Overloading in Java

```
int myMethod() { .. }

int myMethod(String s) { .. }

int myMethod(int[] arr) { .. }


Int x = myMethod();
Int y = myMethod("Dog");
Int arr[] = {1,2,3};
Int z = myMethod(arr);
```

Overloading in Java

```
void inOrderTraversal() { .. }

void inOrderTraversal(DLinkedList dll) {
    ..
    this.getLeftChild().inOrderTraversal(dll);
    ..
}

RootNode.inOrderTraversal();

DLinkedList dll = new DLinkedList();
RootNode.inOrderTraversal(dll);
```

Software Development 3 (F27SG)

Lecture 17

Efficient Sorting

Rob Stewart

Outline

- By the end of this lecture you should
  - know about the **Divide-and-Conquer** pattern
  - understand the *n-log-n* function, *O(n log n),* and analyse the difference with O(n)
  - understand the *O(n log n)* Divide-and-Conquer sorting algorithm called **Merge-Sort**
  - be able to compare the Insertion-Sort and Merge-Sort algorithms using Big-O
  - be familiar with the **Quick-Sort** algorithm

Analysis of Insertion Sort

- **Insertion-Sort** has 2 nested loops
  - worst case each loop iterates over every element
  - thus it is $O(n^2)$
- For ***search*** we saw that we could
  - **decrease** growth rate from O(n) to O(log n)
    - by **halving the problem in each step**
- Can a similar approach be applied to ***sorting***?

The Divide and Conquer Pattern

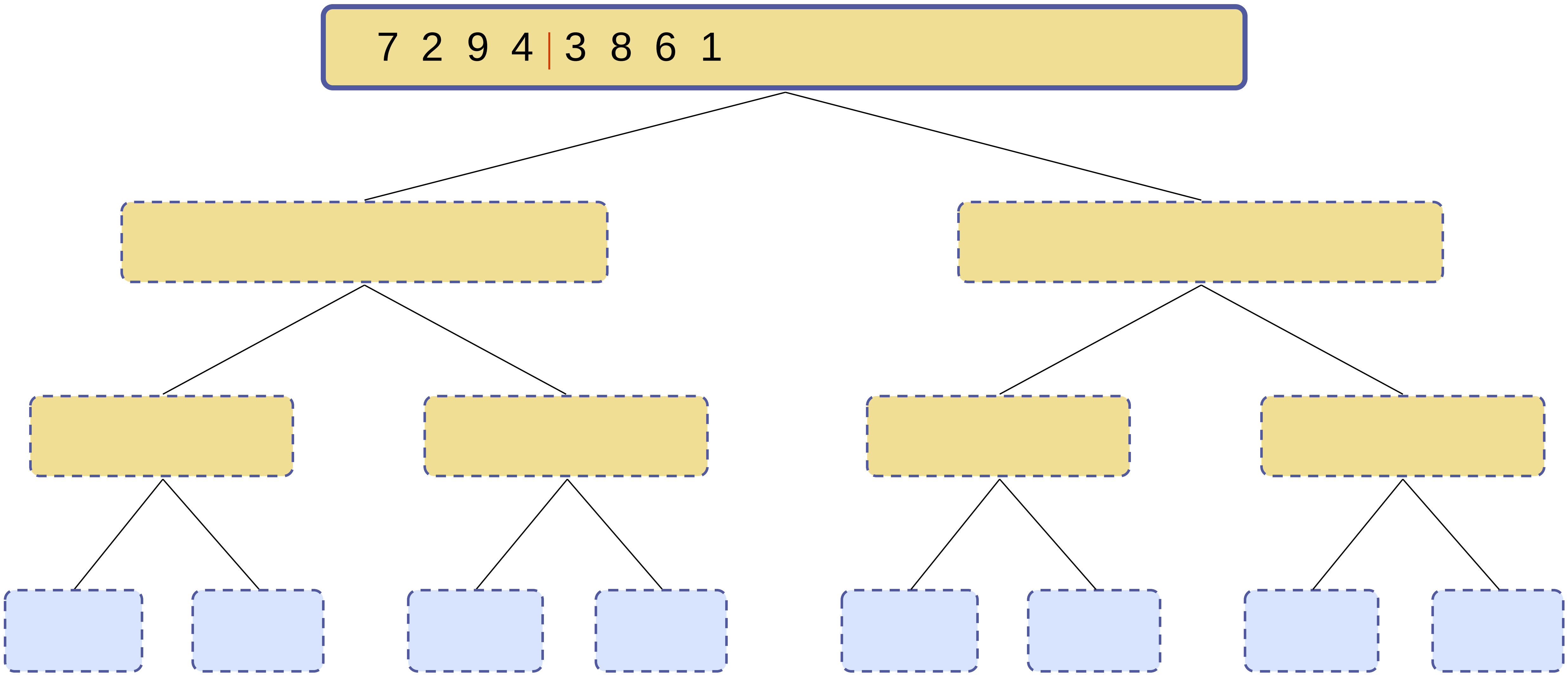
- The **Divide-and-Conquer** design pattern is recursively defined as follows:
  - **Base case:** If the size is smaller than a certain threshold then solve the problem in a straightforward manner
  - **Step case:** the step case consists of 3 steps:
    1. **Divide:** divide the data into 2 disjoint subsets
    2. **Recur:** solve the subproblems
    3. **Conquer:** combine the solutions of the subproblems into a solution of the main problem

