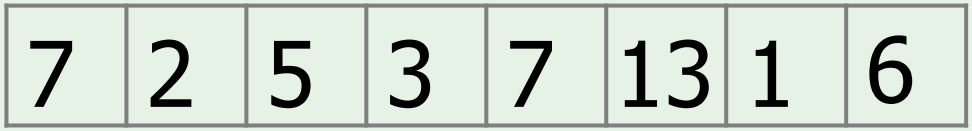
Idea: merge sort



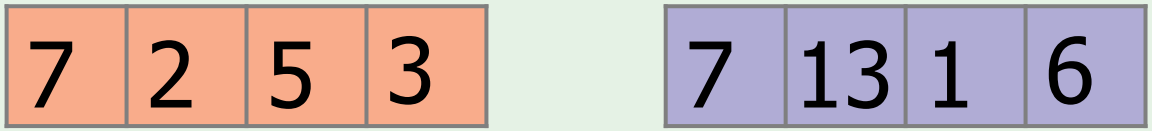
split the array into two halves



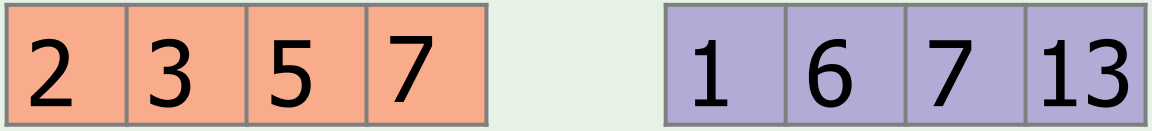
Idea: merge sort



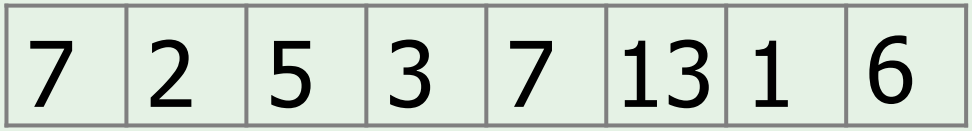
split the array into two halves



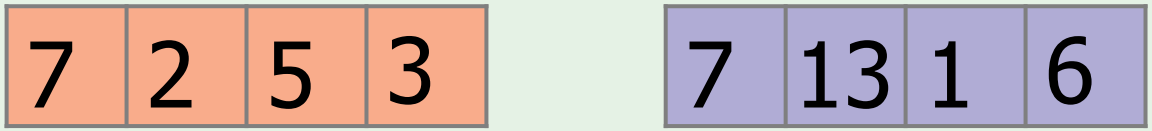
sort the halves recursively



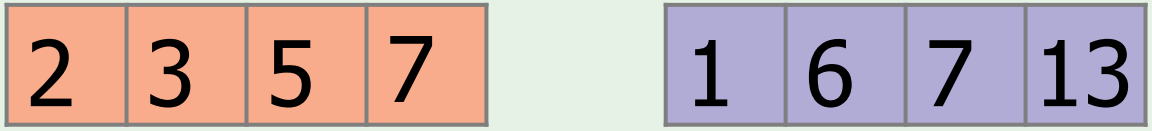
Idea: merge sort



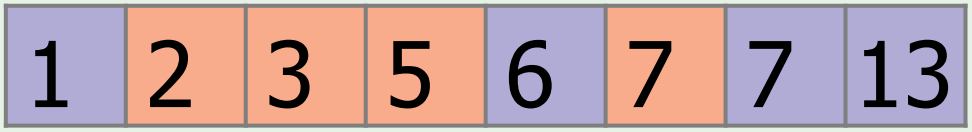
split the array into two halves



sort the halves recursively



merge the sorted halves into one array



Algorithm MergeSort (array $A[1...n]$)

if $n = 1$: return A *# already sorted*

$m \leftarrow \lfloor n/2 \rfloor$

$B \leftarrow$ MergeSort($A[1 \ldots m]$)

$C \leftarrow$ MergeSort($A[m + 1 \ldots n]$)

$A' \leftarrow$ merge(B, C)

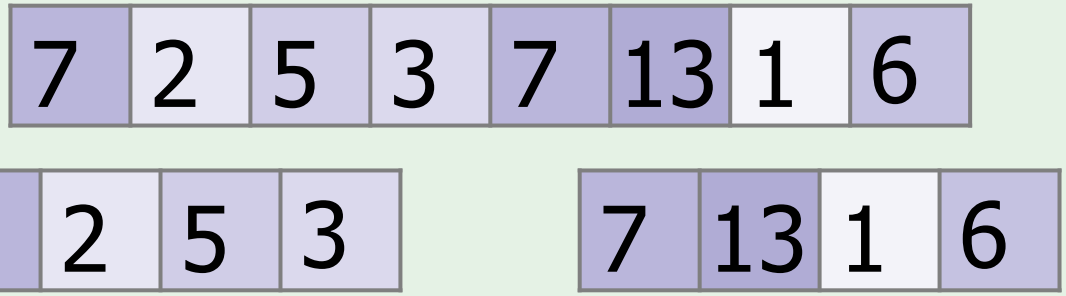
return A'

Merging Two Sorted Arrays

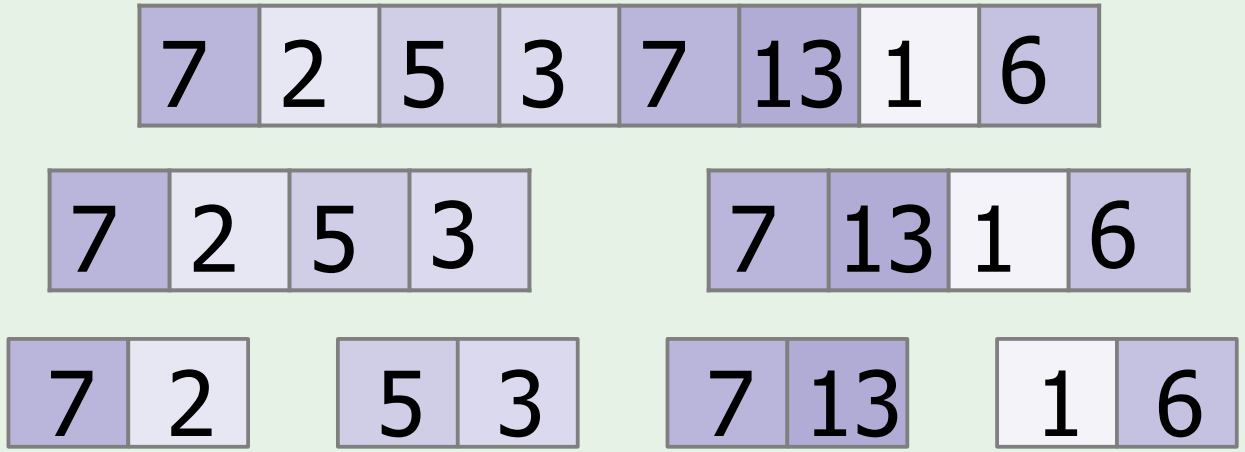
Algorithm Merge($B[1...p]$, $C[1...q]$)

```
# B and C are sorted
D ← empty array of size p + q
while B and C are both non-empty:
    b ← the first element of B
    c ← the first element of C
    if b ≤ c:
        move b from B to the end of D
    else:
        move c from C to the end of D
move what remains of B or C to the end of D
return D
```

