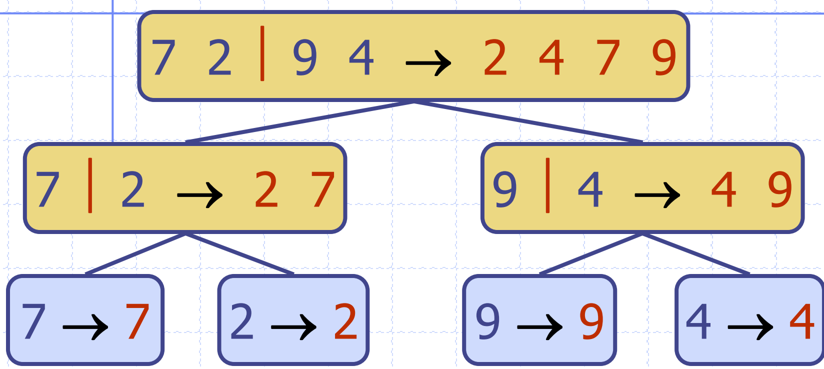
Lesson 4
Merge Sort: Collapsing Infinity To a Point



Wholeness of the Lesson

Merge Sort is a Divide and Conquer sorting algorithm which, by overcoming the limitations inherent in inversion-bound sorting algorithms, is able to sort lists in O(n log$n$) time, even in the worst case. The Divide and Conquer strategy is an example of the simple principle of “Do Less and Accomplish More.” This technique makes it possible to break the inversion-bound barrier for sorting alrgorithms, to obtain very fast running times.

The Divide and Conquer Algorithm Strategy

- **Divide** the problem into subproblems
- **Conquer** the subproblems by solving them recursively
- **Combine** the solutions to the subproblems into a solution to the problem

Merge-Sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Merge-sort on an input sequence S with n integers consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Conquer: recursively sort S_1 and S_2
  - Combine: merge S_1 and S_2 into a single sorted sequence

Algorithm ***mergeSort***(S)

  Input sequence S with n *integers*
  Output sequence S sorted

  if $S.size() > 1$ **then**
    $(S_1, S_2) \leftarrow$ *partition*$(S, n/2)$
    mergeSort(S_1)
    mergeSort(S_2)
    $S \leftarrow$ *merge*(S_1, S_2)
  return S

Merging Two Sorted Sequences

- The *combine* step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted arrays, each with $n/2$ elements takes $O(n)$ time

Algorithm *merge*(A, B)
  Input sorted sequences A and B with $n/2$ integers each
  Output sorted sequence S of $A \cup B$

  $S \leftarrow$ empty sequence
  while $\neg A.isEmpty() \wedge \neg B.isEmpty()$ **do**
    if $A.first() <= B.first()$ **then**
      $S.insertLast(A.remove(A.first()))$
    else
      $S.insertLast(B.remove(B.first()))$
  while $\neg A.isEmpty()$ **do**
    $S.insertLast(A.remove(A.first()))$
  while $\neg B.isEmpty()$ **do**
    $S.insertLast(B.remove(B.first()))$
  return S

Implementation of Merge

```
public void merge(int[] tempStorage,
                  int lowerPointer,
                  int upperPointer,
                  int upperBound) {
   int j = 0; //tempStorage index
   int lowerBound = lowerPointer;
   //total number of elements to rearrange
   int n = upperBound - lowerBound + 1;
   //view the range [lowerBound,upperBound] as two arrays
   //[lowerBound, mid], [mid+1,upperBound] to be merged
   int mid = upperPointer -1;
   while(lowerPointer <= mid && upperPointer <= upperBound){
      if(theArray[lowerPointer] <= theArray[upperPointer]){
          tempStorage[j++] = theArray[lowerPointer++];
      }
      else {
          tempStorage[j++] = theArray[upperPointer++];
      }
   }
```

Merge (continued)

```
    //left array may still have elements
    while(lowerPointer <= mid) {
        tempStorage[j++] = theArray[lowerPointer++];
    }
    //right array may still have elements
    while(upperPointer <= upperBound){
        tempStorage[j++] = theArray[upperPointer++];
    }
    //replace the range [lowerBound,upperBound] in theArray with
    //the range [0,n-1] just created in tempStorage
    for(j=0; j<n; ++j) {
        theArray[lowerBound+j] = tempStorage[j];
    }
}
```

Implementation of MergeSort

```
int[] theArray;

//public sorter
public int[] sort(int[] input){
    int n = input.length;
    int[] tempStorage = new int[n];
    theArray = input;
    mergeSort(tempStorage,0,n-1);
    return theArray;
}
```

(continued)

```
void mergeSort(int[] temp, int lower, int upper) {
        if(lower==upper){
            return;
        }
        else {
            int mid = (lower+upper)/2;
            mergeSort(temp,lower,mid);
            mergeSort(temp,mid+1, upper);
            merge(temp,lower,mid+1,upper);

        }
}
```

MergeSort Worked Example

```
Input array:  [4,3,2,1]

ms([0,0,0,0], 0, 3)
    lower = 0, upper = 3, mid = 1
    ms([0,0,0,0], 0, 1)
        lower =0, upper  = 1, mid = 0
        ms([0,0,0,0], 0, 0)
            return
        ms([0,0,0,0], 1, 1)
            return
        merge([0,0,0,0], 0, 1, 1)
            //theArray = [3, 4, 2, 1]
            //temp = [3,4, 0,0]
    ms([3,4,0,0], 2, 3)
        lower = 2, upper = 3, mid = 2
        ms([3,4,0,0], 2, 2)
            return
        ms([3,4,0,0], 3, 3)
            return
        merge([3,4,0,0], 2, 3, 3)
            //theArray = [3,4, 1,2]
            //temp = [1,2,0,0]
    merge([1,2,0,0], 0, 2, 3)
            //theArray = [1, 2, 3, 4]
            //temp = [1, 2, 3, 4]
```

theArray = [4, 3, 2, 1]
temp = [0, 0, 0, 0]

theArray [4, 3, 2, 1]
$\Rightarrow$ temp [4, 3, 0, 0]
$\Rightarrow$ temp [3, 4, 0, 0]
$\Rightarrow$ theArray [3, 4, 2, 1]

theArray [3, 4, 2, 1]
$\Rightarrow$ temp [2, 1, 0, 0]
$\Rightarrow$ temp [1, 2, 0, 0]
$\Rightarrow$ theArray [3, 4, 1, 2]

theArray [3, 4, 1, 2]
$\Rightarrow$ temp [3, 4, 1, 2]
$\Rightarrow$ temp [1, 2, 3, 4]
$\Rightarrow$ theArray [1, 2, 3, 4]

Worst-case Analysis

- The recurrence relation:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + an + b$$
$$T(1) = d$$

(Note: We could count primitive operations carefully to determine a, b, d)

Worst-Case Analysis: Master Formula

- We wish to apply the Master Formula to the recurrence:

  $$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + an + b$$
  $$T(1) = d$$

  but it is not quite in the right form :

  - arguments to T should be either both ceilings or both floors
  - the last term should be of the form cn^k, not $an + b$.

We note that for some c (actuallly, $c = a + 1$), we have $an + b < cn$ (when n is large enough). So we have

$$T(n) \leq 2T(\lceil n/2 \rceil) + cn$$

The next lemma shows we can use the Master Formula for this kind of formula too.