Merge-Sort

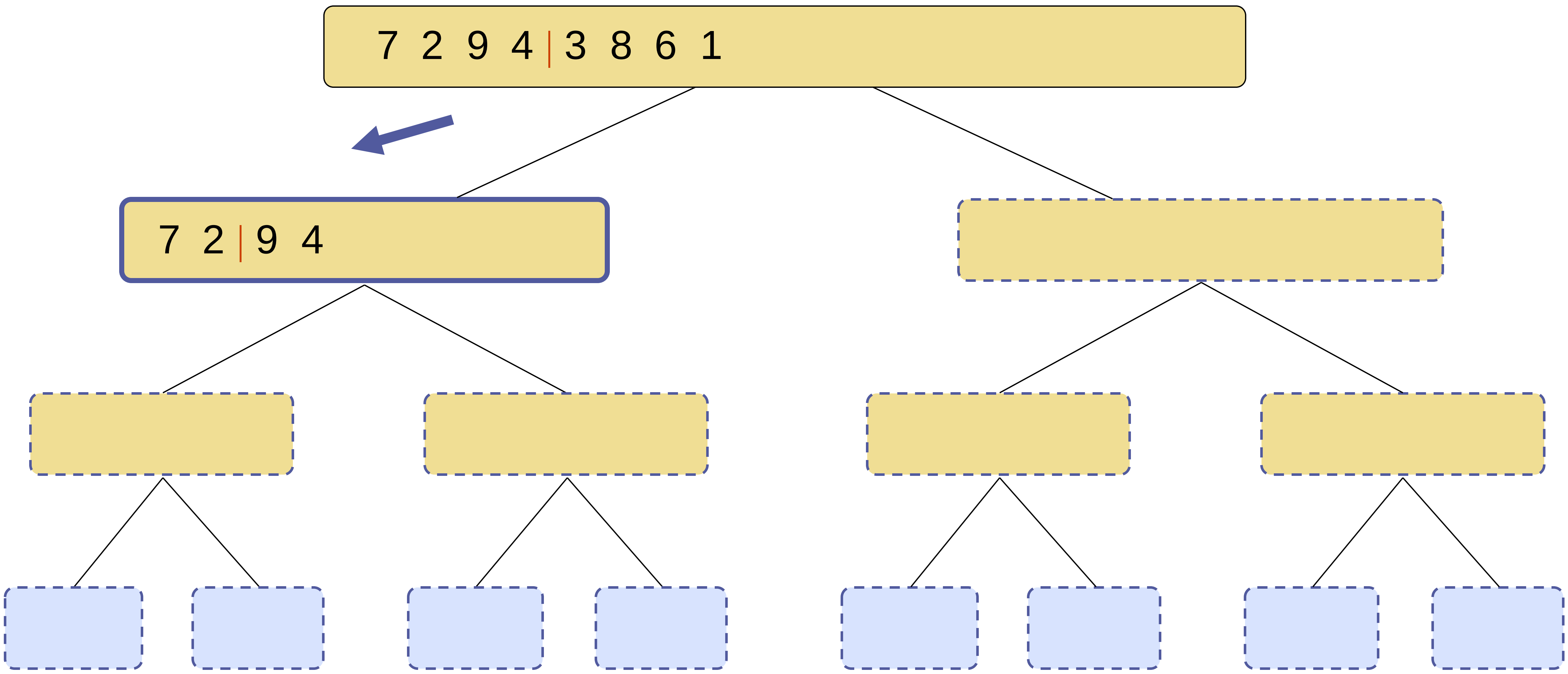
- **Merge-Sort** is a **Divide-and-Conquer** algorithm:
  - **Base case**: if size is less than 1 return
    - or use e.g. *Insertion-Sort* when the list is “small” (e.g. < 10)
  - **Step case:**
    1. split input list into two equal halves
    2. *recursively* sort left half, *recursively* sort right half
    3. merge two sorted halves into original list
       - while both lists have any items
         - » compare first element of left and right
         - » delete smallest and insert it into first free place of original list
       - one of the lists will still have elements left
         - » move them to the end of the original list

Merge-Sort Example (1)

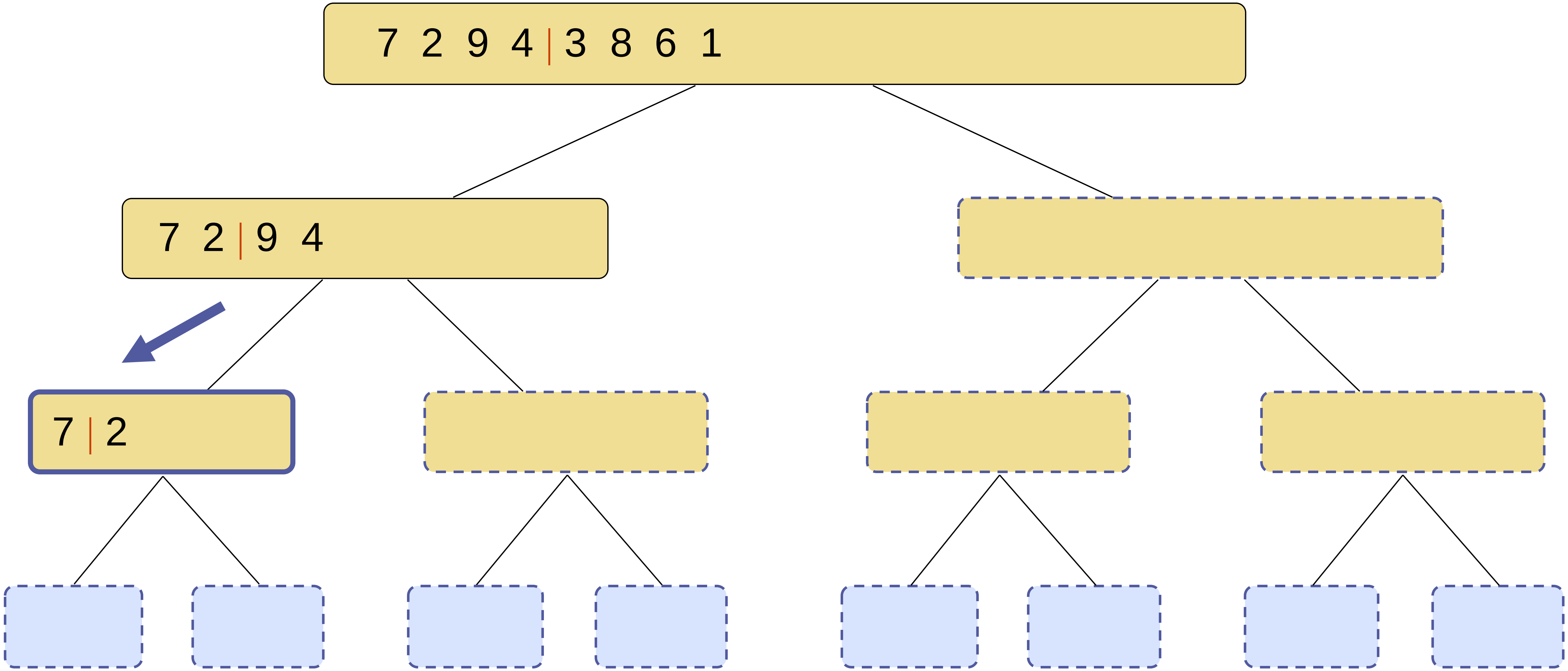
- Here, we will sort the list **[7,2,9,4,3,8,6,1]** :