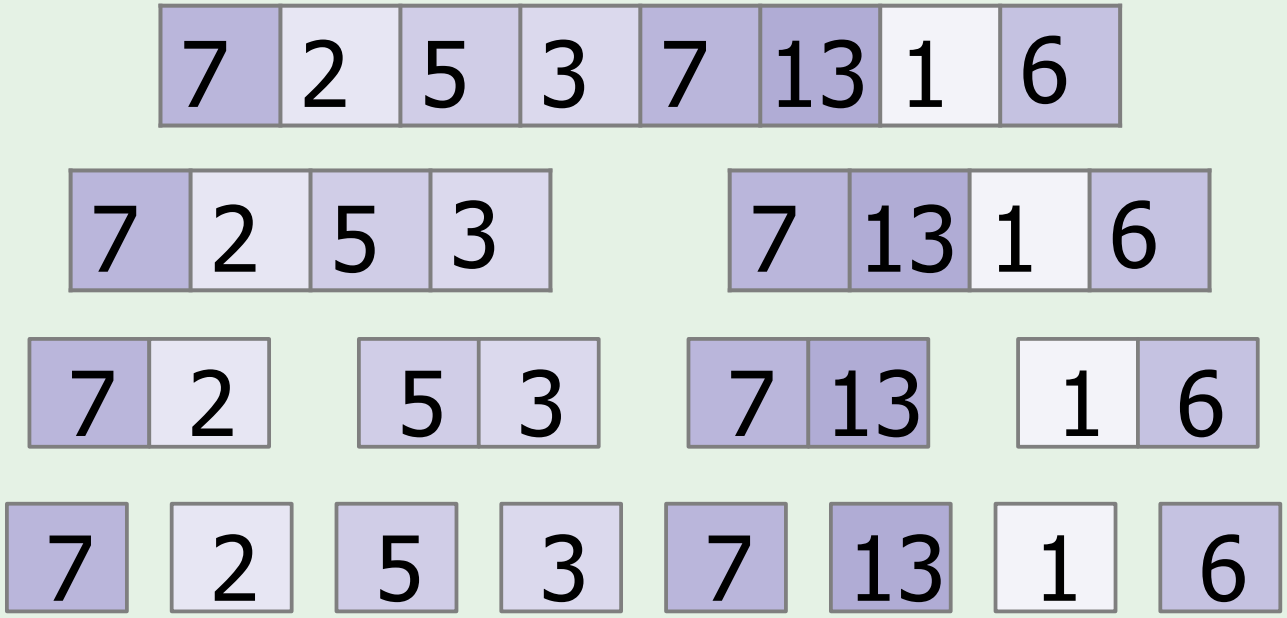
Merge sort: example



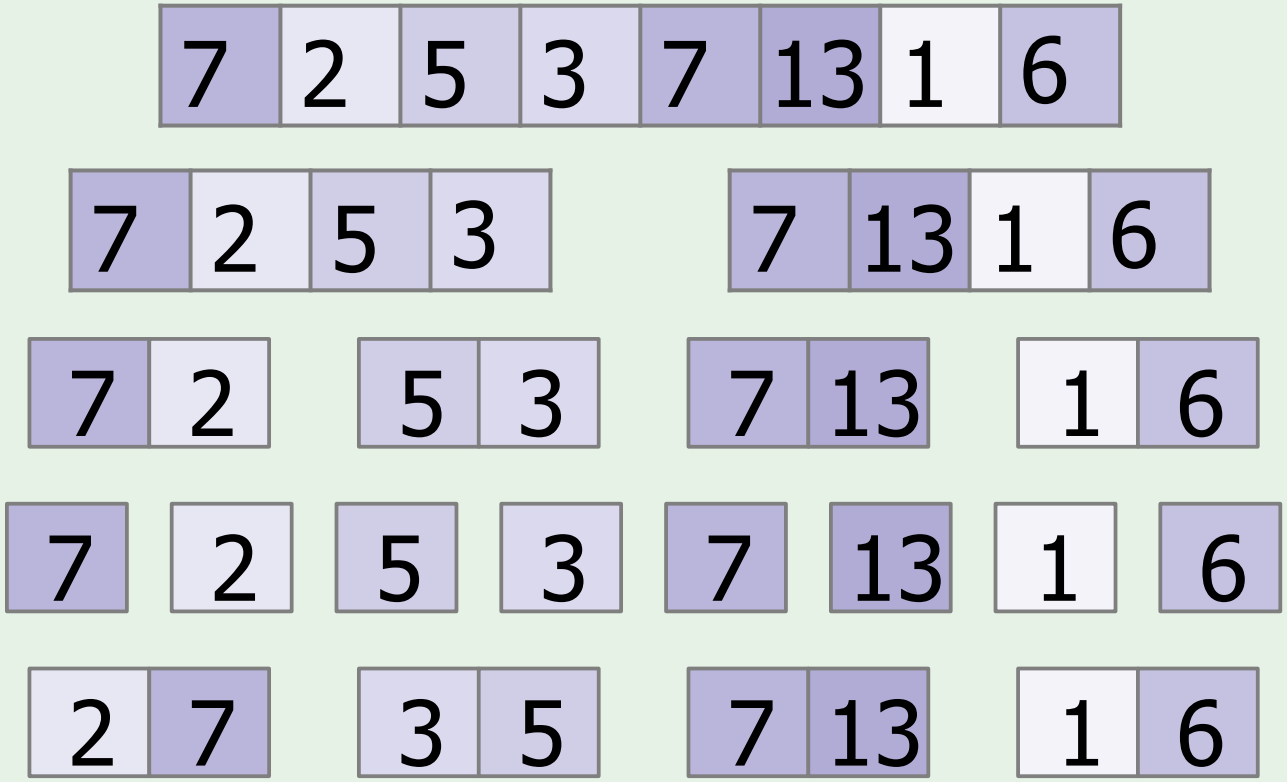
Merge sort: example



Merge sort: example

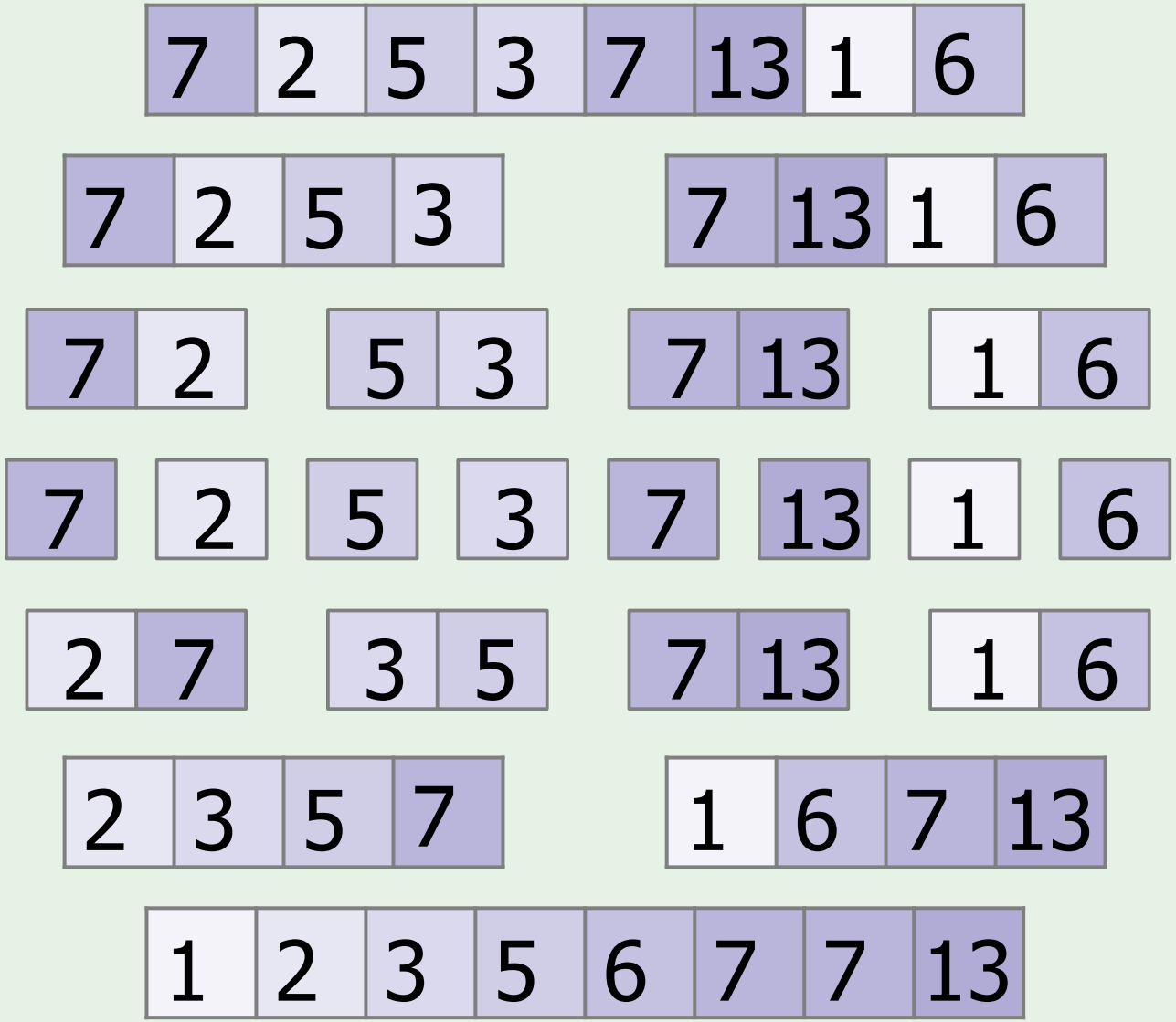


Merge sort: example

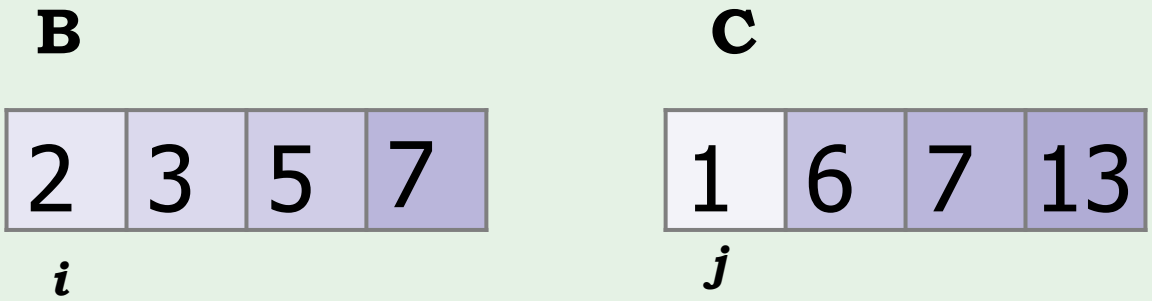


Merge sort: example

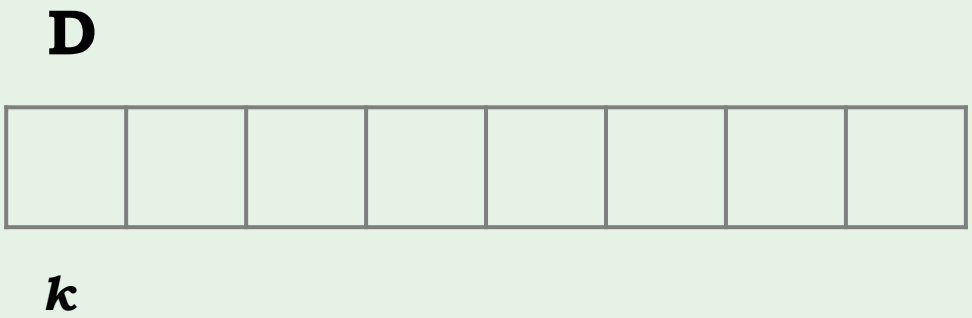
Merge sort: example



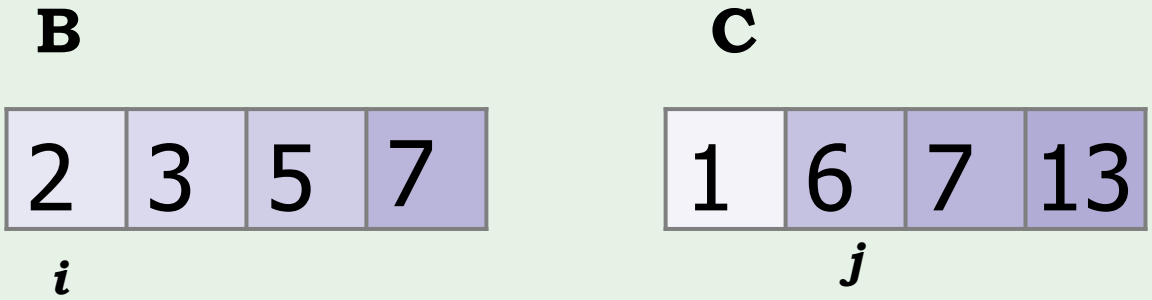
Merge: example



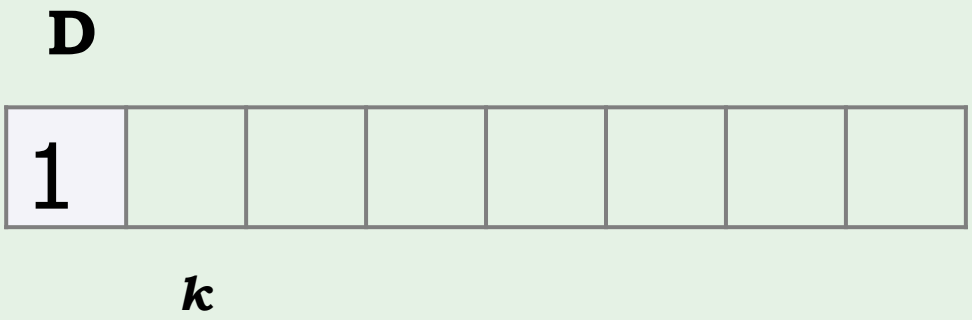