Master Formula (continued)

Lemma. Suppose $T(1) = d$, $T(n) \le 2T(\lceil n/2 \rceil) + cn$. Define a recurrence $S(1) = d$, $S(n) = 2S(\lceil n/2 \rceil) + cn$. Then for all n,

$$T(n) \le S(n)$$

In particular, if it can be shown that $S(n)$ is $\Theta(g(n))$ (using the Master Formula, for example), then $T(n)$ is $O(g(n))$

Proof. Proceed by induction on n to show $T(n) \le S(n)$. This is obvious for $n = 1$. Assume $T(k) \le S(k)$ whenever $k < n$. Then

$$\begin{aligned} T(n) &\le 2T(\lceil n/2 \rceil) + cn \\ &\le 2S(\lceil n/2 \rceil) + cn \\ &= S(n) \end{aligned}$$

Master Formula (continued)

- Now we can use Master Formula on

  $T(1) = d, T(n) \le 2T(\lceil n/2 \rceil) + cn$

  in the following way. Define a recurrence relation

  $S(1) = d, S(n) = 2S(\lceil n/2 \rceil) + cn$

- Here we have:

  $a = 2, b = 2, c = c, d = d, k = 1$ and $a = b^k$

- The Master Formula then tells us:

  $S(n) = \Theta(n^k \log n) = \Theta(n \log n)$

  Now (by the Lemma), since $T(n) \le S(n)$, it follows that

  $T(n)$ is in $O(n \log n)$

The Master Formula

For recurrences that arise from Divide-and-Conquer algorithms (like Binary Search), there is a general formula for finding a closed-form solution:

Theorem. Suppose $T(n)$ satifies

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT\left(\lceil \frac{n}{b} \rceil\right) + cn^k & \text{otherwise} \end{cases}$$

where k is a non-negative integer and a, b, c, d are constants with $a > 0, b > 1, c > 0, d \geq 0$. Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta\left(n^{\log_b a}\right) & \text{if } a > b^k \end{cases}$$

Generalized Master Formula

Sometimes in a recurrence relation, $T(n)$ is found to be $\leq$ some recurrence expression, rather than equal to it. The following variation of the Master Formula can be used in such cases.

Theorem. Suppose $T(n)$ satifies

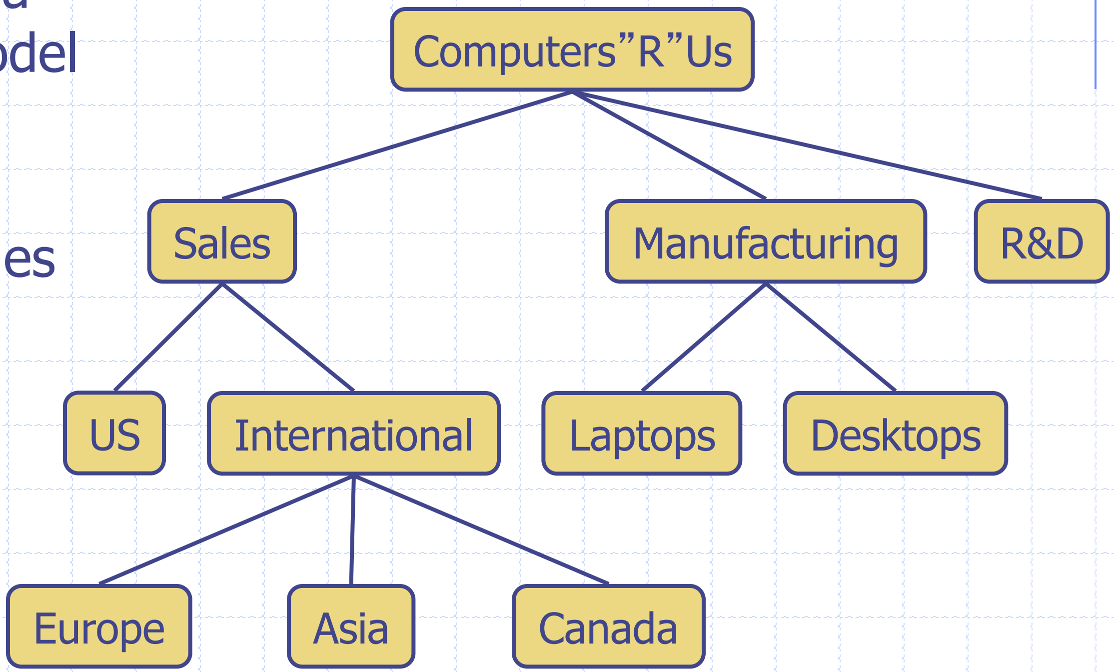
$$T(1) = d; \qquad T(n) \leq aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + cn^k \qquad \text{for } n > 1,$$

where k is a non-negative integer and a, b, c, d are constants with $a > 0, b > 1, c > 0, d \geq 0$. Then

$$T(n) = \begin{cases} \mathrm{O}(n^k) & \text{if } a < b^k \\ \mathrm{O}(n^k \log n) & \text{if } a = b^k \\ \mathrm{O}\left(n^{\log_b a}\right) & \text{if } a > b^k \end{cases}$$

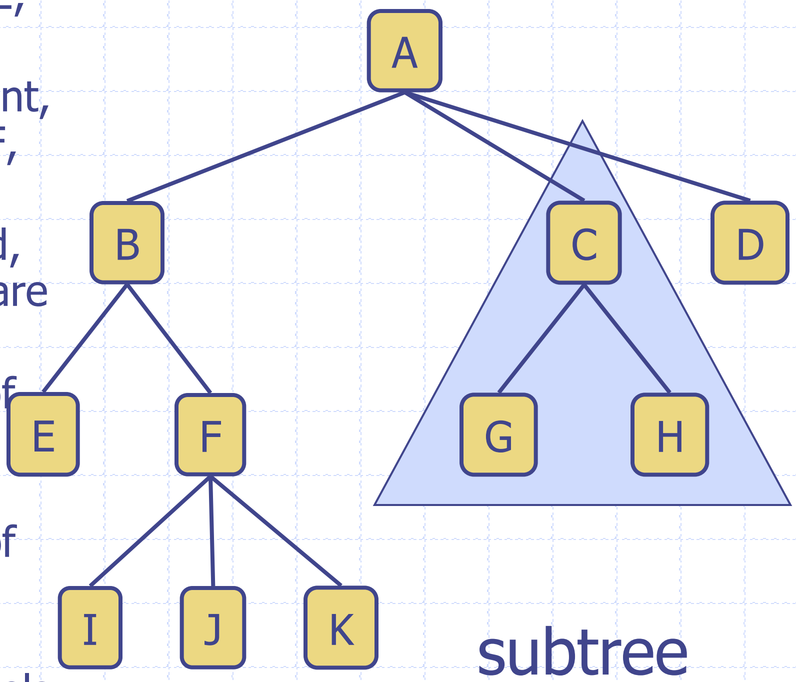
Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