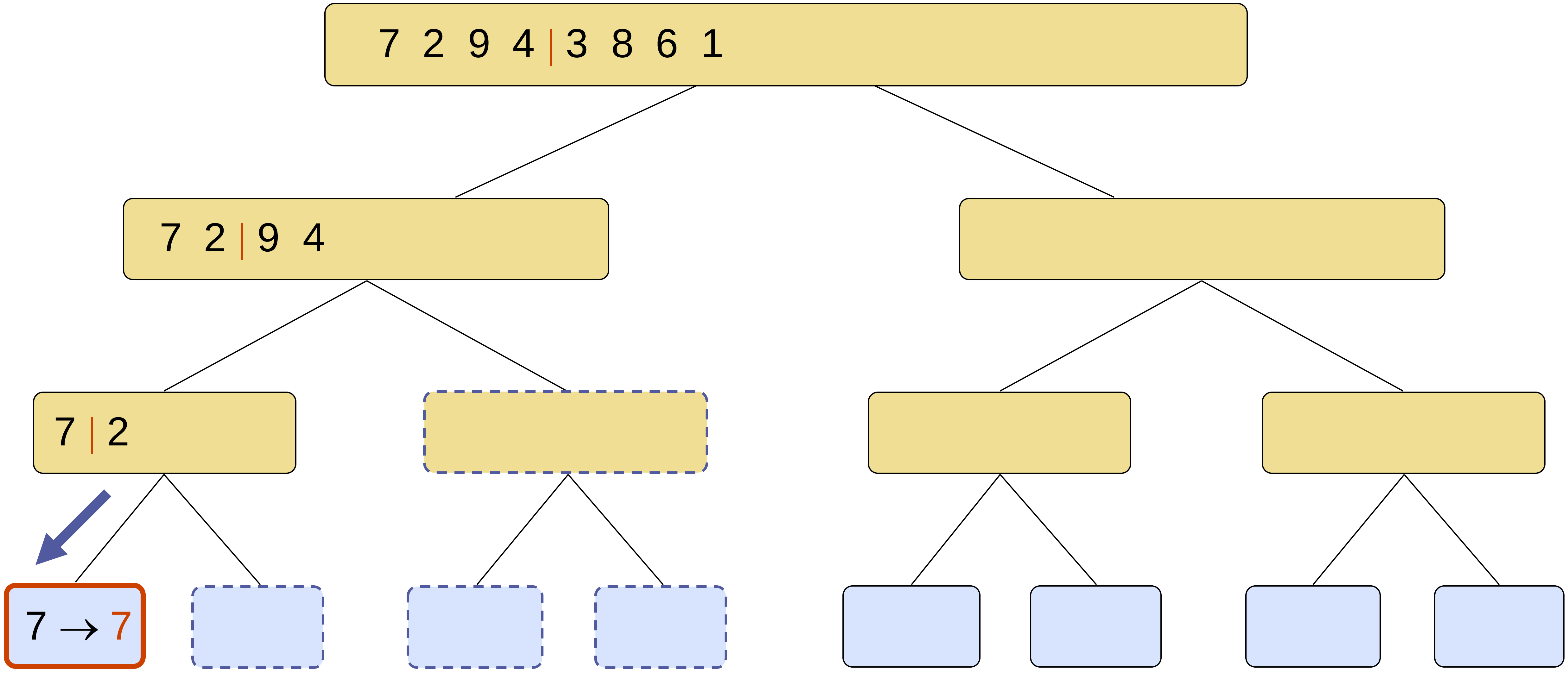
Merge-Sort Example (2)



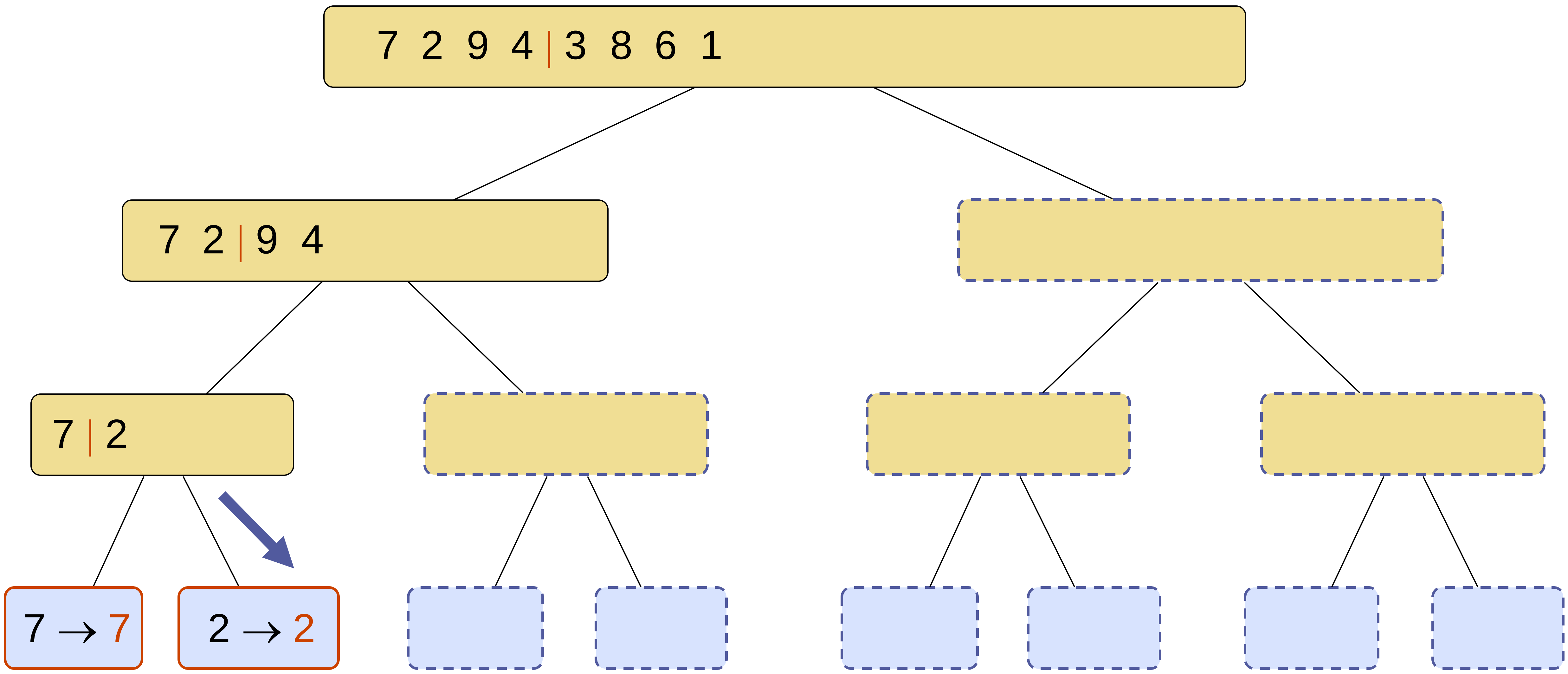
Merge-Sort Example (3)



