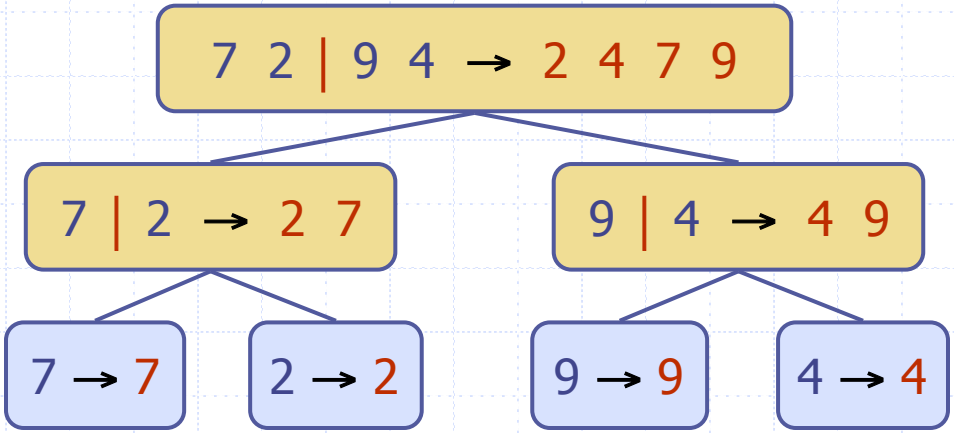
# Sorting



# Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

**Algorithm** *PQ-Sort*(*S*, *C*)
**Input** sequence *S*, comparator *C* for the elements of *S*
**Output** sequence *S* sorted in increasing order according to *C*

```
P ← priority queue with comparator C
while ¬S.isEmpty()
    e ← S.removeFirst()
    P.insert(e, 0)
while ¬P.isEmpty()
    e ← P.removeMin().key()
    S.insertLast(e)
```

# Analyzing Queue Efficiency for Sorting

- N insertElement( ) operations followed by N extractMin( ) operations

Johns Hopkins Department of
Computer Science
Course 600.226: Data Structures,
Professor: Greg Hager

# Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with $n$ insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with $n$ removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

Johns Hopkins Department of
Computer Science
Course 600.226: Data Structures,
Professor: Greg Hager

# Selection-Sort Example

| | ***Sequence*** *S* | ***Priority Queue*** *P* |
|---|---|---|
| Input: | (7,4,8,2,5,3,9) | () |
| Phase 1 | | |
| (a) | (4,8,2,5,3,9) | (7) |
| (b) | (8,2,5,3,9) | (7,4) |
| .. | .. .. | |
| . | . . | |
| (g) | () | (7,4,8,2,5,3,9) |
| Phase 2 | | |
| (a) | (2) | (7,4,8,5,3,9) |
| (b) | (2,3) | (7,4,8,5,9) |
| (c) | (2,3,4) | (7,8,5,9) |
| (d) | (2,3,4,5) | (7,8,9) |
| (e) | (2,3,4,5,7) | (8,9) |
| (f) | (2,3,4,5,7,8) | (9) |
| (g) | (2,3,4,5,7,8,9) | () |

# Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with $n$ insert operations takes time proportional to
     $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of $n$ removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Johns Hopkins Department of
Computer Science
Course 600.226: Data Structures,
Professor: Greg Hager

# Insertion-Sort Example

| | ***Sequence** S* | ***Priority queue** P* |
|---|---|---|
| Input: | (7,4,8,2,5,3,9) | () |
| Phase 1 | | |
| (a) | (4,8,2,5,3,9) | (7) |
| (b) | (8,2,5,3,9) | (4,7) |
| (c) | (2,5,3,9) | (4,7,8) |
| (d) | (5,3,9) | (2,4,7,8) |
| (e) | (3,9) | (2,4,5,7,8) |
| (f) | (9) | (2,3,4,5,7,8) |
| (g) | () | (2,3,4,5,7,8,9) |
| Phase 2 | | |
| (a) | (2) | (3,4,5,7,8,9) |
| (b) | (2,3) | (4,5,7,8,9) |
| .. | .. | .. |
| . | . | . |
| (g) | (2,3,4,5,7,8,9) | () |

# Sort Analysis

- foreach element, $E_i$, in S
  PQ.insert($E_i$)
- while !PQ.empty()
  PQ.extractMin()

| | Sel. Sort | Ins. Sort |
|---|---|---|
| $O(n)$ | | |
| | $\sum_{i=0}^{i=n-1} O(1)$ | $\sum_{i=0}^{i=n-1} O(i)$ |
| $O(n)$ | | |
| | $\sum_{i=0}^{i=n-1} O(i)$ | $\sum_{i=0}^{i=n-1} O(1)$ |

$$O(n) + \sum_{i=0}^{i=n-1} O(1) + \sum_{i=0}^{i=n-1} O(i) = O(n) + O(n) + O(n^2) = O(n^2)$$

# Insertion Sort

for (p=1; p< n; p++)

- tmp = a[p]
- for (j=p; j > 0 && p[j] > tmp; j-=increment)
  - p[j] = p[j-increment]  // SWAP
- a[j] = tmp

- Consider increment = 1
- Complexity governed by # of swaps required to reorder array (O(n + I))
- What are worst case and avg. # of swaps in a randomly generated array?