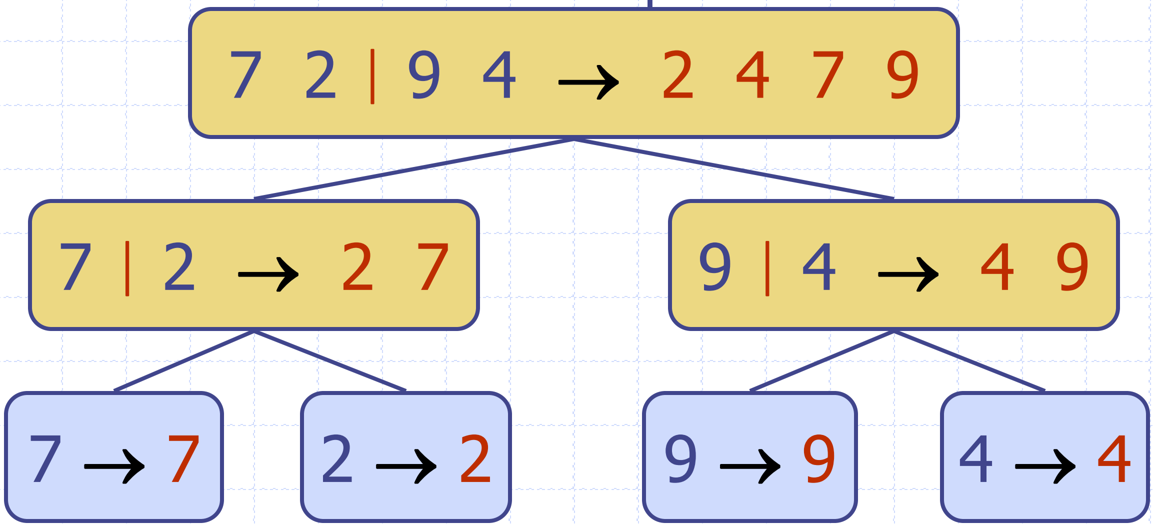
Tree Terminology

- ***Root***: node without parent (A)
- ***Internal node***: node with at least one child (A, B, C, F)
- ***Leaf*** : ***node*** is a node without children (E, I, J, K, G, H, D)
- ***Ancestors of a node***: parent, grandparent, grand-grandparent, etc. (ancestors of K: F, B, A)
- ***Descendant of a node***: child, grandchild, grand-grandchild, etc. (descendants of B are E, F, I, J, K)
- ***Depth of a node***: number of ancestors of the node (depth of K = 3)
- ***Levels of a tree***: Level n of a tree is the set of all nodes having depth n. (Level 1 of this tree is {B, C, D})
- ***Height of a tree***: maximum depth of all nodes (height of tree = 3). Note: Num levels = height + 1.
- ***Subtree***: tree consisting of a node and its descendants



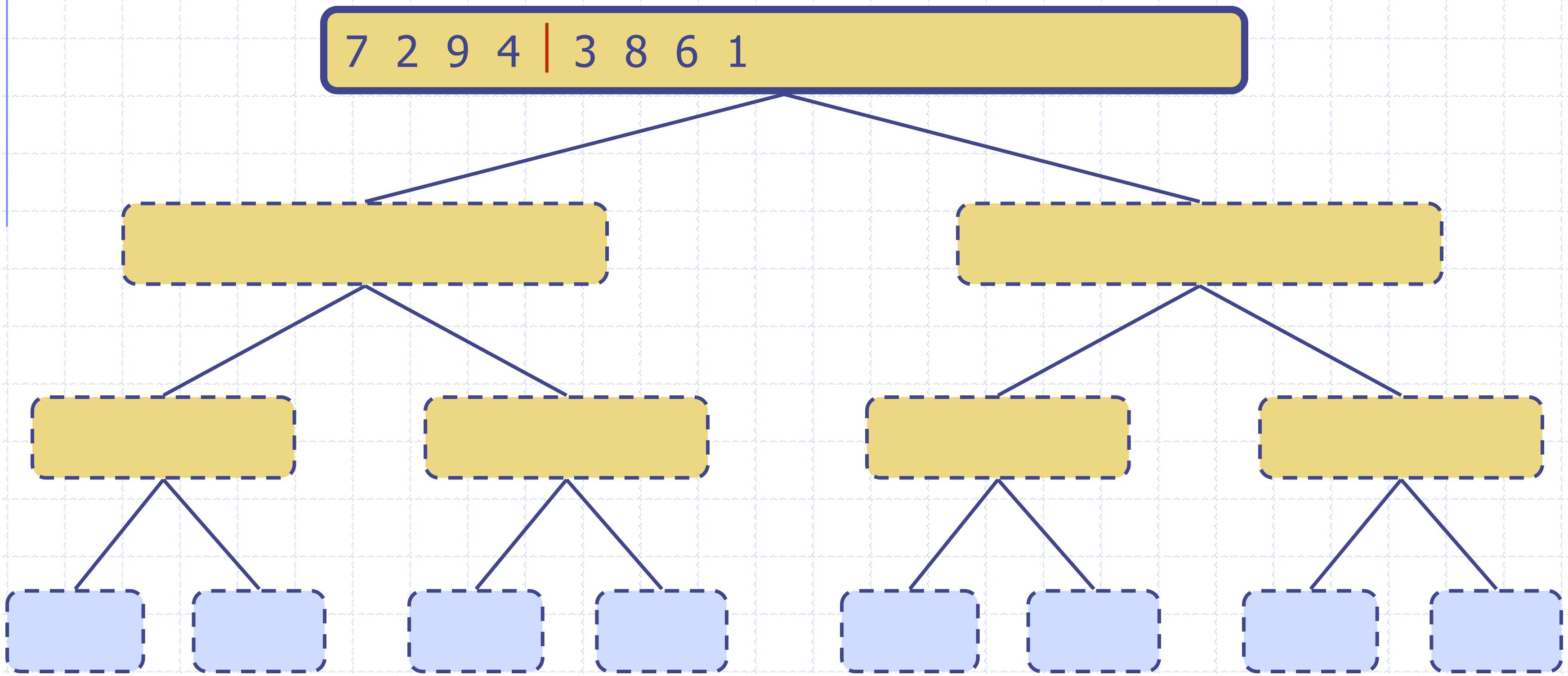
Merge-Sort Tree

- An execution of merge-sort may be depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution
    - its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