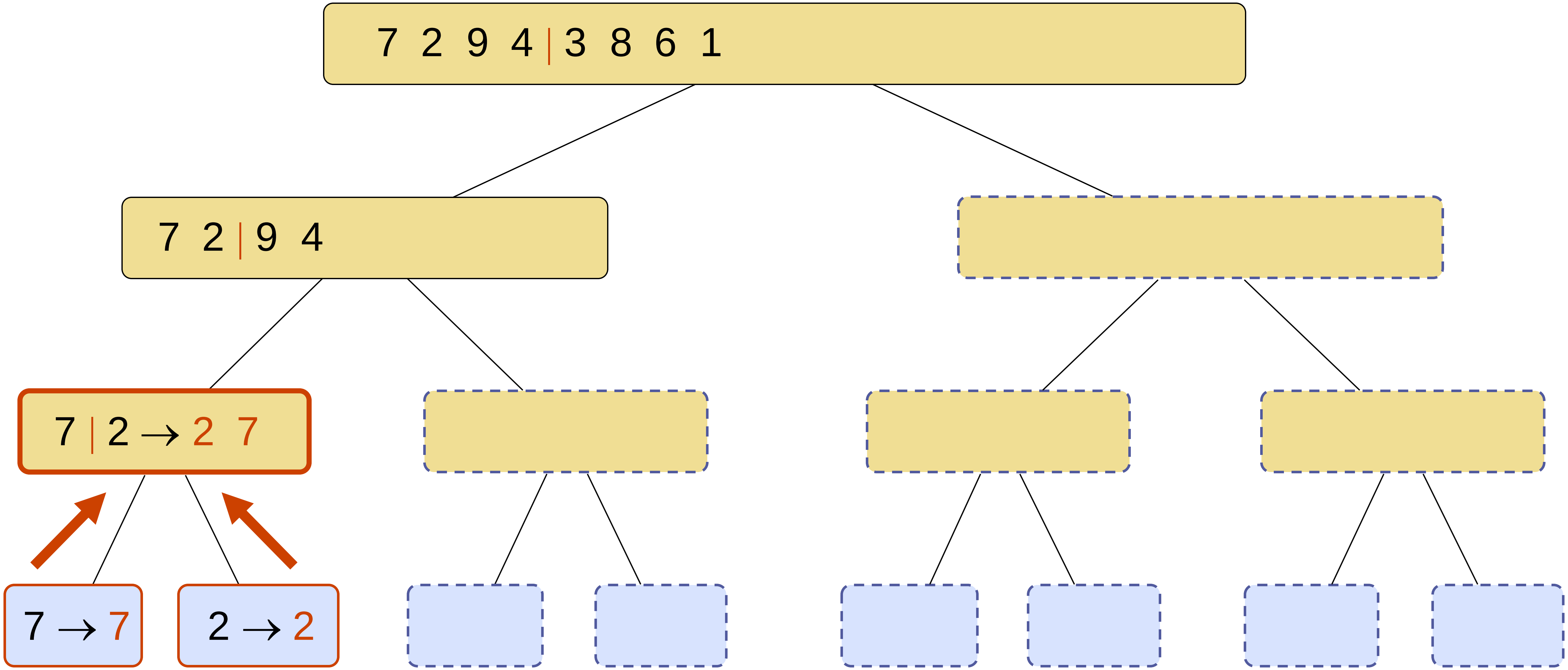
Merge-Sort Example (4)



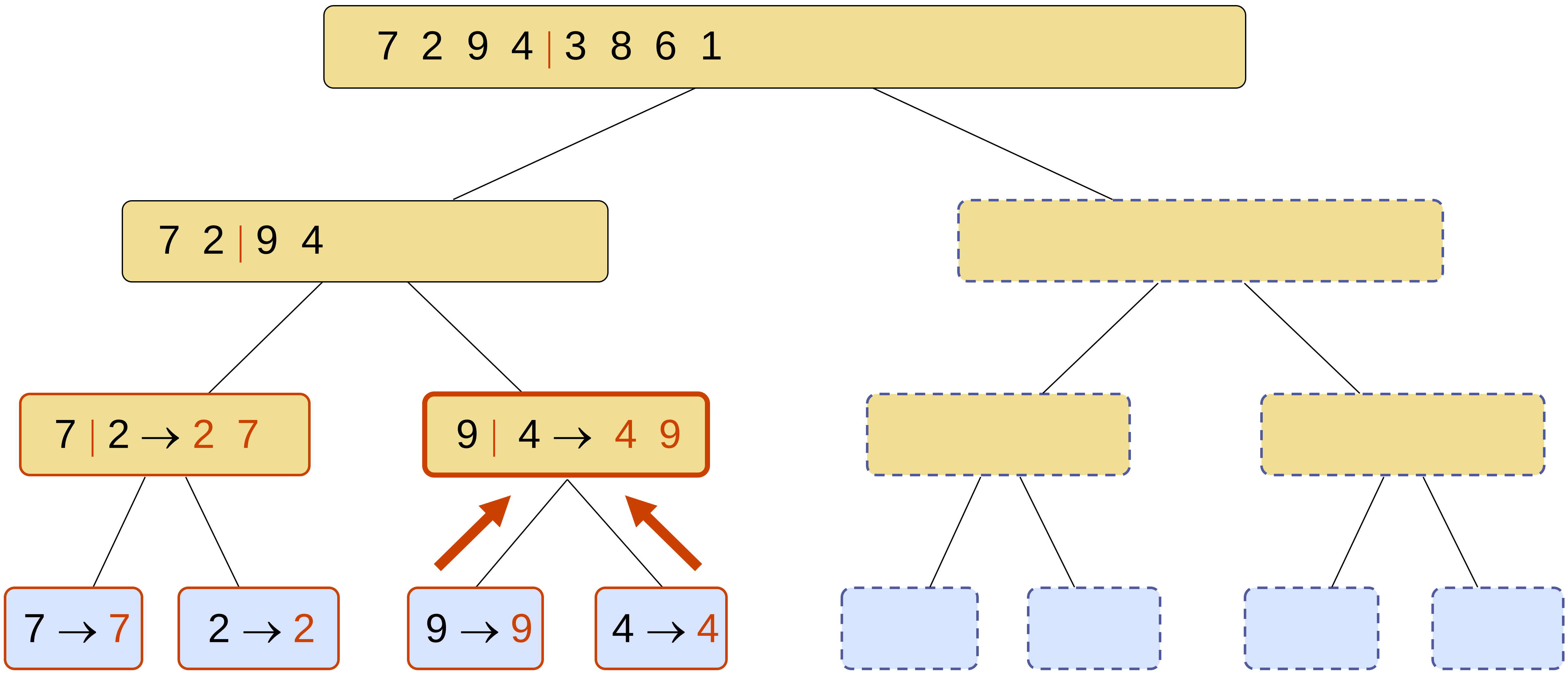
Merge-Sort Example (5)