# Insertion Sort

for (p=1; p< n; p++)

- tmp = a[p]
- for (j=p; j > 0 && p[j] > tmp; j--)
  - p[j] = p[j-1]  // SWAP
- a[j] = tmp

- Complexity governed by # of swaps required to reorder array
- What are worst case and avg. # of swaps in a randomly generated array?

# A Result

- Any sort based on swaps of adjacent elements is $\Omega(n^2)$ on average
- Argument based on average number of disordered elements in a sequence.

# Shell Sort

- Idea: try to do “long-range” swaps
  - Create several “sub-arrays”
  - Sort
  - Reassemble
  - Repeat for other choices of sub-array
- e.g. h = 1, 2, 4, 8, 16 … N/2
  - Is this a good sequence?
- For k = t; k>= 0; k—
  - Insertion sort with increment h[k]

# An Example

| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ | $a_{11}$ | $a_{12}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input data: | 62 | 83 | 18 | 53 | 07 | 17 | 95 | 86 | 47 | 69 | 25 | 28 |
| after 5-sorting: | 17 | 28 | 18 | 47 | 07 | 25 | 83 | 86 | 53 | 69 | 62 | 95 |
| after 3-sorting: | 17 | 07 | 18 | 47 | 28 | 25 | 69 | 62 | 53 | 83 | 86 | 95 |
| after 1-sorting: | 07 | 17 | 18 | 25 | 28 | 47 | 53 | 62 | 69 | 83 | 86 | 95 |

# Some Bounds

- $N/2^k$ (N/2, N/4, … 1): $O(n^2)$
- $2^k$-1 (1 ,3, 7… in reverse order): $O(n^{3/2})$
- $2^p3^q$ (1,2,3,4,8 … IRO): $O(n \log^2 n)$
- $4^k+3*2^{k-1}$ + 1 (1, 8, 23 ..): $O(n^{4/3})$

# Heap Sort

Heap Sort

- foreach element, $E_i$, in S
  PQ.insert($E_i$) — $O(n)$
- while !PQ.empty() — $O(n)$
  PQ.extractMin() — $\sum_{i=0}^{i=n-1} O(\log i)$

$\sum_{i=0}^{i=n-1} O(\log i)$

$$O(n) + 2\sum_{i=0}^{i=n-1} O(\log i) < O(n) + 2\sum_{i=0}^{i=n-1} O(\log n)$$

$$= O(n) + 2n*O(\log n) = O(n \log n)$$

(showing $\theta(n \log n)$ is a bit harder)

# Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $S$ in two disjoint subsets $S_1$ and $S_2$
  - Recur: solve the subproblems associated with $S_1$ and $S_2$
  - Conquer: combine the solutions for $S_1$ and $S_2$ into a solution for $S$
- The base case for the recursion are subproblems of size 0 or 1
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

# Merge-Sort

- Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:
  - Divide: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
  - Recur: recursively sort $S_1$ and $S_2$
  - Conquer: merge $S_1$ and $S_2$ into a unique sorted sequence

**Algorithm** *mergeSort*($S$, $C$)
  **Input** sequence $S$ with $n$ elements, comparator $C$
  **Output** sequence $S$ sorted according to $C$
  **if** $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    *mergeSort*($S_1$, $C$)
    *mergeSort*($S_2$, $C$)
    $S \leftarrow merge(S_1, S_2)$

# Merging Two Sorted Sequences

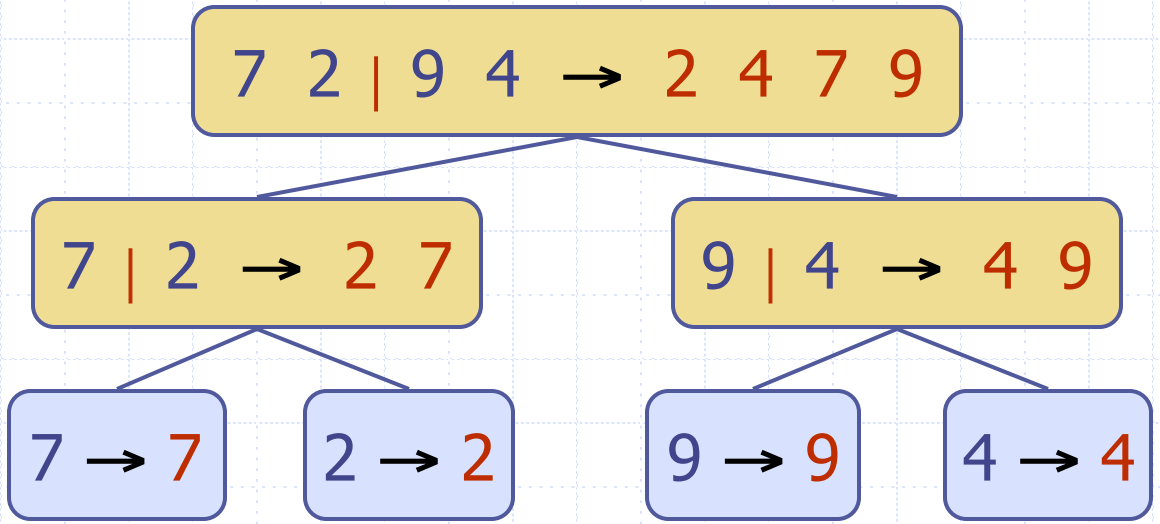
- The conquer step of merge-sort consists of merging two sorted sequences $A$ and $B$ into a sorted sequence $S$ containing the union of the elements of $A$ and $B$
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