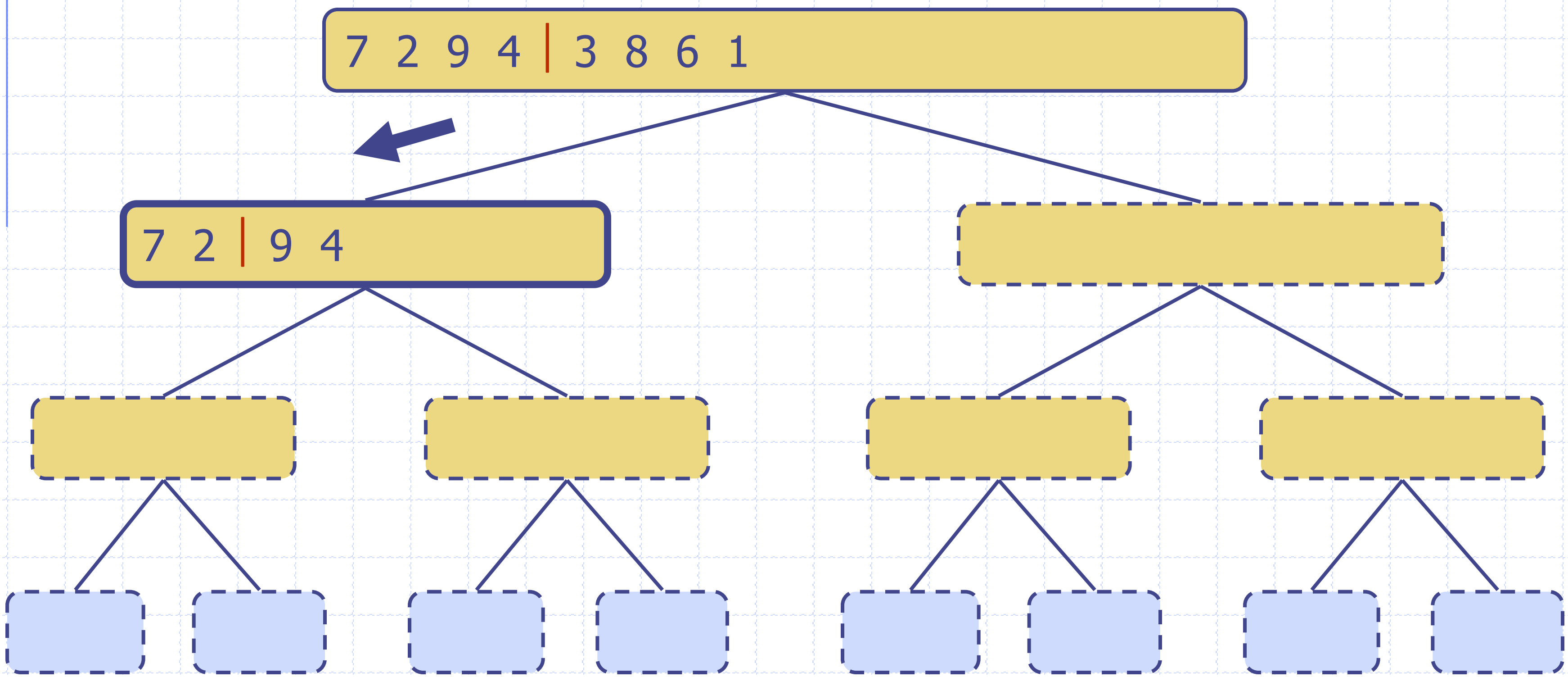
Execution Example

- Partition



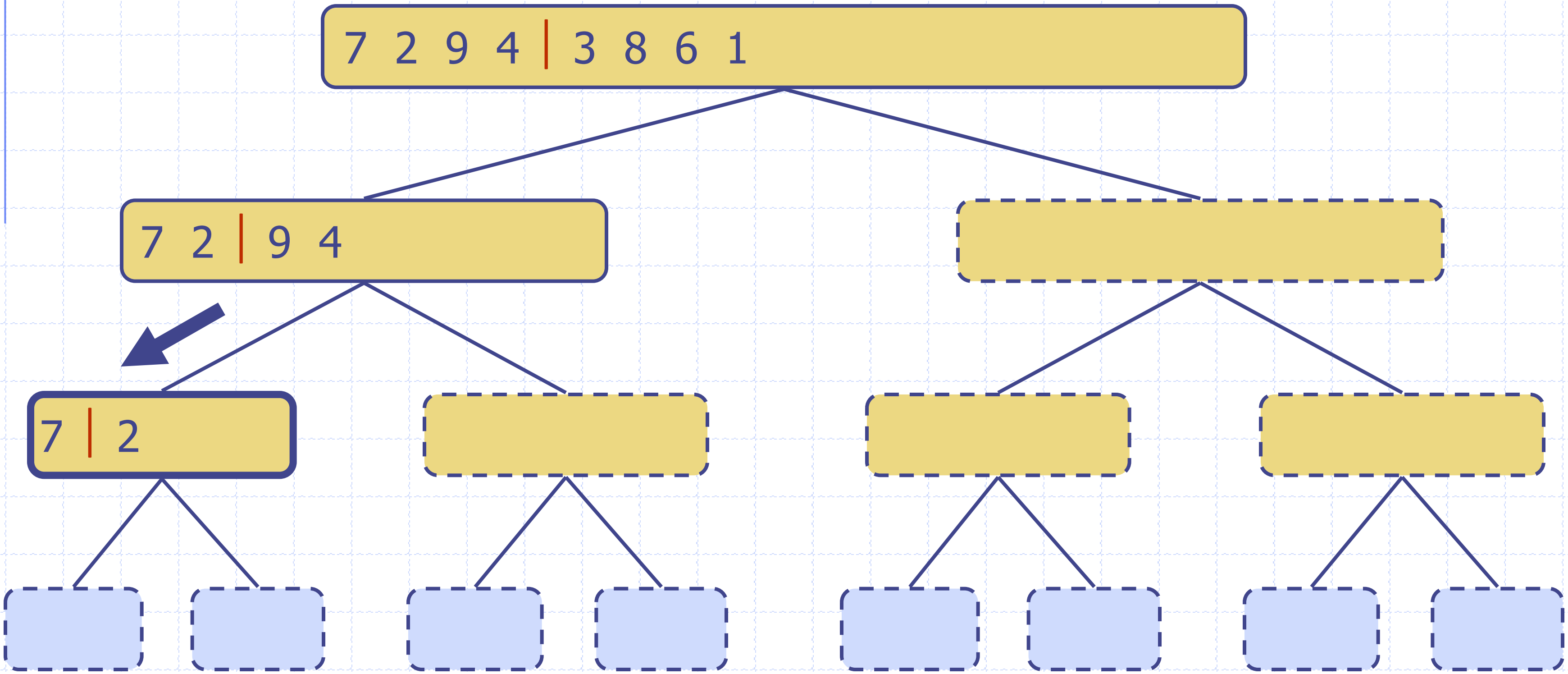
Execution Example (cont.)