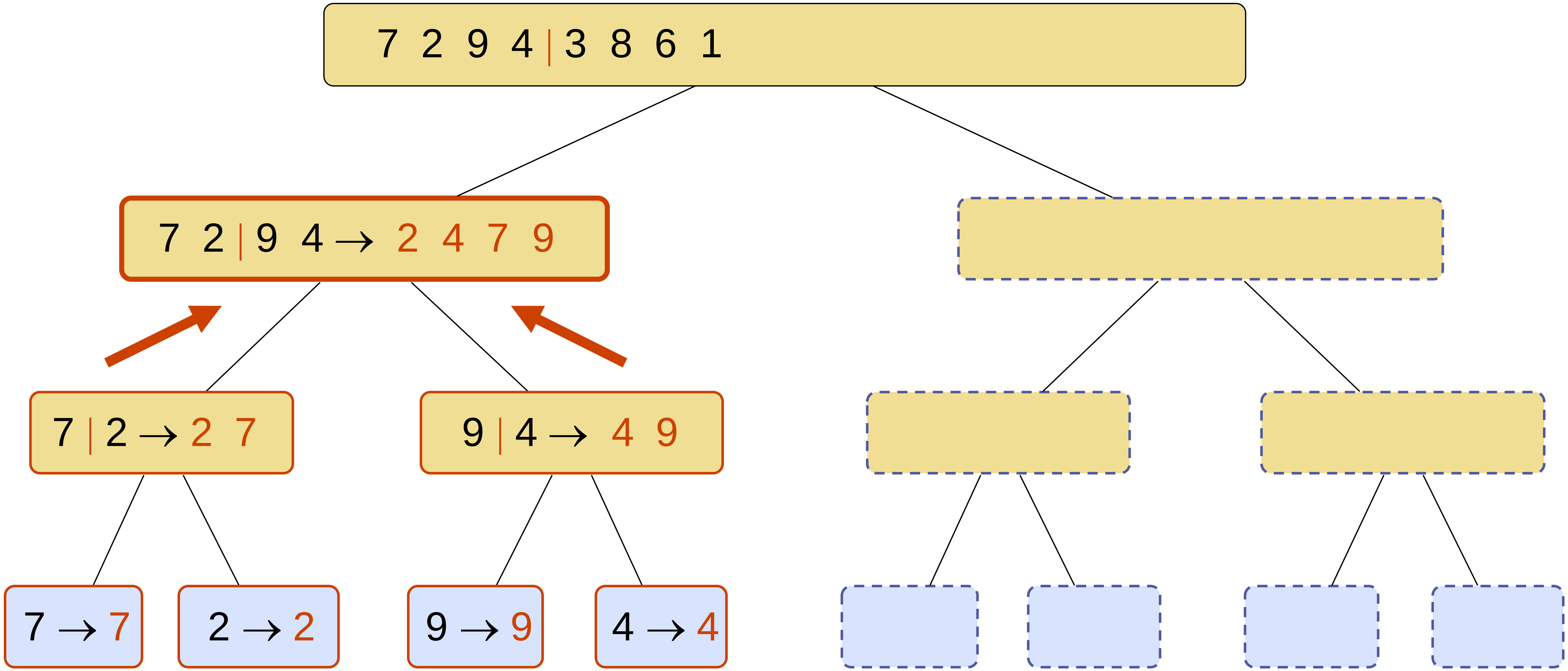
Merge-Sort Example (6)



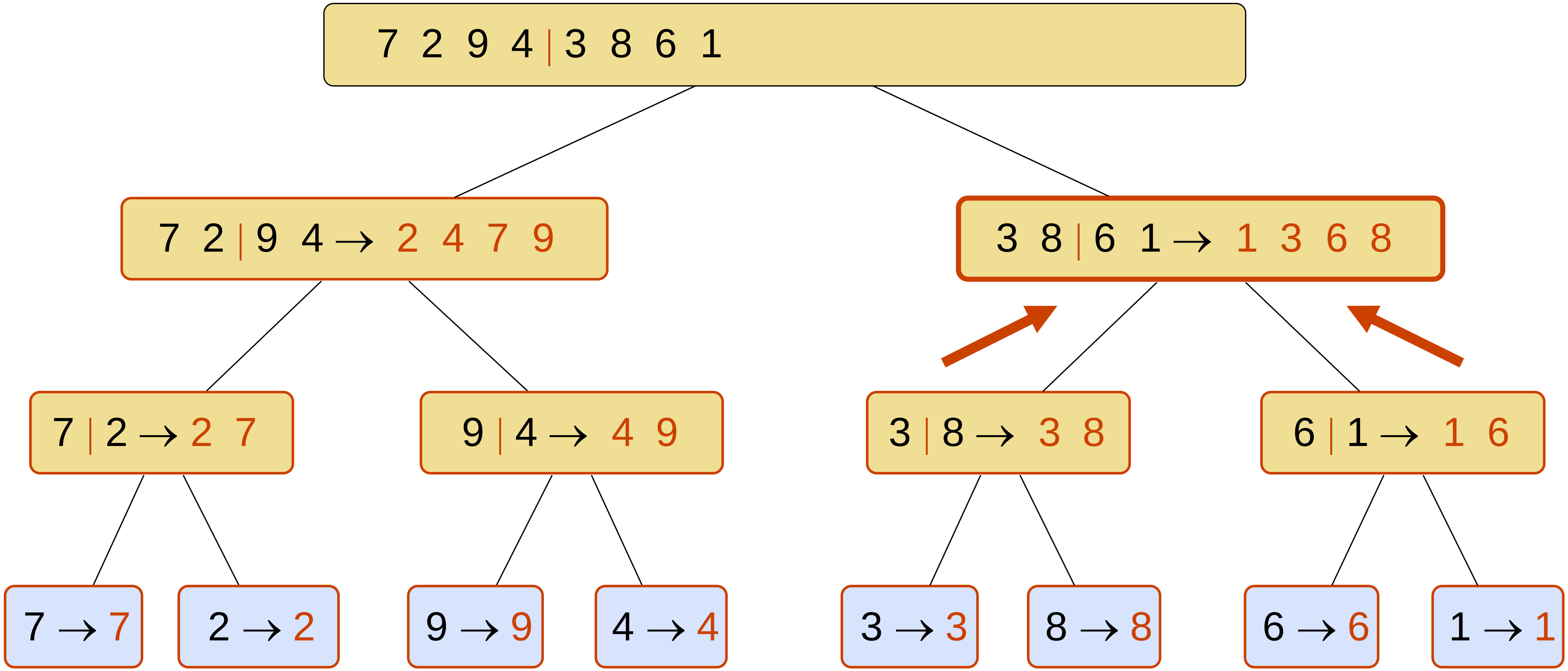
Merge-Sort Example (7)