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

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

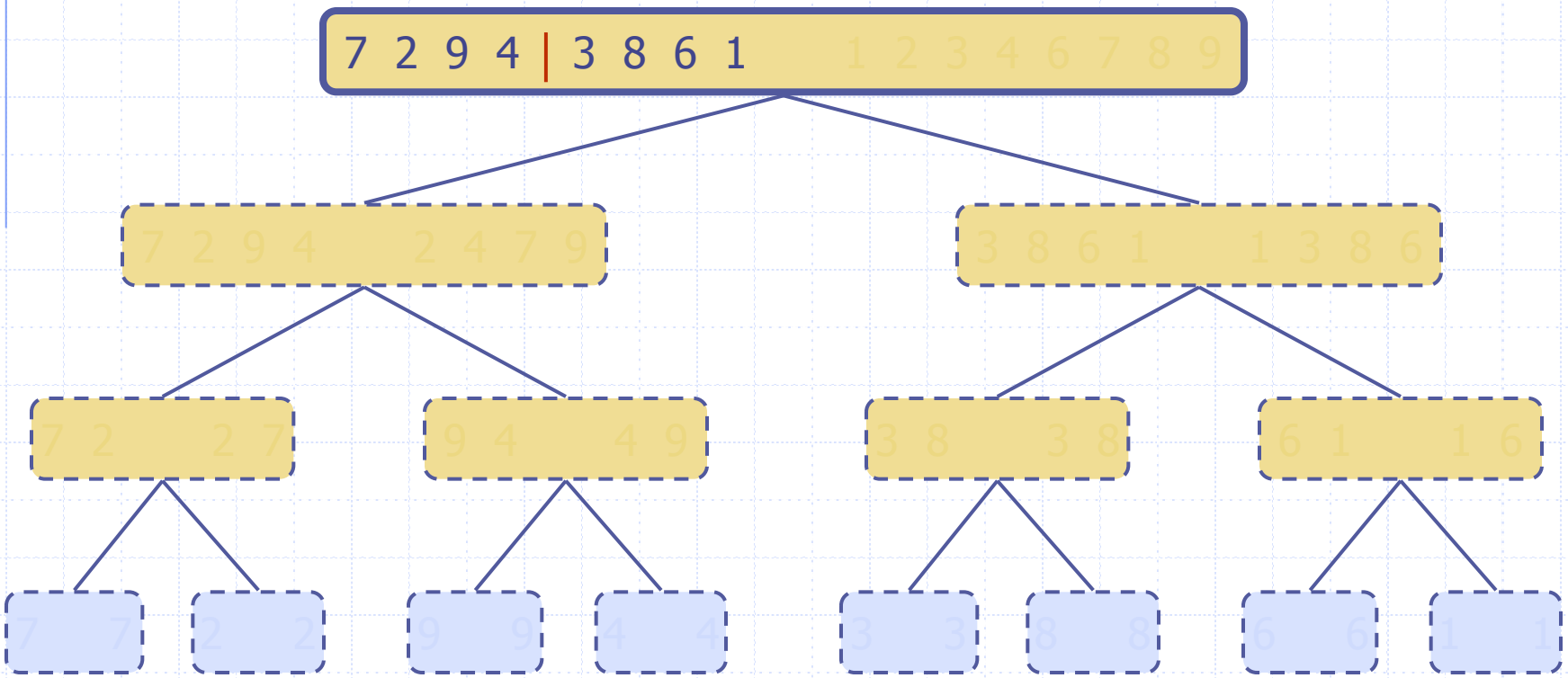
# Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and 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