Compare **B[i]** and **C[j]**



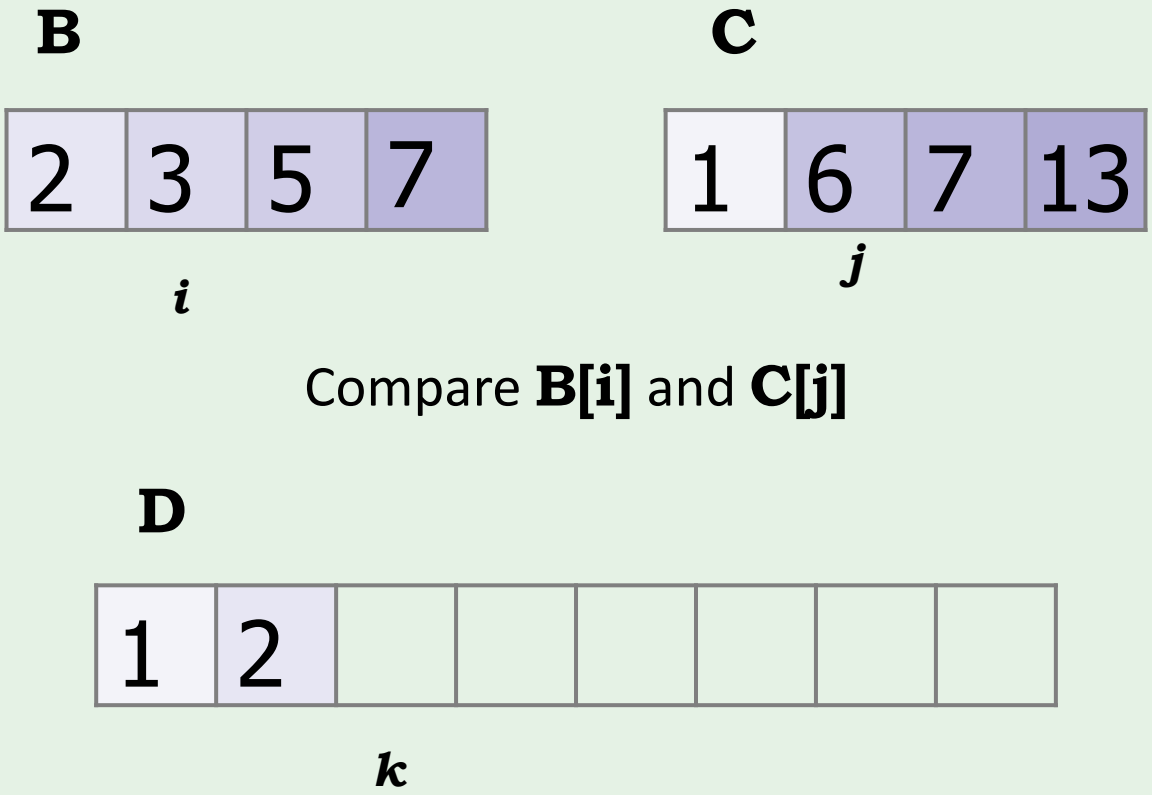
Merge: example



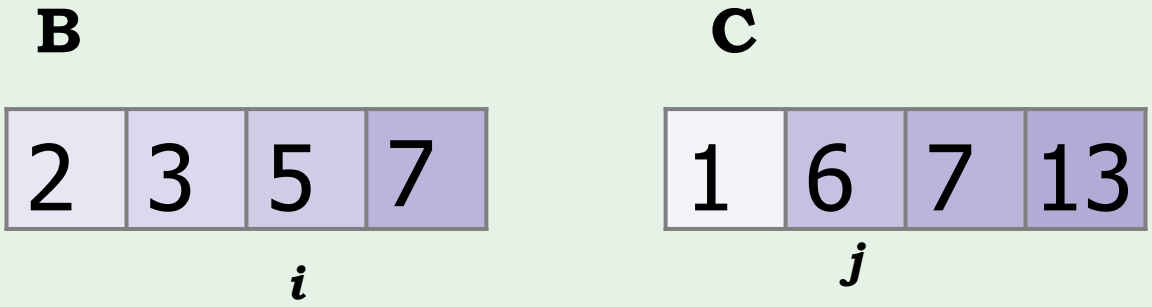
Compare **B[i]** and **C[j]**



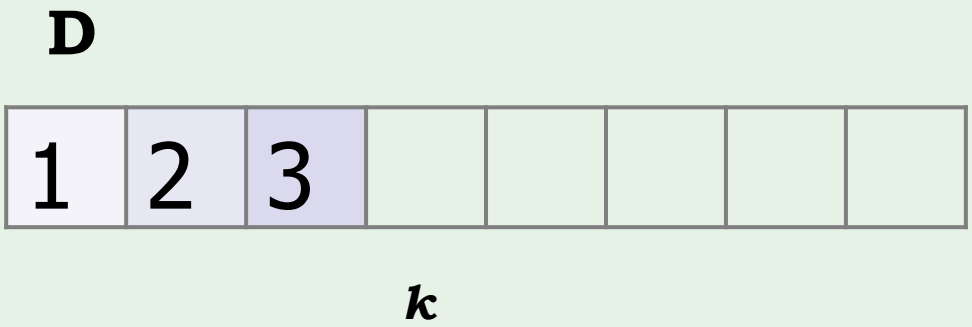
Merge: example



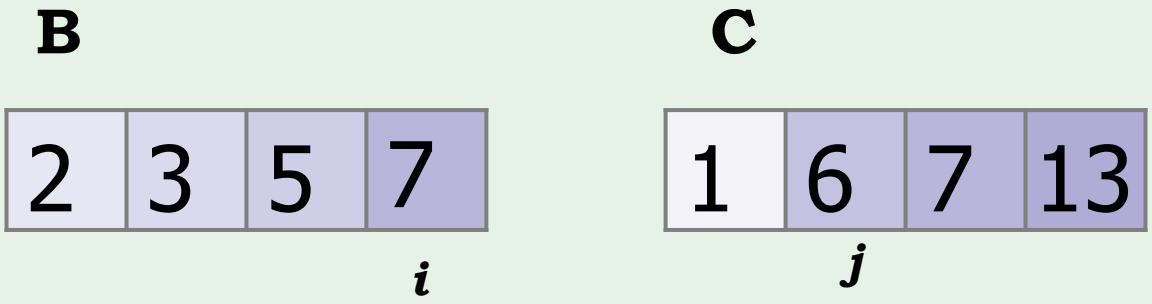
Merge: example



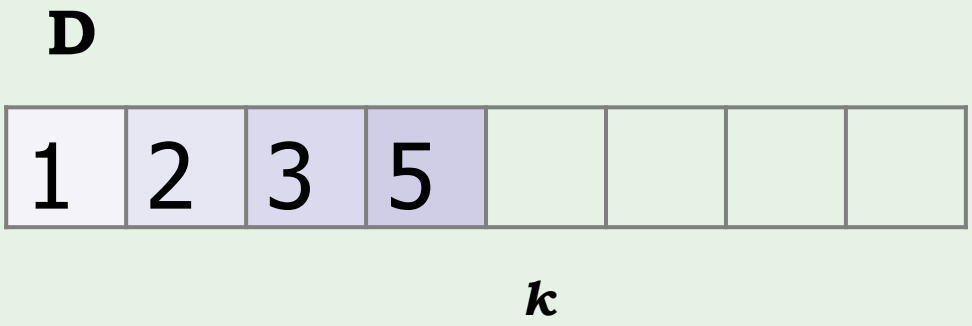
Compare **B[i]** and **C[j]**



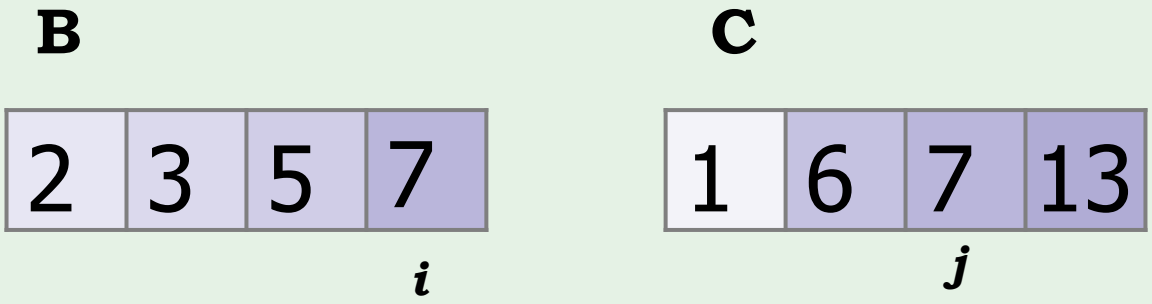
Merge: example



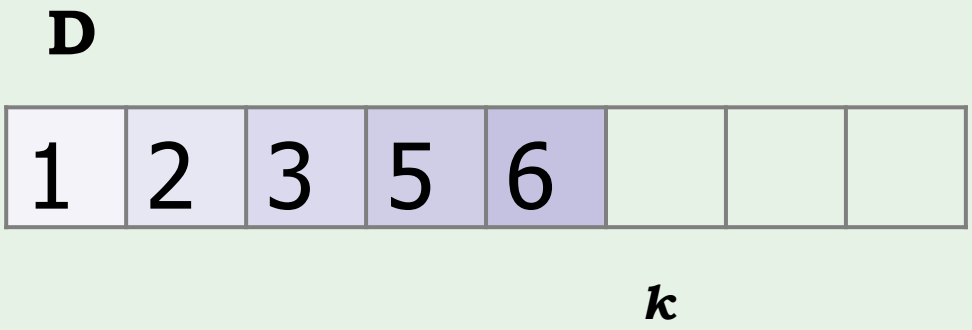