- Recursive call, partition



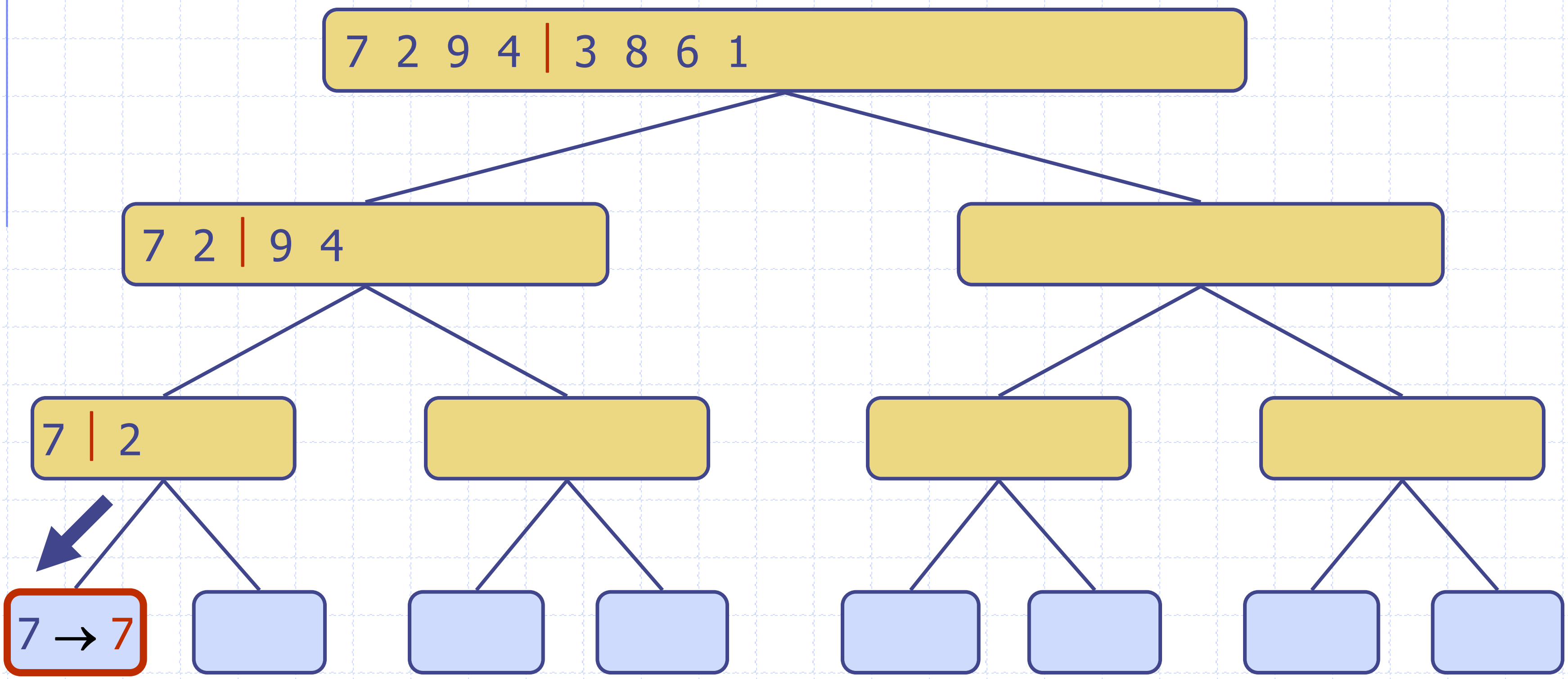
Execution Example (cont.)

- Recursive call, partition



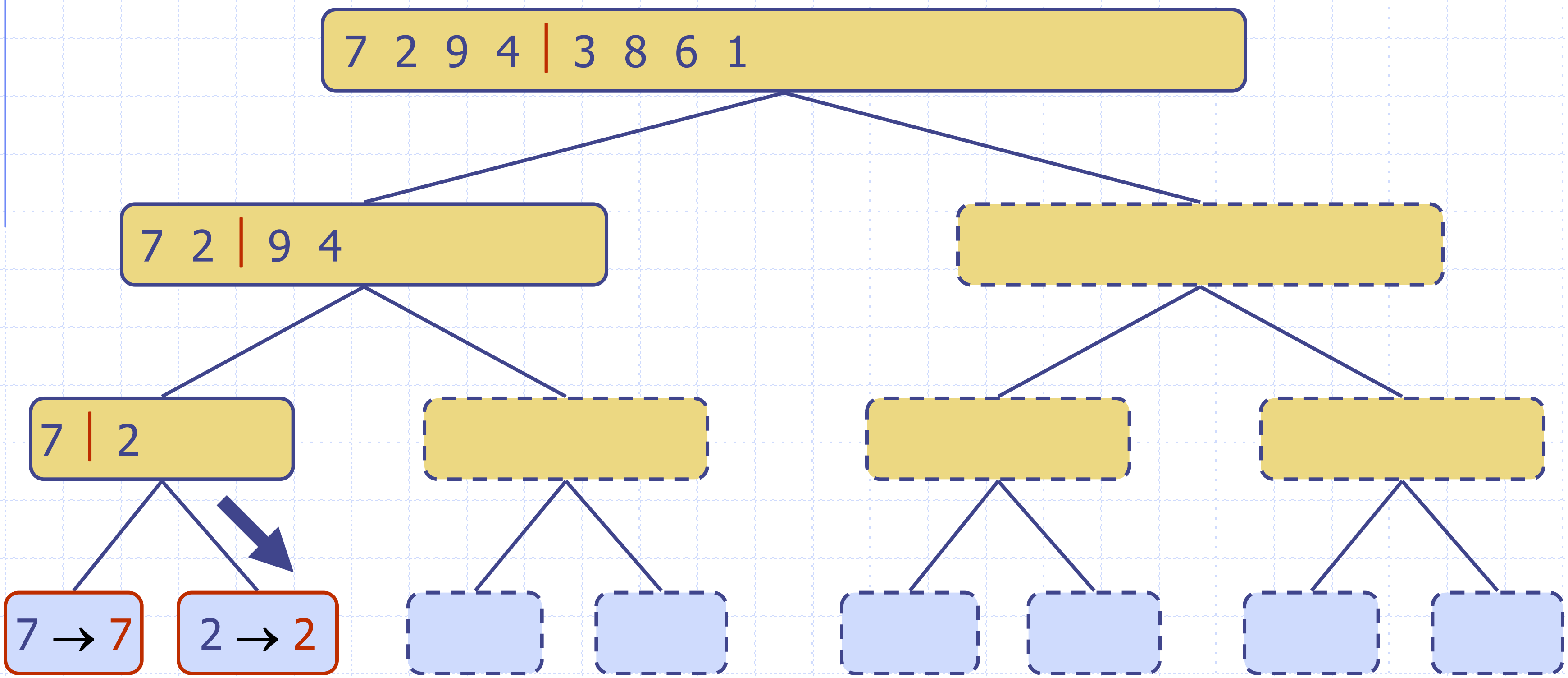
Execution Example (cont.)