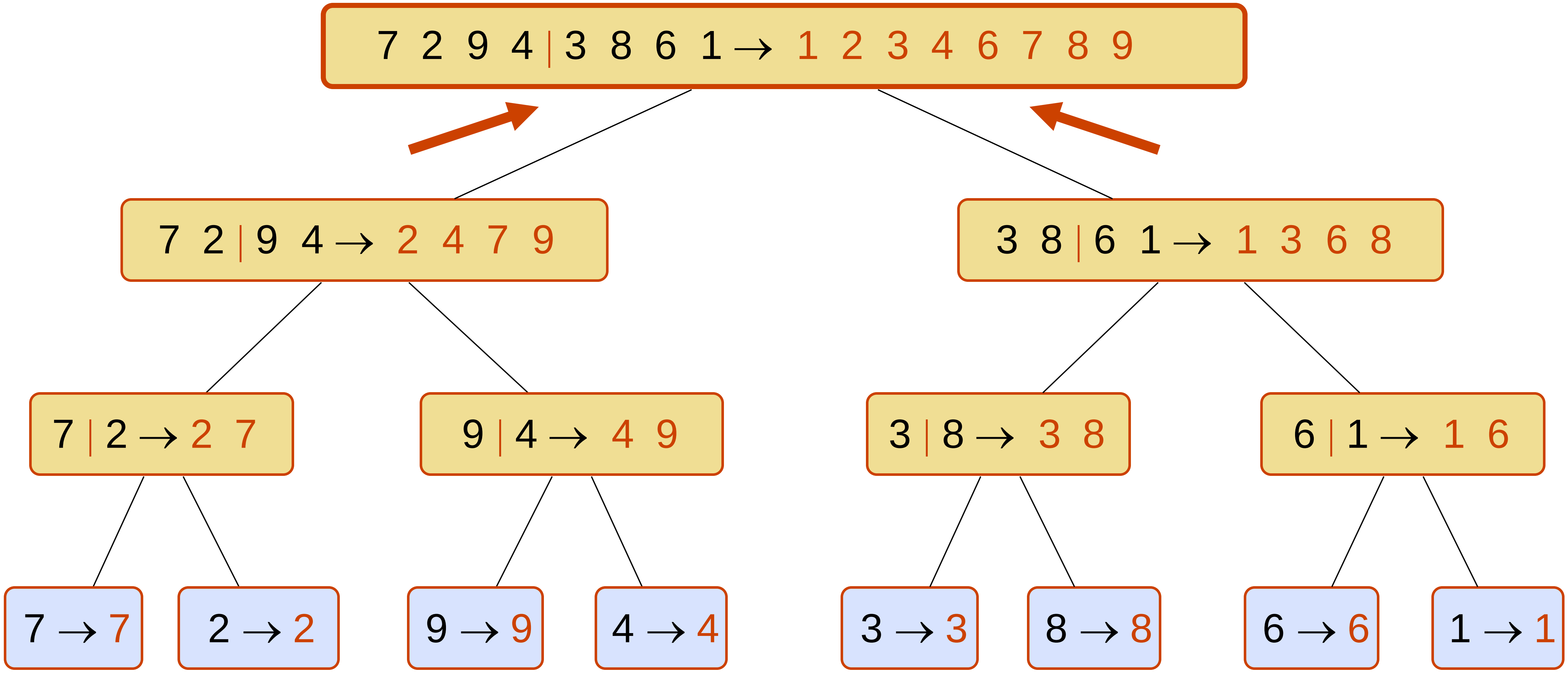
Merge-Sort Example (8)

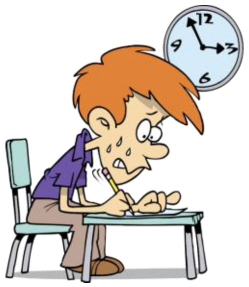


Merge-Sort Example (9)



Merge-Sort Example (10)





Exercise

Base case: if the size is less than 1 return

Step case:

1. split the input list into two equal halves
2. *recursively* sort the left half
   recursively sort the right half
3. merge the two sorted halves into the original list
   – while both lists have any items
      » compare the first element of left and right
      » delete the smallest and insert it into first free place of original list
   – one of the lists will still have elements left
      » move them to the end of the original list

Use **Merge-Sort** to sort the list {4,3,6,5,2,1}. Show

- the recursion trace,
- and each step of the merge operation.
- To get you started:

  - we start with mergeSort({4,3,6,5,2,1})
  - halfSize = 3, so two new lists are created
    - left = {4,3,6} and right = {5,2,1}
  - we then call mergeSort(left) and mergeSort(right) …

The Merge Sort

Eclipse demonstration

Merge-Sort in Java (1)

- We will now implement Merge-Sort for a list of integers.
- What are the arguments, return value, and base case(s) of the method?

```
public static void mergeSort(int[] list){
  int length = list.length;
  if (length < 2) // list of 1 or 0 elements is sorted
     return;
```

Merge-Sort in Java (2)

- In the step case we
  - divide the list into two halves
  - recursively sort the left side and the right side
  - then merge the result into the original list
- There are several ways we can implement this
  1. we can sort the list “in place” and create a new temp array we merge into, or
  2. we can create two temp arrays for left and right and use the actual array to merge
- We will follow the 2nd approach. First we create the arrays:

```
int halfSize = length/2;
int[] left = new int[halfSize];
int[] right = new int[length-halfSize];
```

Merge-Sort in Java (3)

- Then we copy the elements to the two new arrays
- We call this **partitioning** the list

```
int index = 0;
while(index < halfSize){
  left[index] = list[index];
  index++;
}
index = 0;
while(index < length-halfSize){
 right[index] = list[index+halfSize];
 index++;
}
```

Merge-Sort in Java (4)

- Finally we recursively sort each half
- … and merge the results

```
  mergeSort(left);
  mergeSort(right);
  merge(left,right,list);
}
```

The Merge Method (1)

- The merge method takes two sorted arrays
  - the left side, and
  - the right side
- ... and *merges* the into a third array (given as an argument):

```
public static void merge(int[] left,int[] right,int[] list){
  int index = 0; // index of list
  int lindex = 0; // index of left
  int rindex = 0; // index of right
```

The Merge Method (1)

Eclipse demonstration

The Merge Method (2)

- To merge the arrays we use the fact that both left and right is sorted
- We iterate through the array and compare the “current” element of each list
  - and pick the smallest
- When we reach the end of one array we stop

```
while(lindex < left.length && rindex < right.length){
  if (left[lindex] <= right[rindex]) // the “current” left is smallest
    list[index++] = left[lindex++];
  else // the “current” right is smallest
    list[index++] = right[rindex++];
}
```

The Merge Method (3)

- At the end
  - one of the arrays will still have elements left
  - these elements are then added to the end of the merged array

```
  while(lindex < left.length)
    list[index++] = left[lindex++];
  while(rindex < right.length)
    list[index++] = right[rindex++];
}
```

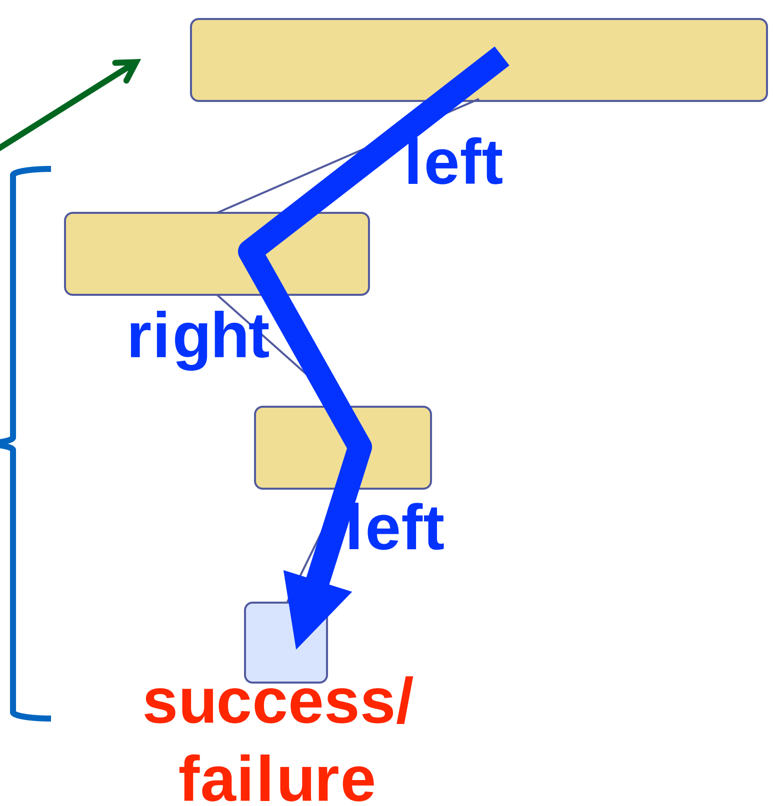
Analysis of MergeSort

- Complexity of MergeSort
  - Similar to complexity of Binary Search
- So lets revisit the complexity of Binary Search...