Compare **B[i]** and **C[j]**



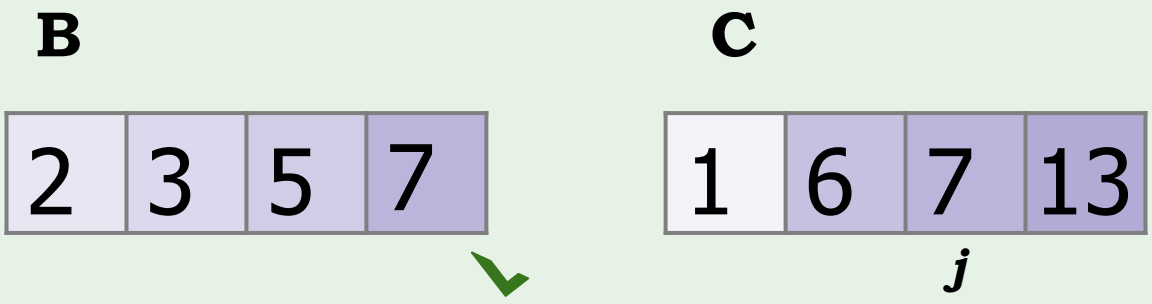
Merge: example



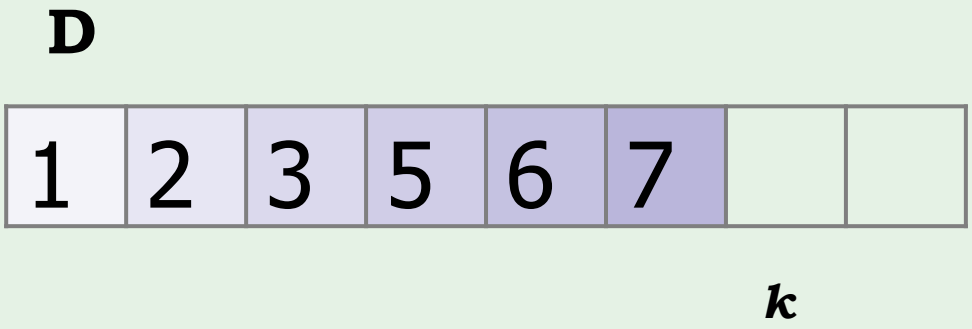
Compare **B[i]** and **C[j]**



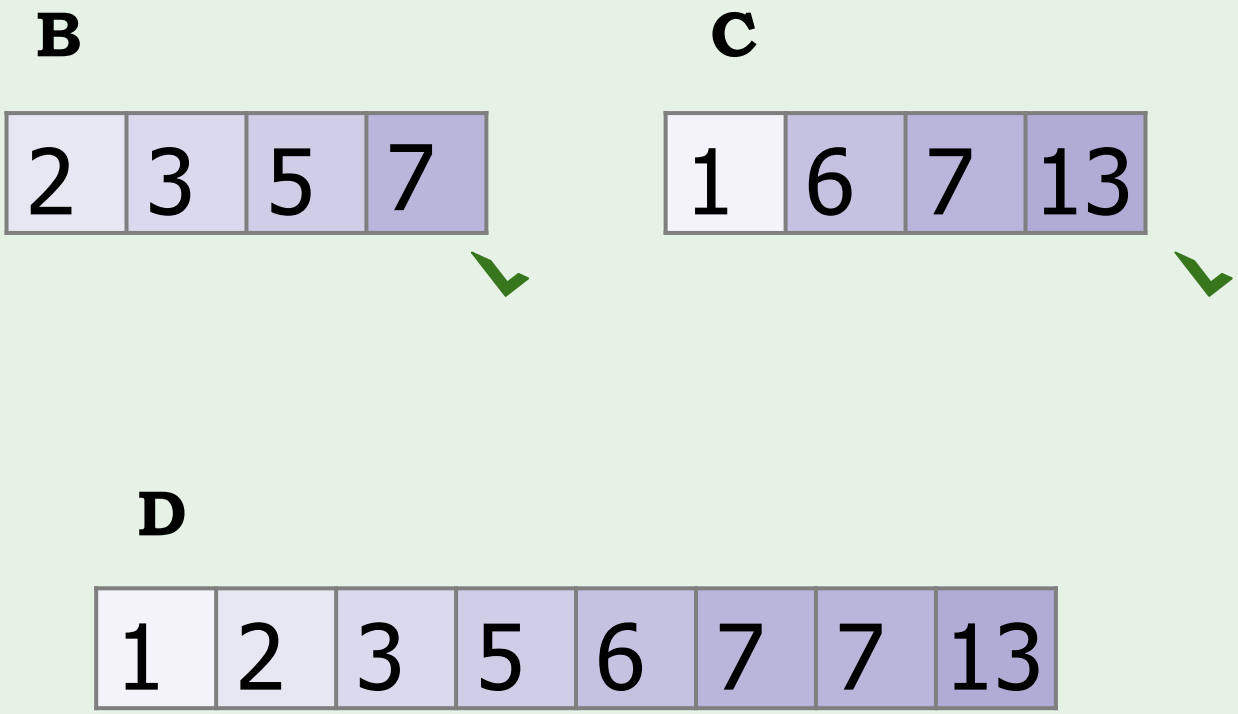
Merge: example



Copy what remains in **C**

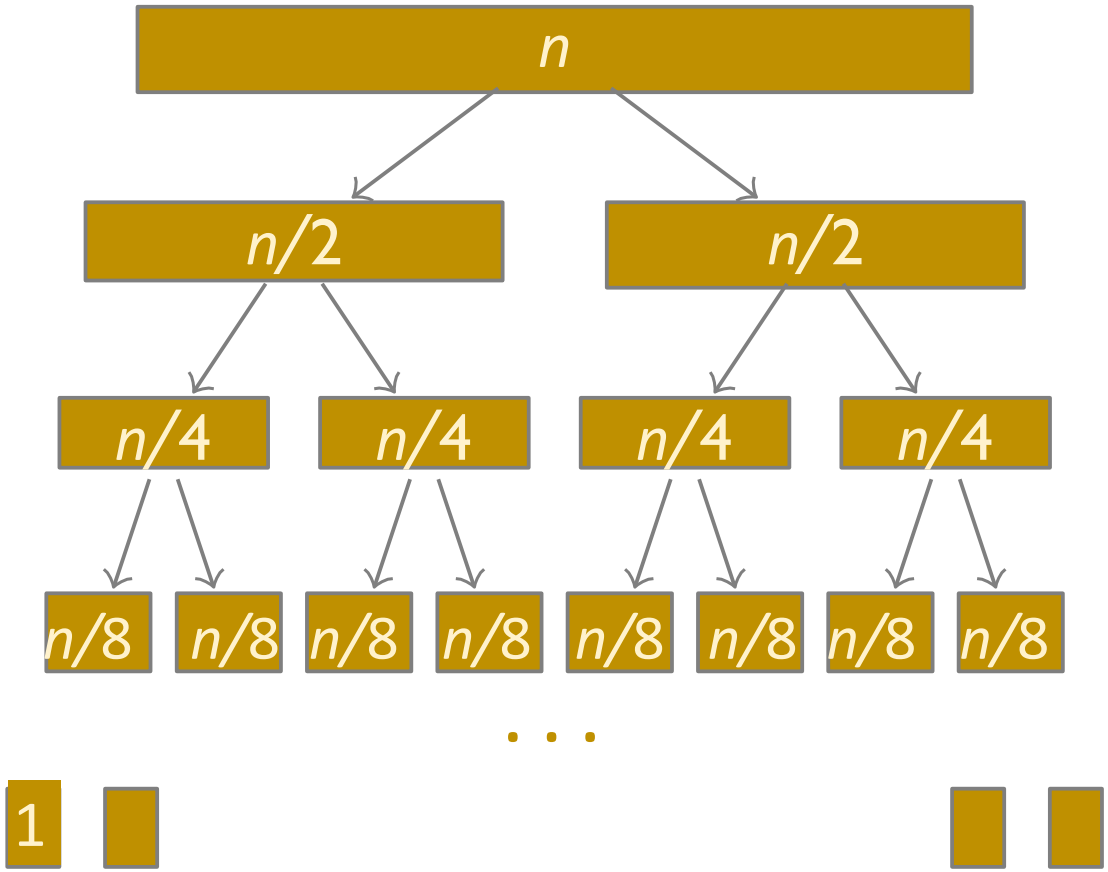


Merge: example

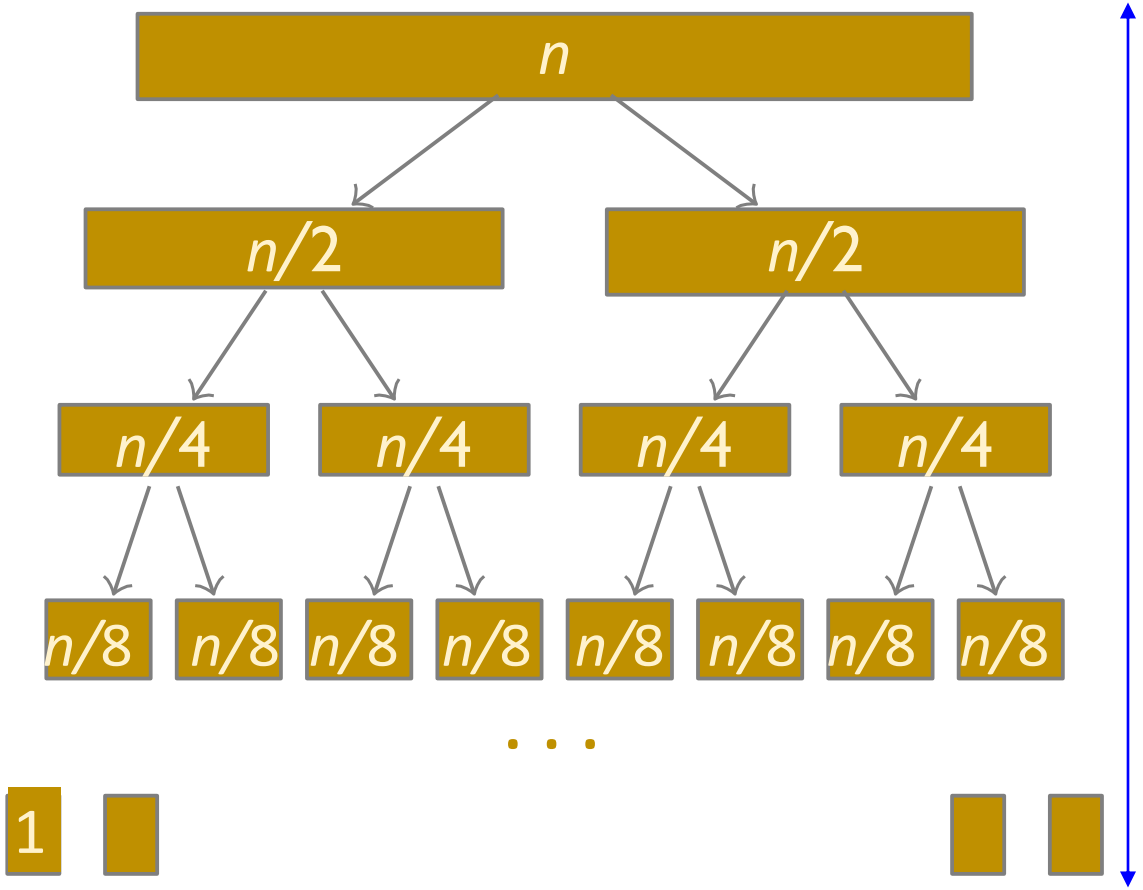


Merge sort: running time

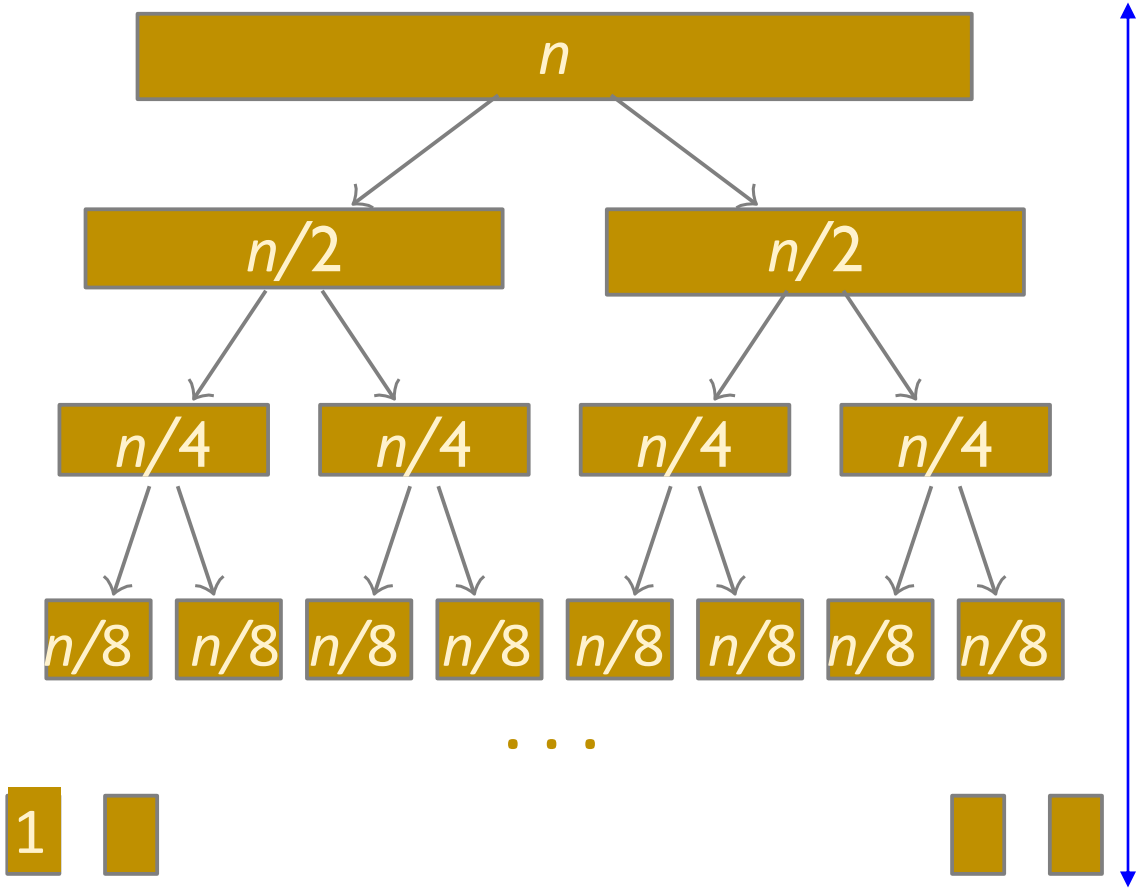