- Recursive call, base case



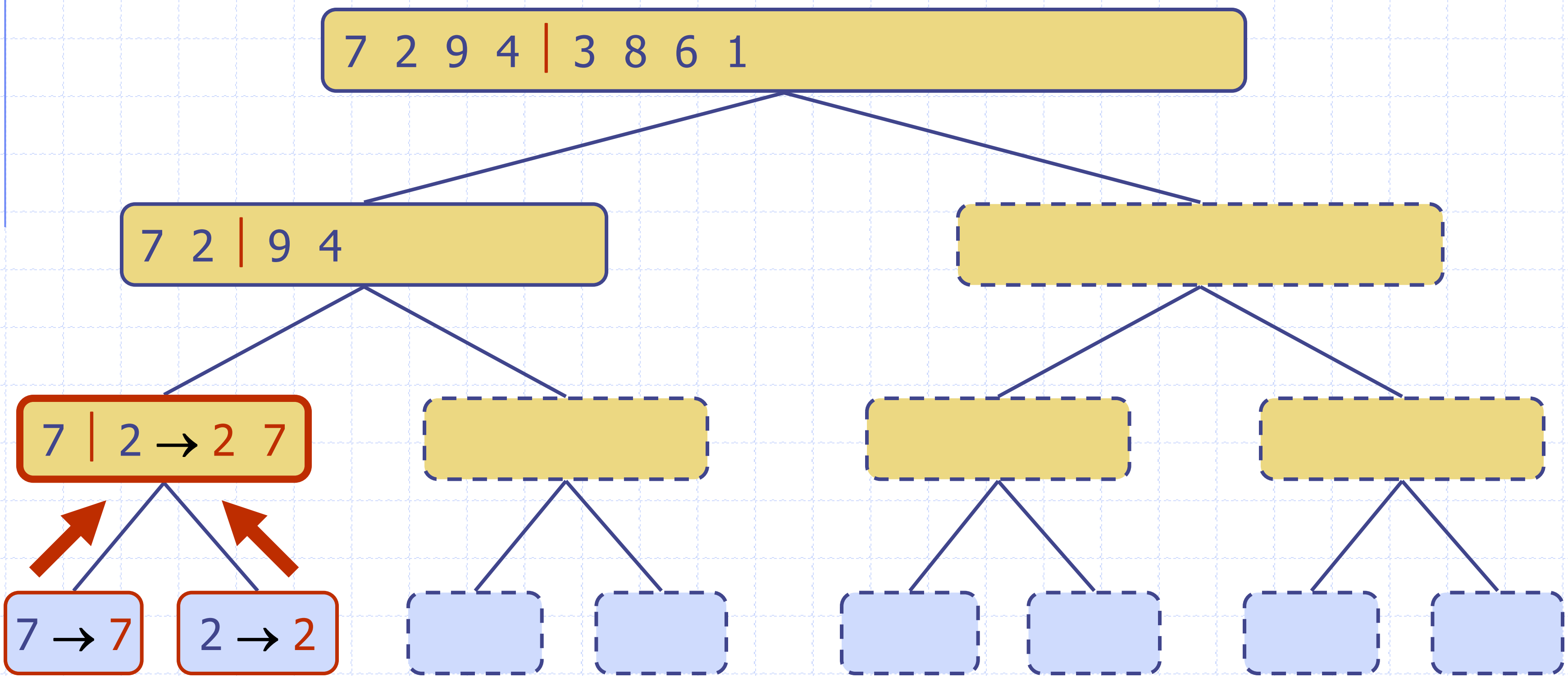
Execution Example (cont.)

- Recursive call, base case



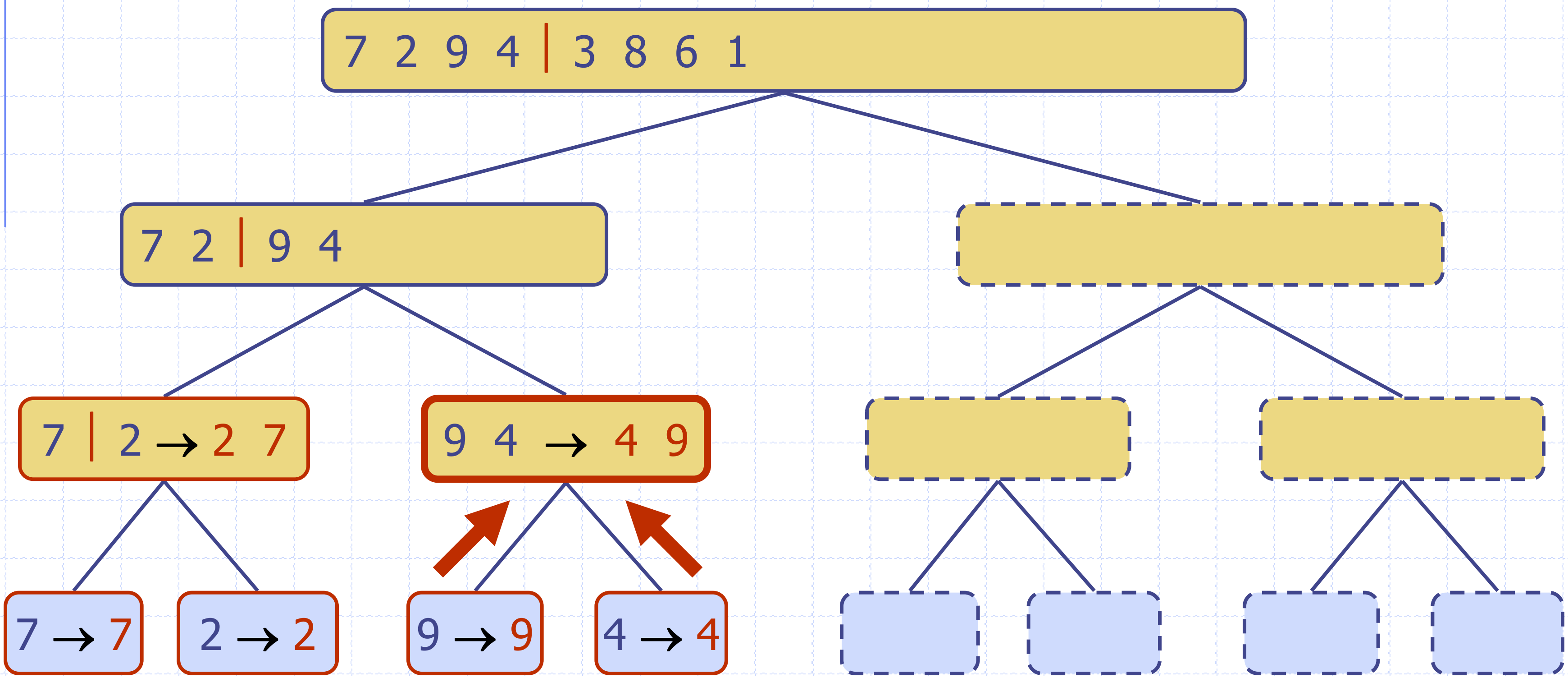
Execution Example (cont.)