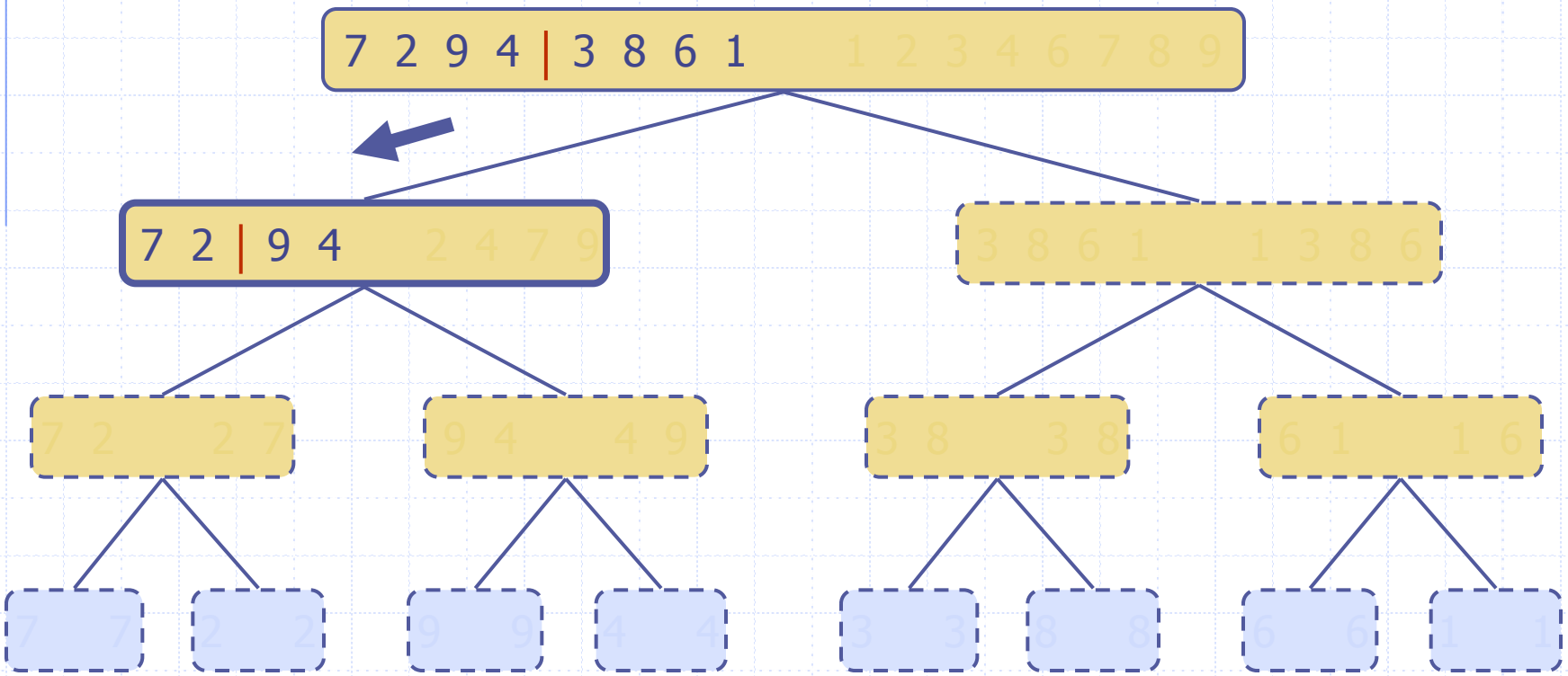
# Execution Example

- Partition



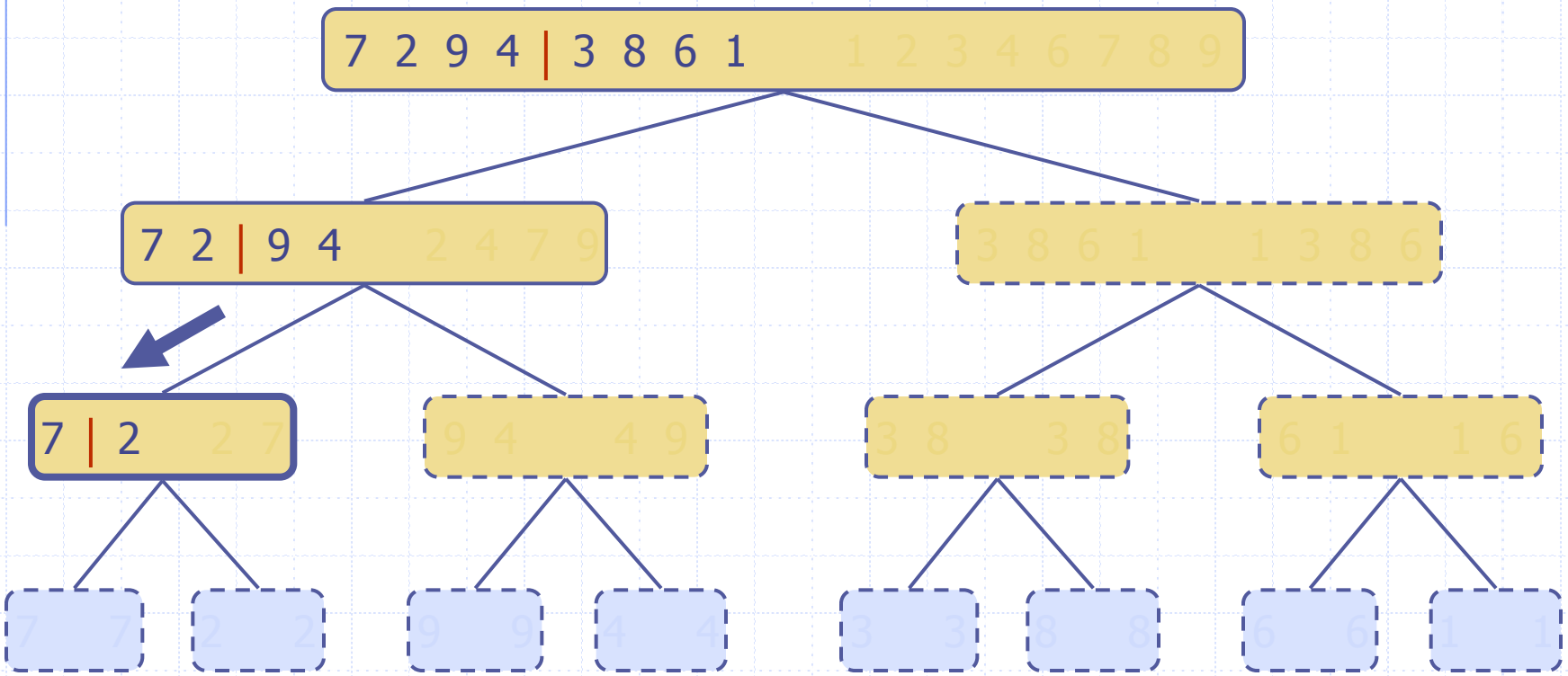