Subproblem size at each level



Merge sort: recursion tree



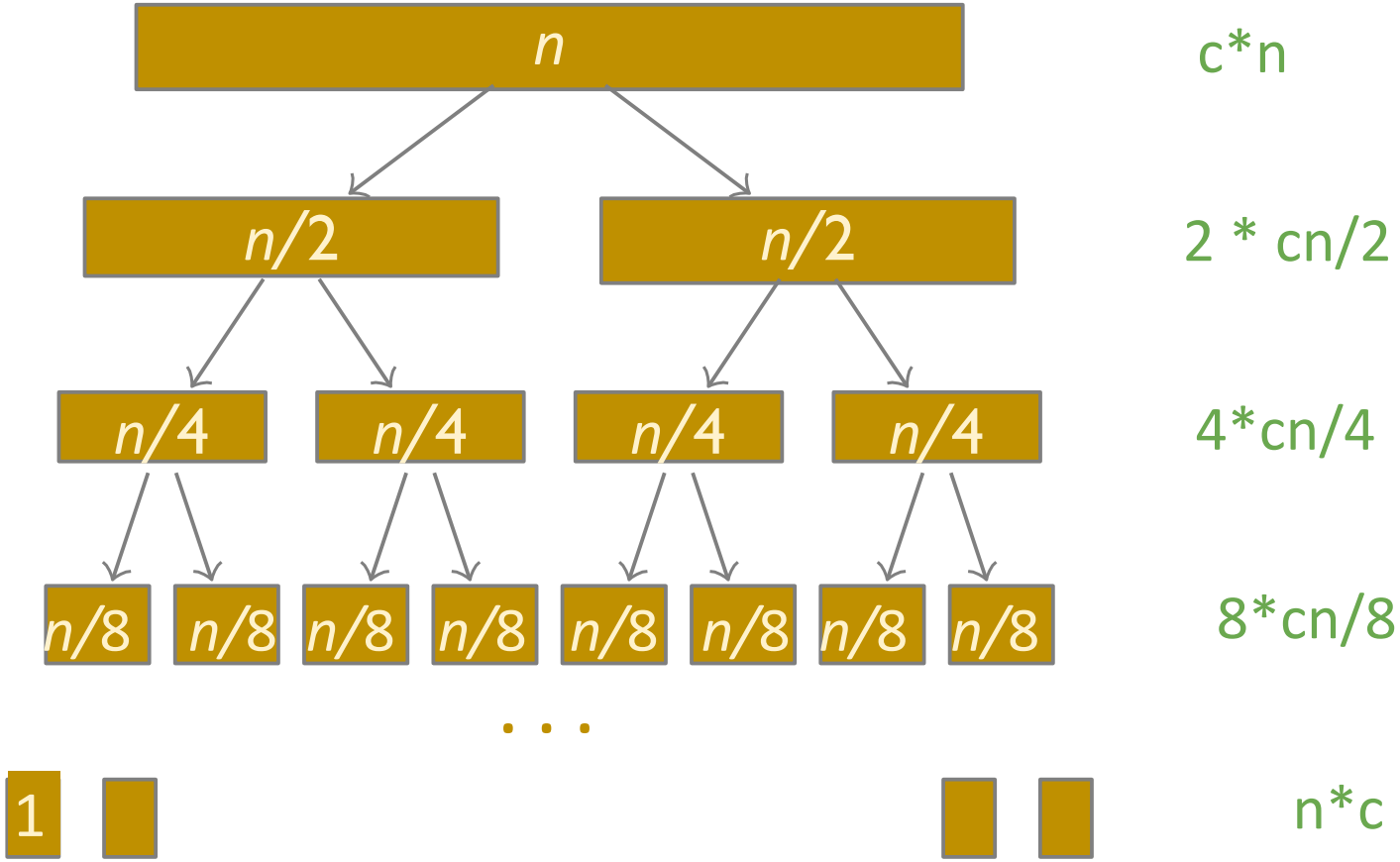
Merge sort: recursion tree



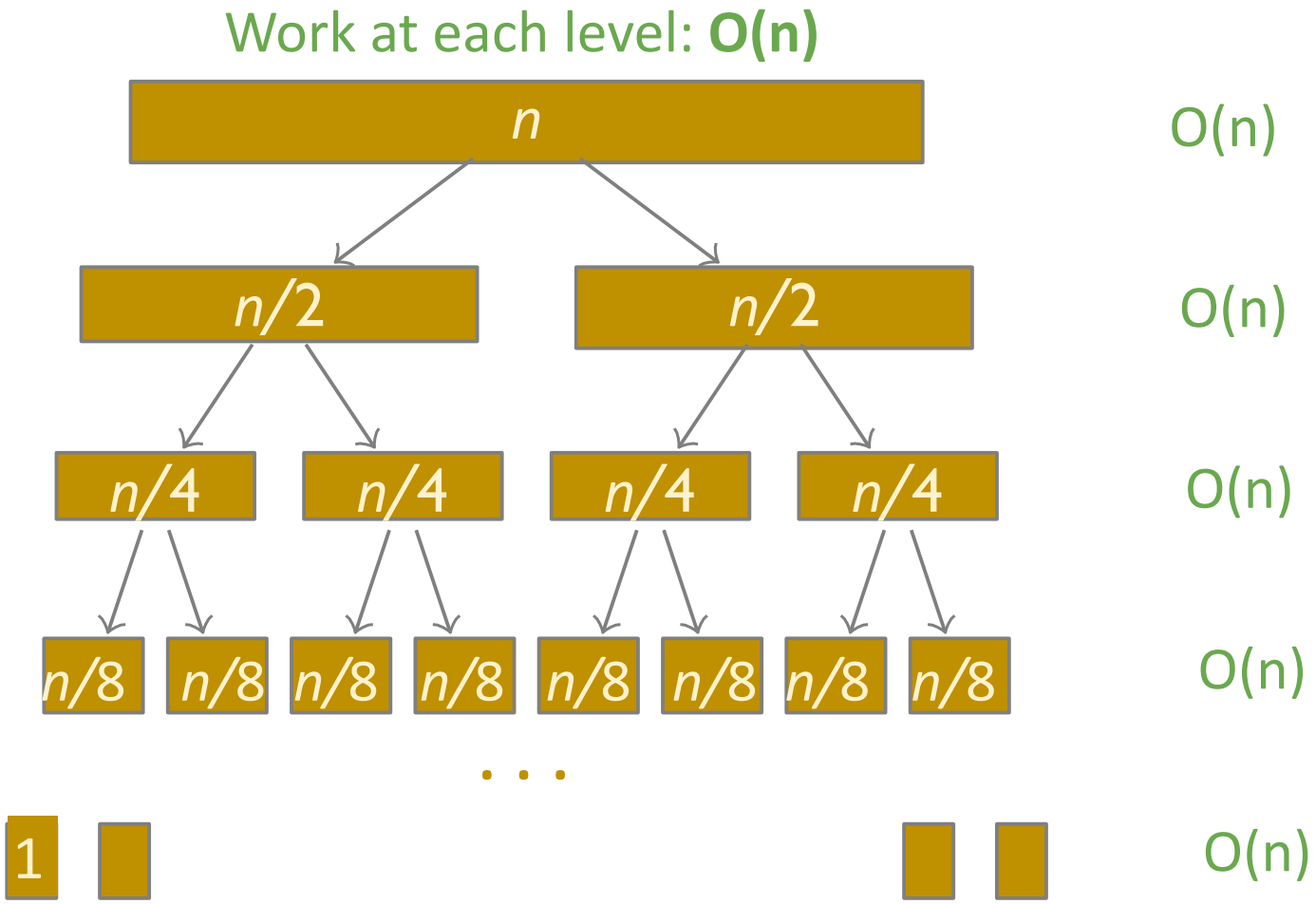
The height of tnis tree is *log n*

Merge sort: recursion tree

Work at each level: **all the work during *merge***



Merge sort: recursion tree



Total: O(*n*)*log *n* = O(*n* log *n*)

Sorting: Java way

Demo code: LINK

Sorting with `java.util.Collections`