- Merge



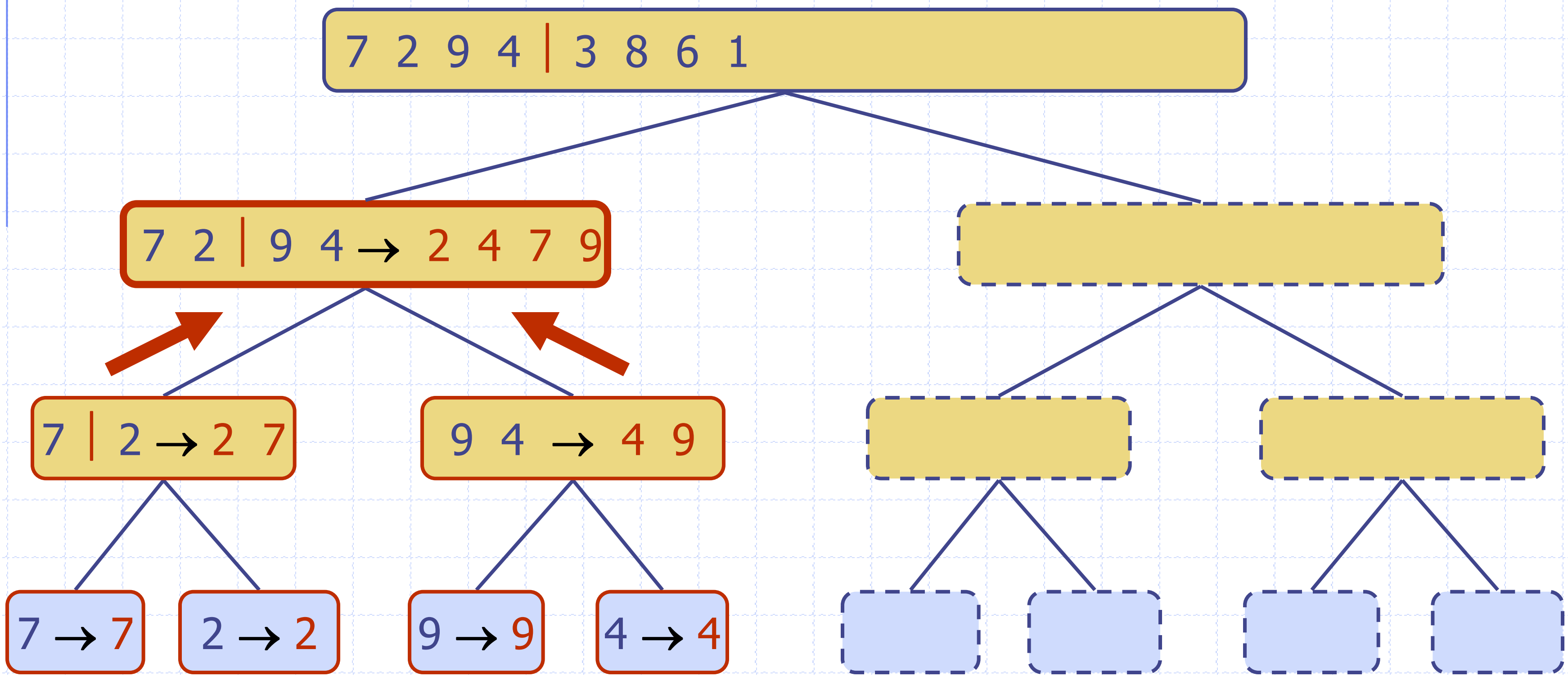
Execution Example (cont.)

- Recursive call, …, base case, merge



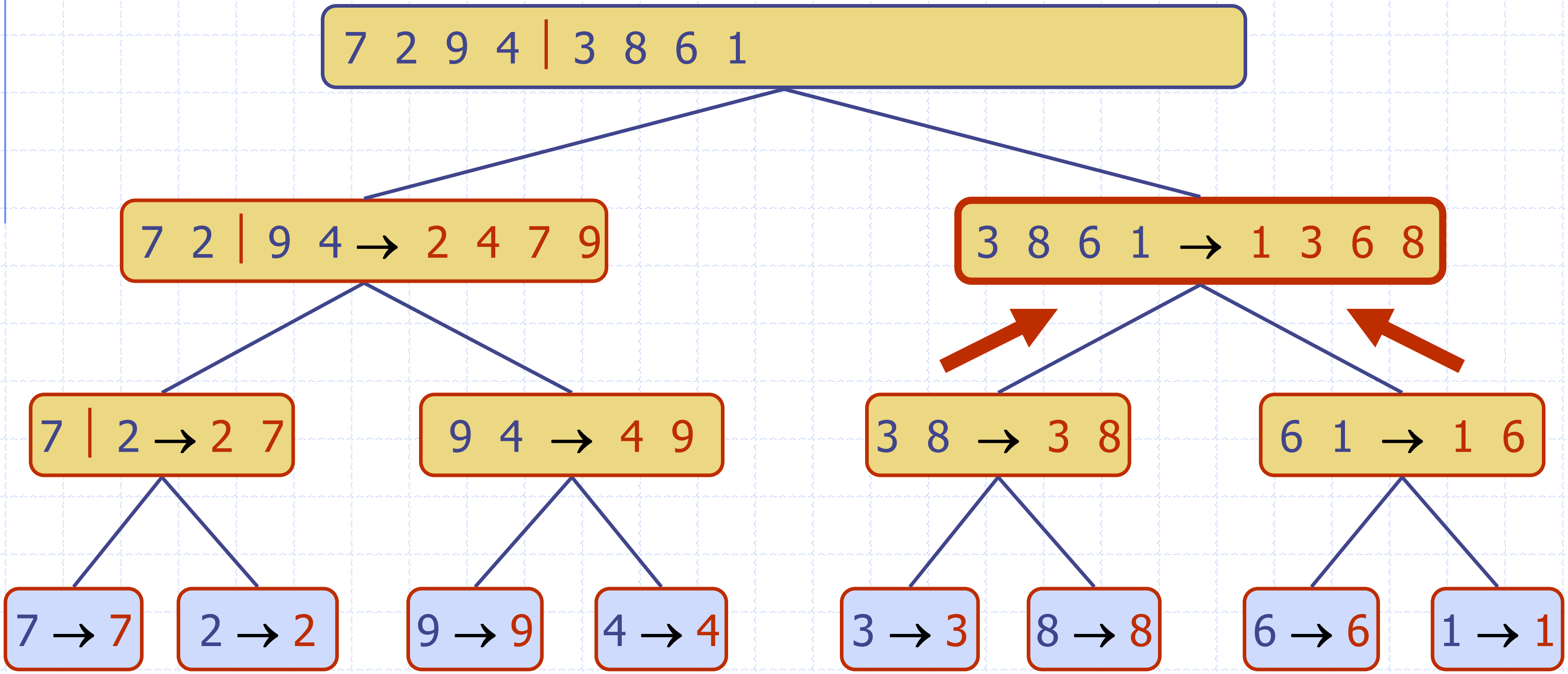
Execution Example (cont.)