Analysis of Binary Search
(revisited)

- We can think of the complexity as
  - Complexity of current step
    + complexity of the recursive calls
  - For current step it is constant
  - Each step halves the size of array
  - Hence complexity **is the number of recursive calls**

```
public int binarySearch (int key, int first, int last)
  throws NotFoundException {
    if (first > last)
      throw new NotFoundException ("not found");
    else {
      int middle = (first + last) / 2;
      if (key == arr[middle])
         return arr[middle];
      else if (key < arr[middle])
         return binarySearch (key, first, middle-1);
       else
         return binarySearch (key, middle+1, last);
   }
}
```



Binary Search has log(N) Growth

- If we double input size, the **logarithm** increases by 1, e.g.
  - **log 8 = 3** (since 2^3 = 8)
  - **log 16 = 4** (since 2^4 = 16)

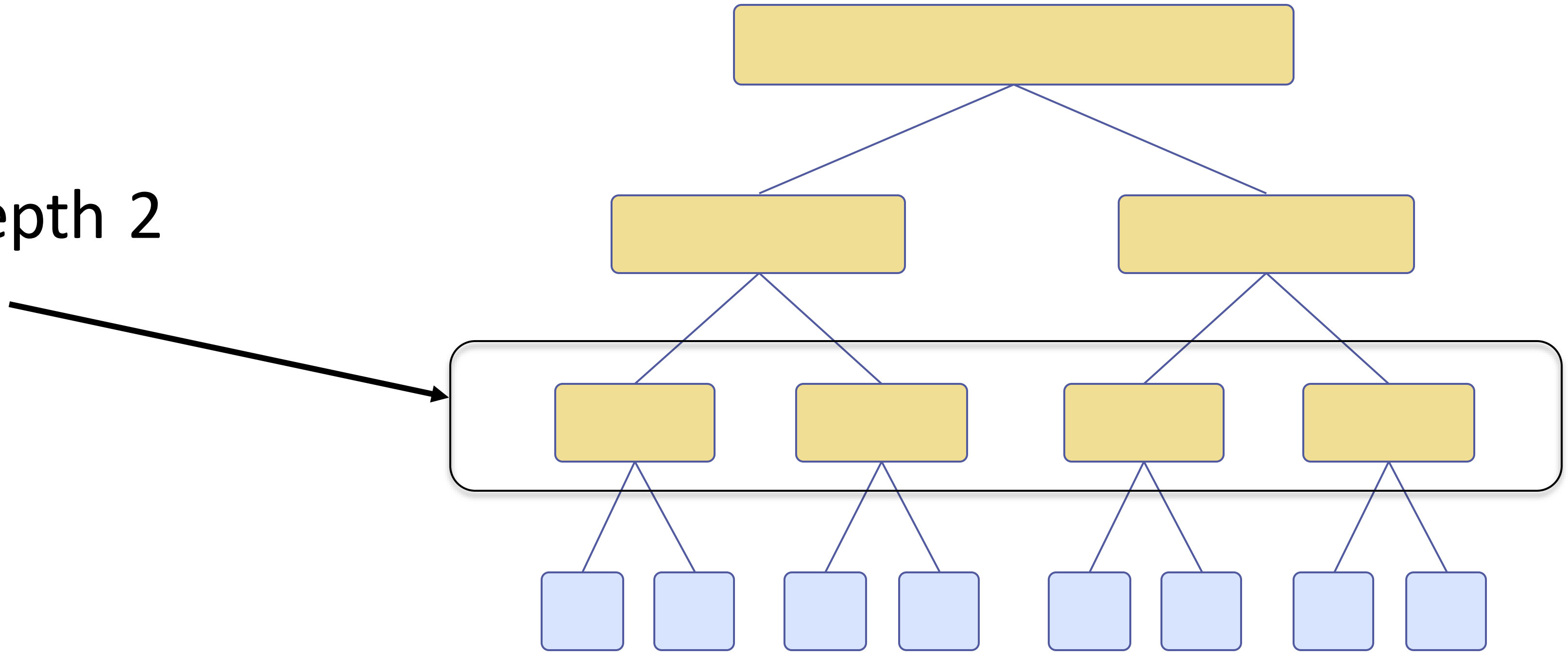
- Similar if we double size of array for **binary search**
  - we make one more recursive call
  - … ***list is halved for each call***
  - E.g. array of size 16
    - will result in a call to one of the half
    - … which is of size 8
  - Thus the **height of the tree is log(N)**
  - **Worst case: thing searching for is a leaf**

- Thus **binary search is worst case O(log N)**

Analysis of Merge-Sort (1)

- Next we observe some properties **for each *depth* in tree**
  - each node has two child nodes
  - list size of child node is half the size of parent's list
- As a consequence, each **depth *i***
  - has ***2^i* nodes**
  - the size of the list for each node is ***$N/2^i$***

- E.g. sorting 8 elements, at depth 2
- Nodes = 2^2 = 4
- Size of each list = 8/4 = 2



Analysis of Merge-Sort (2)

- Now analyse the time spent for each node at depth *i*
  - includes
    - *divide* (**partition**)
    - *conquer* (**merge**)
- **Partition** iterates list at a *linear time O(N)*
  - **while(index < halfSize){ ..}**
  - **while(index < length-halfSize){ .. }**
- **Merge** two sub lists also iterates list at a *linear time O(N)*
  - **while(lindex < left.length && rindex < right.length){ .. }**
- N + N = 2N, remove constant
  - Therefore, partition + merge = **O(N) (**N is size of array)
- Since ***size of list*** at this depth is $N/2^i$, time spent on a node is **$O(N/2^i)$**

Analysis of Merge-Sort (3)

- Total time spent:
  "*time spent at each depth*" <u>*times*</u> *"depth of tree"*