- Java class ***`Collections`*** consists exclusively of static methods implementing various algorithms on *Collections*
- The ***`Collections.sort()`*** implements **merge sort**
- The method takes in any ***`Collection`*** and rearranges its elements in-place – the collection becomes sorted
- You encountered one of subclasses of *`Collection`*: `ArrayList` – which is just a dynamic array
- So we can say: **`Collections.sort(arrayList)`**

To be sorted elements must be Comparable

- To sort elements of any type we use generics
- ArrayList stores parametrized types:

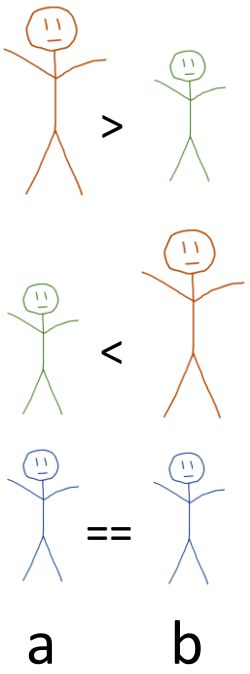
```
public class ArrayList<E>
List<Dog> dogs = new ArrayList<Dog> ();
```

- When we sort array of Strings, Dates or any primitive wrapper class of objects, then for these the order is already defined
- But if we want to sort custom objects – how should the algorithm compare them?