# Execution Example (cont.)

- Recursive call, partition



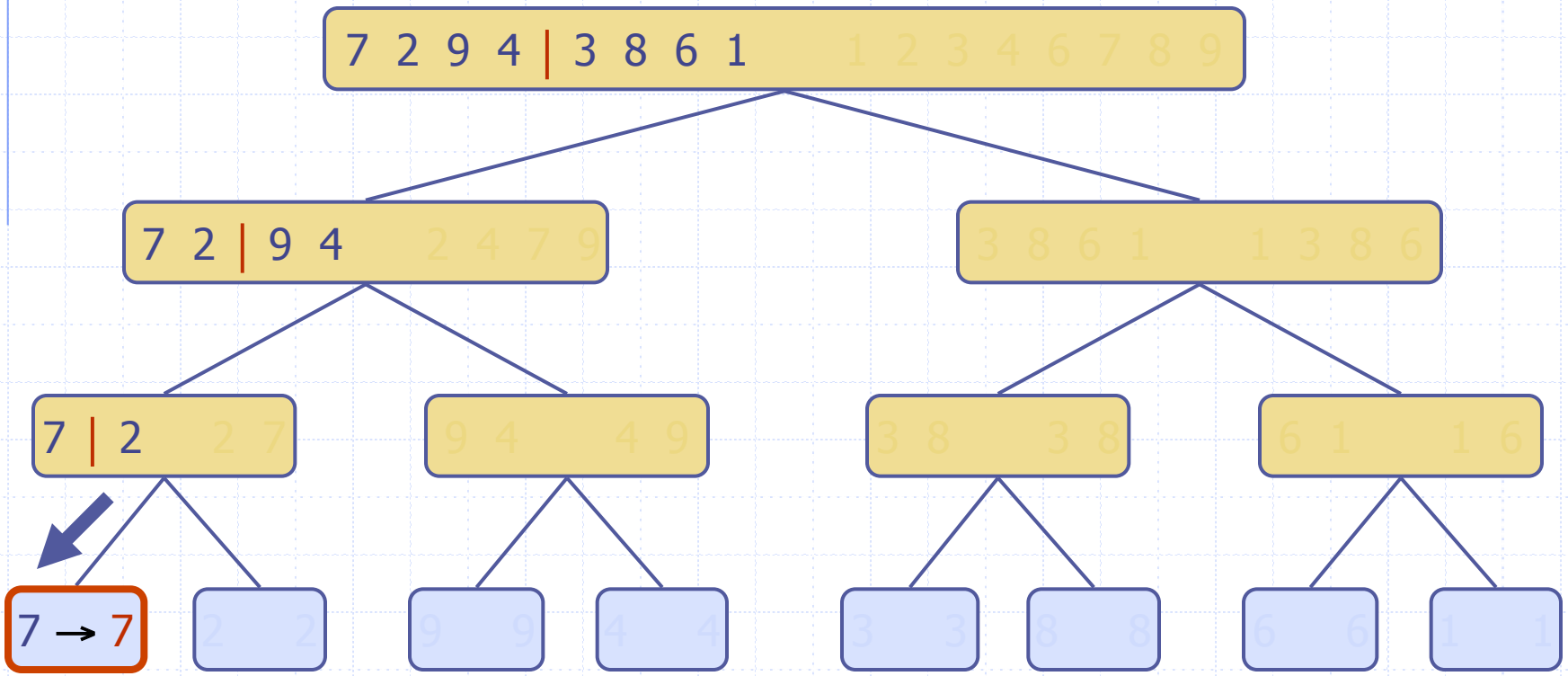
# Execution Example (cont.)