- **Time spent**
  - Time for each node is $\boldsymbol{O(N/2^i)}$
  - At each depth there are $\boldsymbol{2^i}$ nodes
  - at *depth* ***i*** total time is $\boldsymbol{O(2^i \cdot N/2^i)}$
    - $\boldsymbol{2^i \times N/2^i = 2^i/1 \times N/2^i = (2^i \times N)/(2^i \times 1) = (2^i \times N)/2^i = N}$
  - *Hence time spent at each depth is* ***O(N)***

- The **depth of the tree** is $\boldsymbol{O(\log N)}$
- Meaning $O(N) \cdot O(\log N) = \boldsymbol{O(N \log N)}$

The N-log-N function

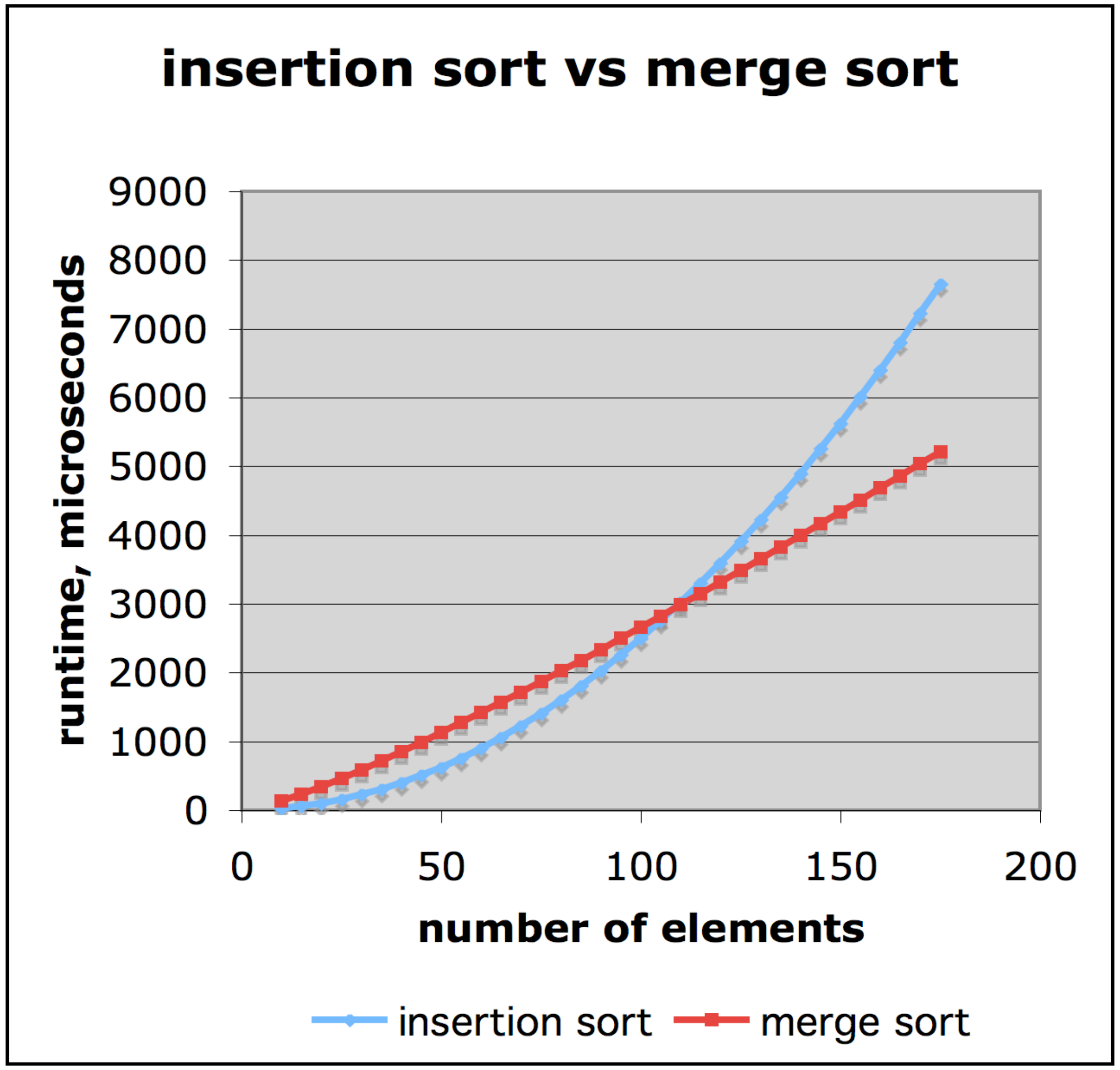
- The final function we will see is the *N-log-N* function:

  $$f(n) = n \log n$$

  which assigns to input n
  - the value of n times the (base-two) logarithm of n.
- The growth rate is
  - much better than the *quadratic* function
  - and slightly higher than the *linear* function

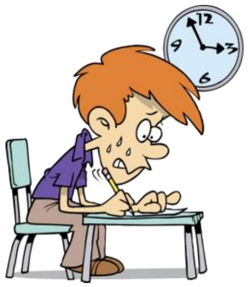
Insertion-Sort vs Merge-Sort

- Insertion-Sort
  - growth rate of $O(n^2)$
  - performs better than merge-sort when collection
    - is almost sorted, or
    - is small
- Merge-Sort
  - growth rate of $O(n \log n)$
  - outperforms Insertion-Sort on larger collections



Quick-Sort

- **Quick-Sort** is another *Divide-and-Conquer* algorithm
- We will work on this in the lab, and will only give a high-level description here for a list S
  1. **Divide**: select an element x from S which is called the **pivot** (often the last element of S).
     Divide S into 3 sub-lists:
     - L storing elements of S less than x
     - E storing elements of S equal to x
     - G storing elements of S larger than x
  2. **Recur:** Recursively sort L and G
  3. **Conquer:** Put back elements in the order of first elements of L, then elements of E, then elements of G.



Exercise

1. **Divide**: select an element x from S which is called the **pivot** (often the last element of S).
   Divide S into 3 sub-lists:
   - L storing elements of S less than x
   - E storing elements of S equal to x
   - G storing elements of S larger than x
2. **Recur:** Recursively sort L and G
3. **Conquer:** Put back elements in the order of first elements of L, then elements of E, then elements of G.

Use **Quick-Sort** to sort the list {4,3,6,5,2,1}

- You can choose the pivot yourself
- explain the values of L,E,G for each recursive call
- To get you started (assuming first element is the pivot):
  - we start with quickSort({4,3,6,5,2,1}) and pivot = 4
  - we get the lists L = {3,2,1}, E = {4} and G={6,5}.
  - we then recursively call quickSort(L) and quickSort(G) …

Summary

- You have learned about an efficient sorting algorithm called **Merge-Sort**
- You know *O(N log N)* and understand the time complexity of Merge-Sort
- You know how Merge-Sort and Insertion-Sort compares
  - and how to use Big-O to compare them
- You are familiar with **Quick-Sort**

- Visualisation of Bubble-Sort and Quick-Sort:
  - https://www.youtube.com/watch?v=vxENKlcs2Tw