  Imagine you have an array of people. How would you put them in order? By height? By intelligence? By hotness?

We need a custom *Comparator*

- Merge sort algorithm compares pairs of values during the merge step, and pulls them into output according to this order
- We need to tell to the algorithm how two items should be compared
- We communicate this using one of three int values:



Positive number

Negative number

Zero

a b

Example: Sorting Dogs

Custom class of objects

```
public class Dog{
    String name;
    double age;
    int height;
    String owner;

    public Dog(String name, double age,
            int height, String owner) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.owner = owner;
    }
}

public static void main(String [] args) {
    List<Dog> dogs = new ArrayList<Dog> ();

    dogs.add(new Dog("Lisa", 2, 10, …));
    …

    Collections.sort(dogs);
}
```

✖

We cannot sort dogs, because it is not clear how two Dogs should be compared

Comparable interface

- Java provides **`Comparable`** interface which should be implemented by any custom class if we want to use sorting in *Arrays* or *Collections*
- The **`Comparable`** interface has parametrized **`compareTo(T obj)`** method which is used by the sorting algorithm to compare pairs of objects
- Our custom classes must implement this interface if we want to sort objects of a new type

Comparable Dogs

```
public class Dog implements Comparable<Dog>{
    String name;
    …

    public int compareTo(Dog another) {
        return this.name.compareTo((another).name);
    }
}
```

We declare Dog as *Comparable*<Dog>

Note that interface is also parametrized

Comparable interface declares a single method *compareTo* which returns a negative integer, zero, or a positive integer if "this" object is less than, equal to, or greater than another object passed as an argument.

We want to sort by *name*, which is *String*, and Strings already have *compareTo* method – so we reuse it here

We can sort now

```
public static void main(String [] args) {
    List<Dog> dogs = new ArrayList<Dog> ();
    dogs.add(…);…

    System.out.println("Before sorting:");
    printDogs(dogs);

    Collections.sort(dogs);
    System.out.println("After default sorting:");
    printDogs(dogs);
}
```

```
Before sorting:
Dog    Lisa  2.0 years   10 inches owned by   Alice
Dog    Bart  4.0 years   15 inches owned by     Bob
Dog   Marge  7.0 years   12 inches owned by   Alice
Dog    Lisa  3.0 years    8 inches owned by     Bob

After default sorting:
Dog    Bart  4.0 years   15 inches owned by     Bob
Dog    Lisa  2.0 years   10 inches owned by   Alice
Dog    Lisa  3.0 years    8 inches owned by     Bob
Dog   Marge  7.0 years   12 inches owned by   Alice
```

Flexible sorting

- In most real-life scenarios, we want to be able to sort based on different fields

  For example, we would like to be able to sometimes sort the employees based on salary, and another time sort them by last name or sort them by age – depending on the task

- The implementation of ***`Comparable.compareTo()`*** method enables only one default sorting and we can't change it dynamically
- To define multiple ways of sorting we can use Java ***`Comparator`*** interface and implement different comparators

Custom Dog Comparators: 1/3

- We can implement the **Height Comparator** in a **separate class**, and then pass it as a **second parameter** to the Collections.sort()

```
import java.util.Comparator;

public class HeightComparator
                    implements Comparator<Dog> {
    public int compare(Dog d1, Dog d2) {
        return d1.height - d2.height;
    }
}
```

That is implemented in a separate file

```
public static void main(String [] args) {
        …

    Collections.sort(dogs, new HeightComparator());
    System.out.println("After sorting by height:");
    printDogs(dogs);
}
```

```
After sorting by height:
Dog    Lisa  3.0 years    8 inches owned by     Bob
Dog    Lisa  2.0 years   10 inches owned by   Alice
Dog   Marge  7.0 years   12 inches owned by   Alice
Dog    Bart  4.0 years   15 inches owned by     Bob
```

Custom Dog Comparators: 2/3

- We can implement the **Age Comparator** inside the Dog class – as a **static method which returns a new Age Comparator**. Note that we only need to pass its name to Collections.sort()

```
public class Dog implements Comparable<Dog>{
    …

    public static Comparator<Dog> AgeComparator =
                                        new Comparator<Dog>() {
        public int compare(Dog d1, Dog d2) {
            return (int) (d1.age - d2.age);
        }
    };
}
```

This is part of the Dog class

```
public static void main(String [] args) {
    …
    Collections.sort(dogs, AgeComparator);
    System.out.println("After sorting by age:");
    printDogs(dogs);
}
```

```
After sorting by age:
Dog    Lisa  2.0 years   10 inches owned by   Alice
Dog    Lisa  3.0 years    8 inches owned by     Bob
Dog    Bart  4.0 years   15 inches owned by     Bob
Dog   Marge  7.0 years   12 inches owned by   Alice
```

Custom Dog Comparators: 3/3

- We can implement the **Owner Comparator <u>in place</u>** – directly inside the call to Collections.sort()

```
public static void main(String [] args) {
    …
    Collections.sort(dogs, new Comparator<Dog>() {
        public int compare(Dog d1, Dog d2) {
            return d1.owner.compareTo(d2.owner);
        }
    });

    System.out.println("After sorting by owner:");
    printDogs(dogs);
}
```

This is implemented directly as the second parameter to sort(). Note that this comparator does not have a name, so it cannot be reused in any other part of the program.

```
After sorting by owner:
Dog    Lisa  2.0 years   10 inches owned by   Alice
Dog   Marge  7.0 years   12 inches owned by   Alice
Dog    Lisa  3.0 years    8 inches owned by     Bob
Dog    Bart  4.0 years   15 inches owned by     Bob
```

```
public class Dog implements Comparable<Dog>{
    …
    public int compareTo(Dog another) {
        return this.height – another.height;
    }
}
```

height is integer

Which of the following will sort Dogs in reverse order of their height (from the tallest to the shortest)?

A
```
Collections.sort(dogs, new Comparator<Dog>() {
    public int compare(Dog d1, Dog d2) {
        return d2.height – d1.height;
    }
});
```

B
```
Collections.sort(dogs, new Comparator<Dog>() {
    public int compare(Dog d1, Dog d2) {
        return - d1.compareTo(d2);
    }
});
```

C
```
Collections.sort(dogs, new Comparator<Dog>() {
    public int compare(Dog d1, Dog d2) {
        return d2.compareTo(d1);
    }
});
```

- A
- B
- C
- All of the above
- None of the above

Java Merge Sort: notes

- The sorting in Java uses an optimized merge sort algorithm: the merge step is omitted if the highest element in the low sublist is less than the lowest element in the high sublist
- This algorithm offers guaranteed O(n log n) performance

- If we sort a LinkedList, this implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This is faster in practice than attempting to merge-sort a LinkedList directly (we need to add another O(n) at each level during partitioning phase)