- Recursive call, partition



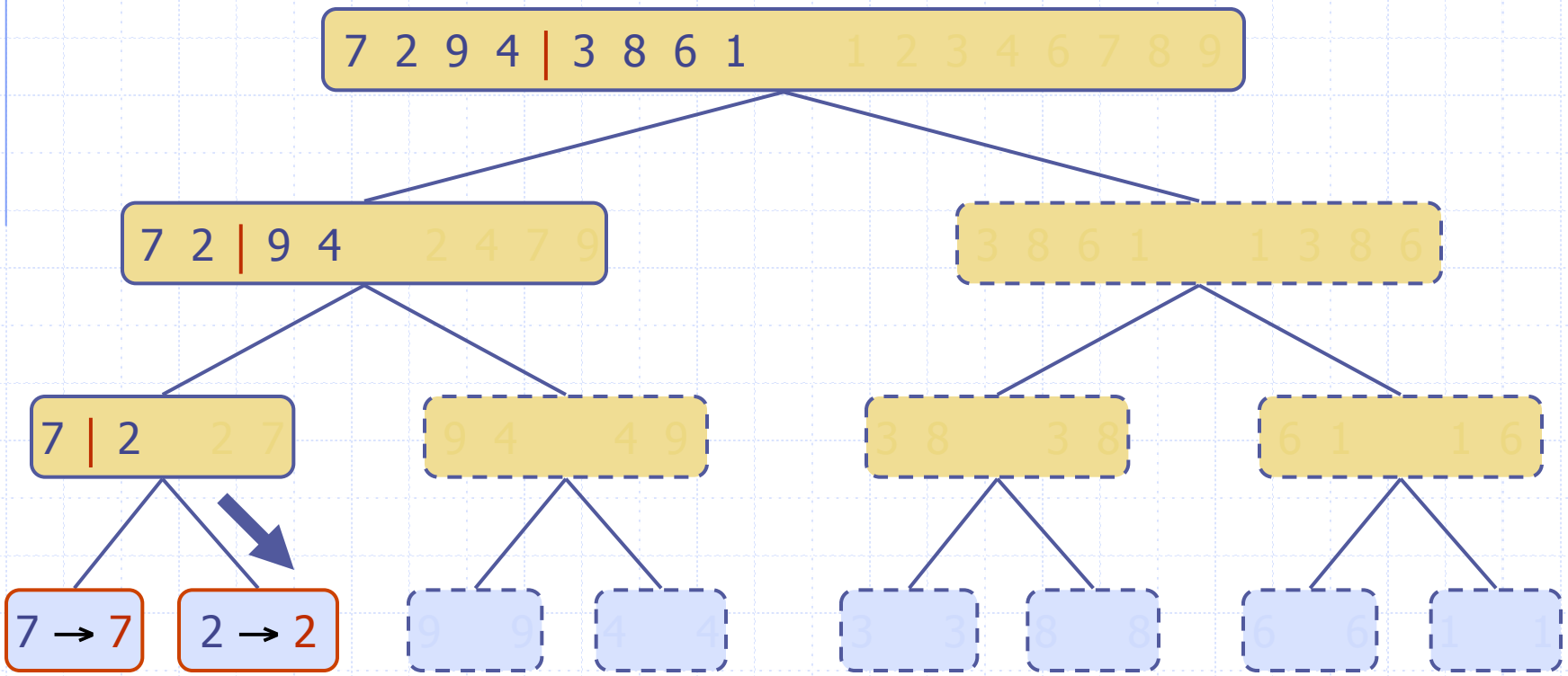
# Execution Example (cont.)