- Merge



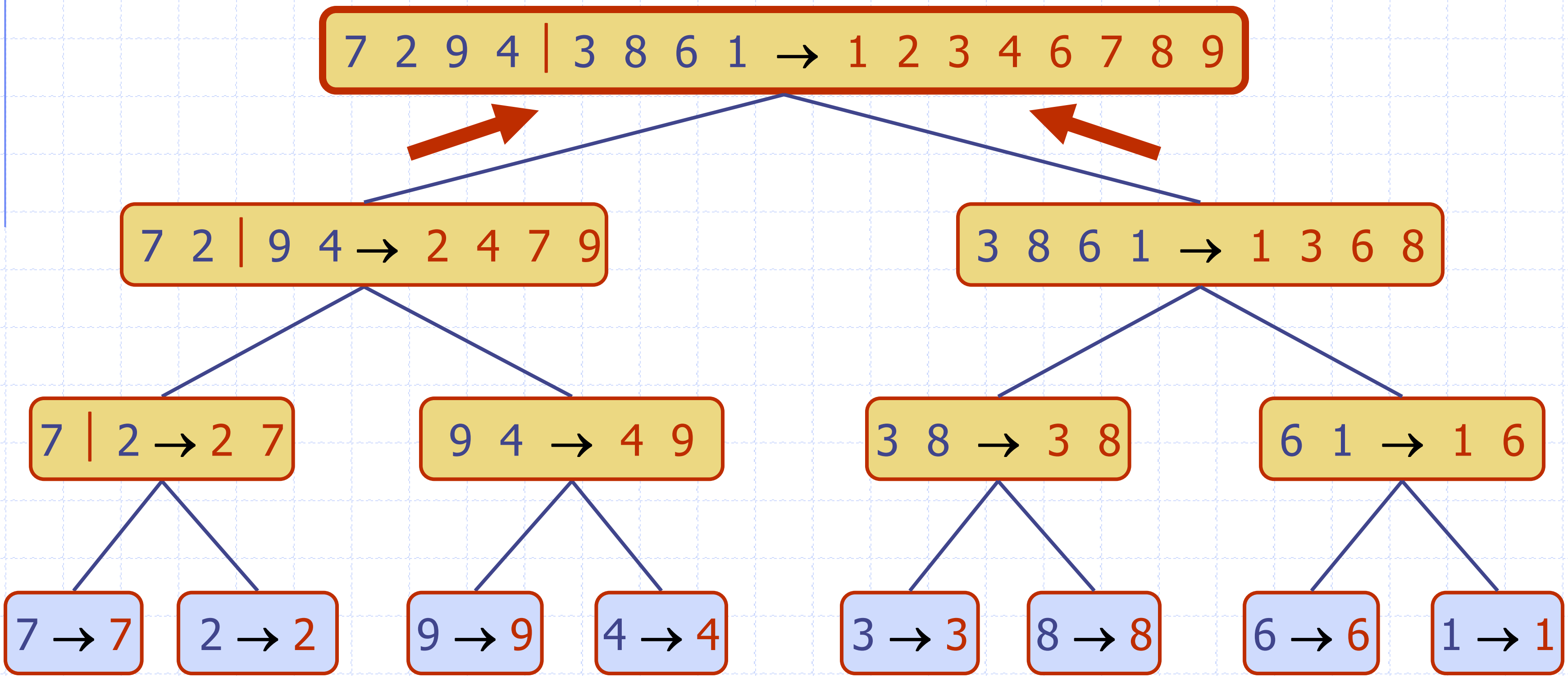
Execution Example (cont.)

- Recursive call, ..., merge, merge

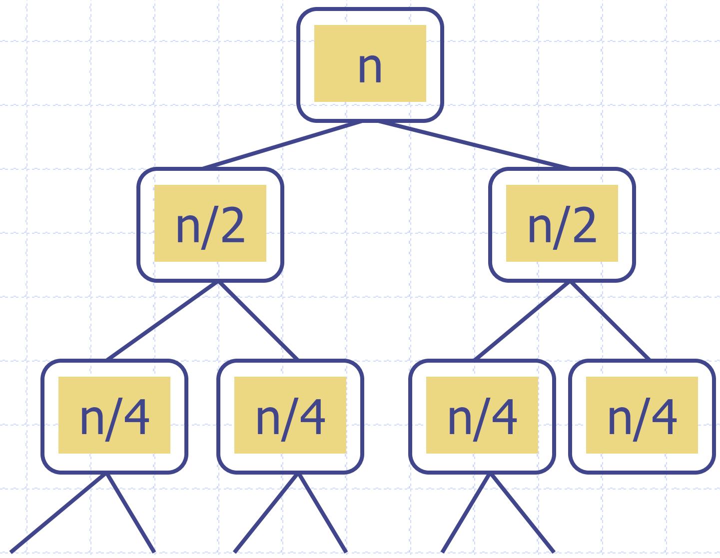


Execution Example (cont.)

- Merge



Tree Exercise



Continue building the tree above until each node at the bottom level contains "1", but no node at a previous level contains a "1" (integer division)

1. What is the height of the tree?
2. Asymptotically, what is the sum of all values contained in the nodes in the tree?

Alternate Analysis of Merge-Sort

- Recall from Lesson 2, there are exactly $1 + m = 1 + \lfloor \log n \rfloor$ terms in the following descending sequence

  $$n, n/2, n/4, \ldots, n/2^m = 1$$

  where 2^m is the largest power of 2 that is $\leq n$.
- The number of levels in the recursion tree $= 1 + \lfloor \log n \rfloor$
- The height h of the merge-sort tree is $O(\log n)$
- The overall amount of work done at each level is $O(n)$
- Thus, the total running time of merge-sort is $O(n \log n)$

Comparison with Other Sorting Algorithms

- Demo confirms that MergeSort's O(nlog n) estimated running time is truly much faster than those of the inversion-bound algorithms and LibrarySort

- Can see why MergeSort is not inversion bound by example: [4, 3, 2, 1]:

Comparison with Other Sorting Algorithms

- Demo confirms that MergeSort's O(nlog n) estimated running time is truly much faster than those of the inversion-bound algorithms and LibrarySort

- Can see why MergeSort is not inversion bound by example: [4, 3, 2, 1]:

  #inversions = 6

  #comparisons = 4

Main Point

By using a Divide and Conquer strategy, MergeSort overcomes the limitations that prevent inversion-bound sorting algorithms from performing faster than n^2. An essential characteristic of this strategy is the relationship of whole to part – wholes are successively collapsed and the collapsed values are combined to produce a new whole. This is different from the incremental approach of inversion-bound algorithms. We see here an application of the MVS principle of *akshara:* Creation arises in the collapse of the unbounded value of wholeness to a point.

Handling Duplicates

- Issue arises during the merge step – if element in left half equals element in right half, insert element in left half first

		d		

		d		

- Stability

Name	Date Received
. . .	. . .
Dave	11/5/2003
Dave	12/1/2004
Dave	1/8/2005
Dave	4/2/2006
. . .	. . .

If you first sort by date (name secondary), then later by name (date secondary), you want dates related to a single name to remain sorted.

Handling Duplicates (cont)

- Definition. Suppose

$$S = <(k_0,e_0), (k_1,e_1),\dots,(k_n,e_n)>$$

is a list of pairs with keys k_0, k_1, …, k_n. A sorting algorithm is *stable* if, whenever it is the case that (k_i, e_i) precedes (k_j, e_j) before sorting (so that $i < j$) and $k_i = k_j$, then it continues to be true after sorting by keys that the pair (k_i, e_i) precedes (k_j, e_j)

Stable sorting does not change the order of duplicates

Stability of Sorting Algorithms

- MergeSort is stable because of our strategy for handling duplicates during Merge

- Are InsertionSort, BubbleSort, SelectionSort stable? (Exercise)

Main Point

Stability of a sorting algorithm requires maintenance of nonchange in the midst of change. This is an example in the world of sorting routines of the inner dynamics of outward success, as described in SCI: The more the inner quality of awareness remains established in silence, the more outer dynamism is supported for success and fulfillment.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Merge Sort

1. Inversion-bound sorting algorithms typically examine each successive element in the input array and perform a further step to place this element in an already sorted area. The style of sorting involves a *sequential unfoldment.*

2. MergeSort proceeds by repeatedly collapsing the wholeness of the current input array into parts and then synthesizing the parts into a sorted whole. This approach yields a much faster sorting algorithm.

3. *Transcendental Consciousness* is the field of *infinite correlation*, where "an impulse anywhere is an impulse everywhere," a field of "frictionless flow".

4. *Impulses within the Transcendental field*. Established in the transcendental field, action reaches fulfillment with minimum effort. Yoga is "skill in action" – efficiency in action, "doing less, accomplishing more", whereby little needs to be done to accomplish great goals.

5. *Wholeness moving within itself.* In Unity Consciousness, the field of action effortlessly unfolds as the play of one's own Self, one's own pure conscioueness.