- Recursive call, base case



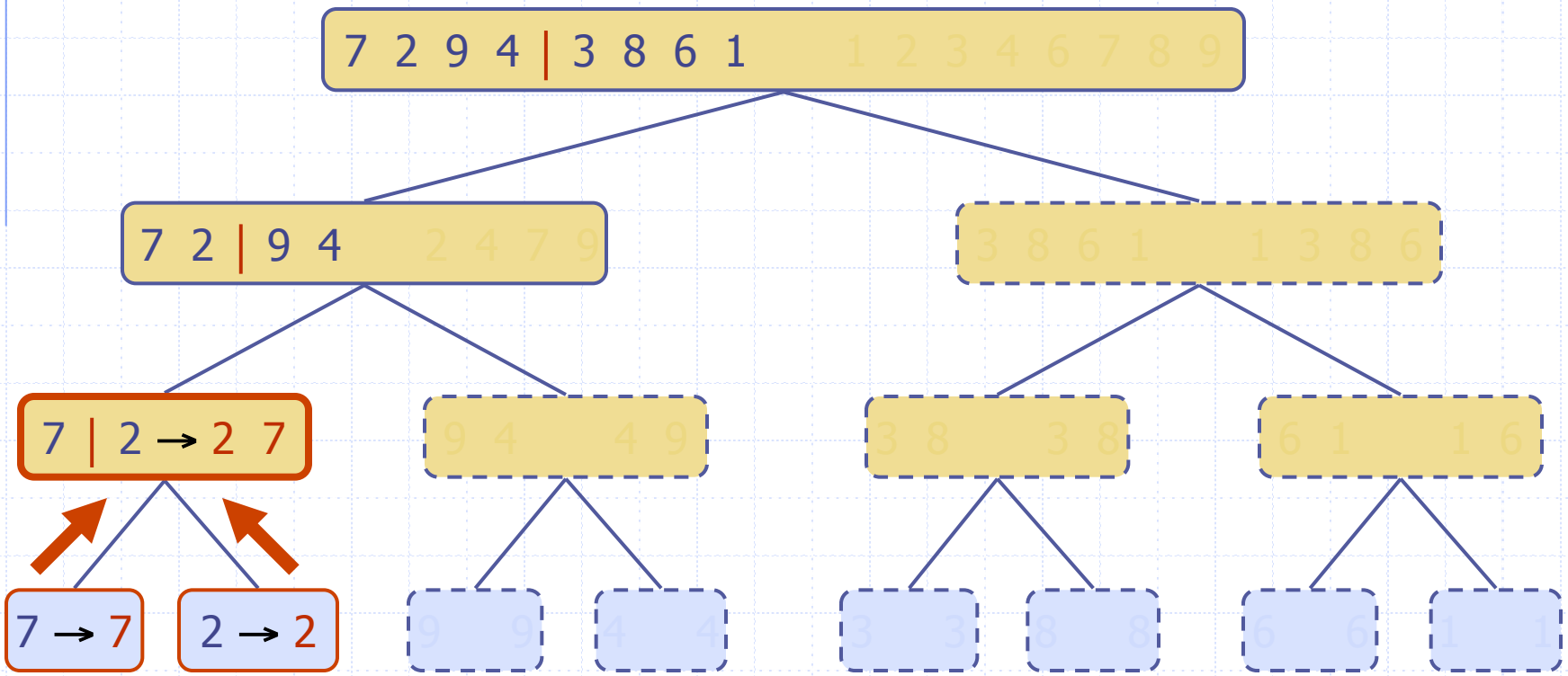
# Execution Example (cont.)

- Recursive call, base case



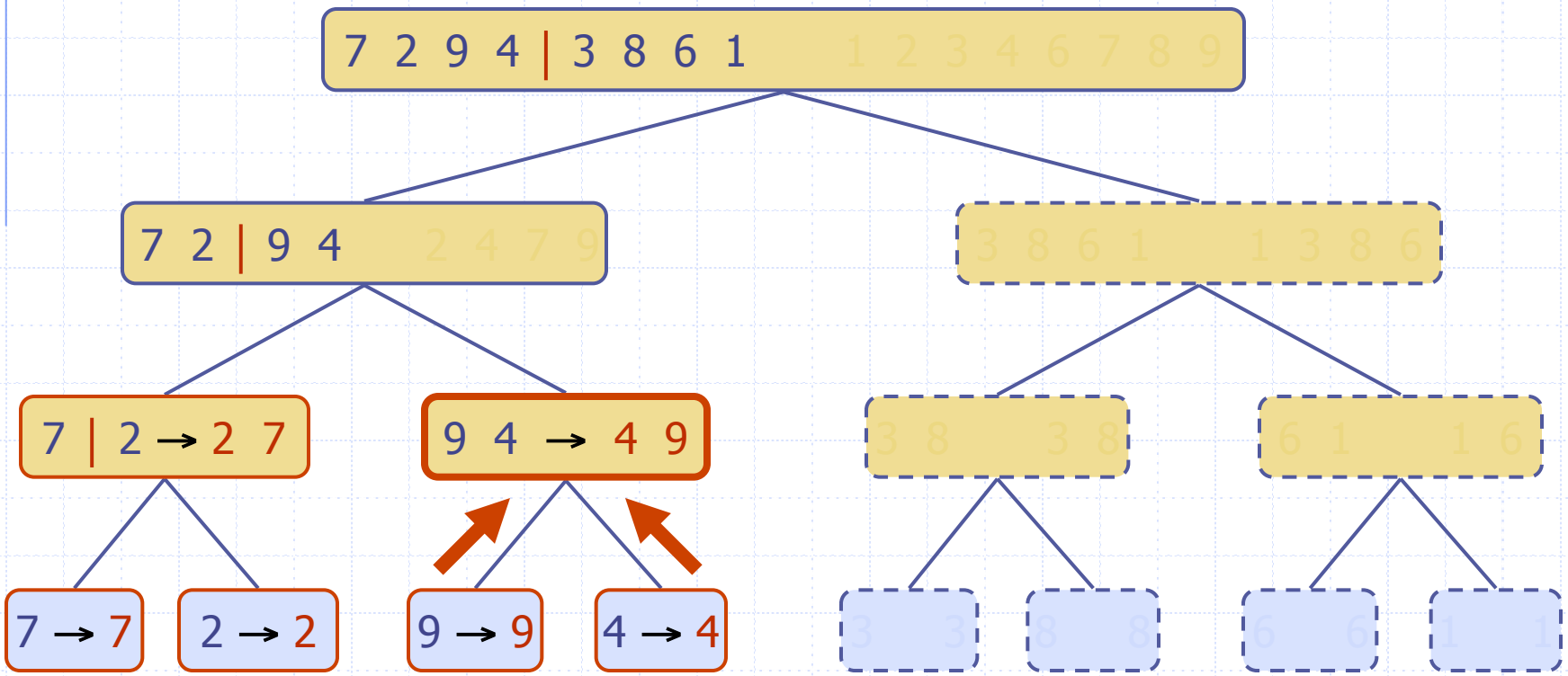
# Execution Example (cont.)

- Merge



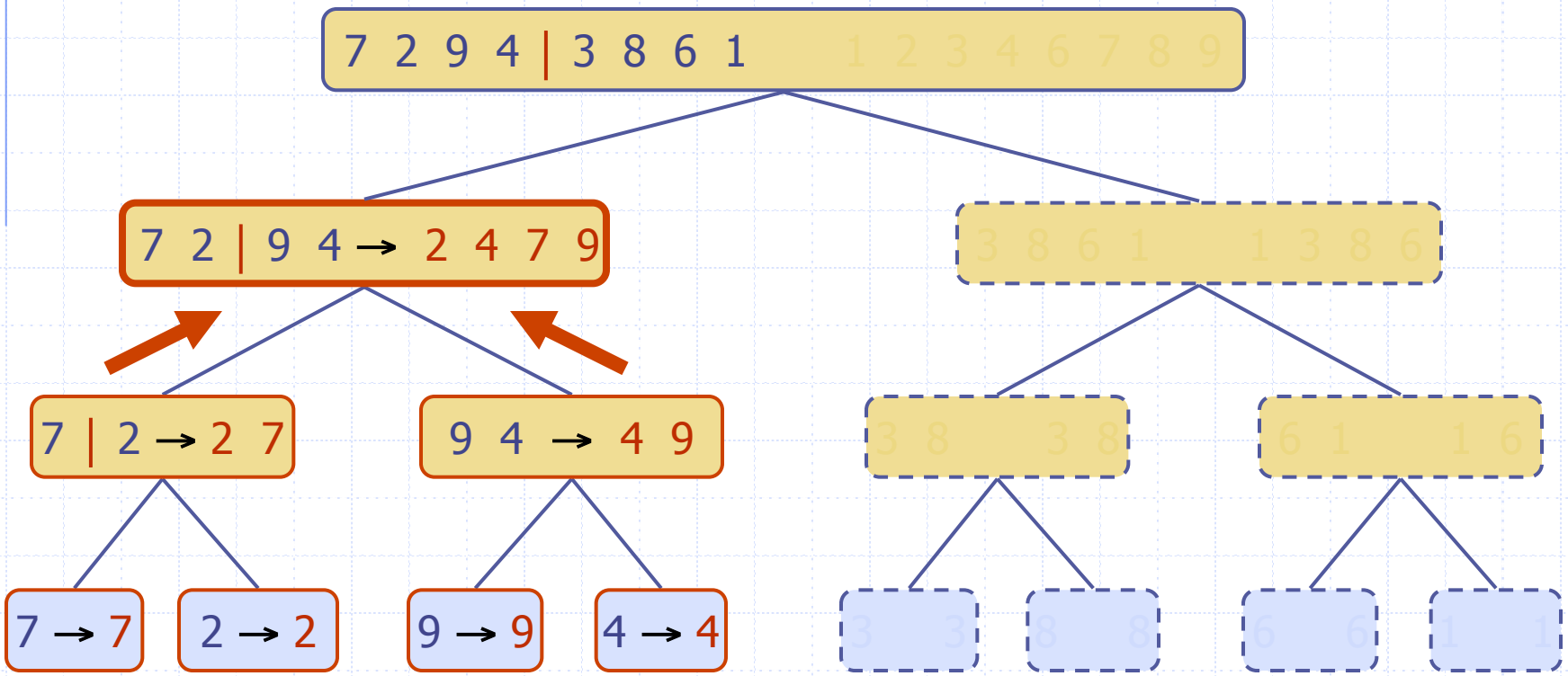
# Execution Example (cont.)

- Recursive call, …, base case, merge



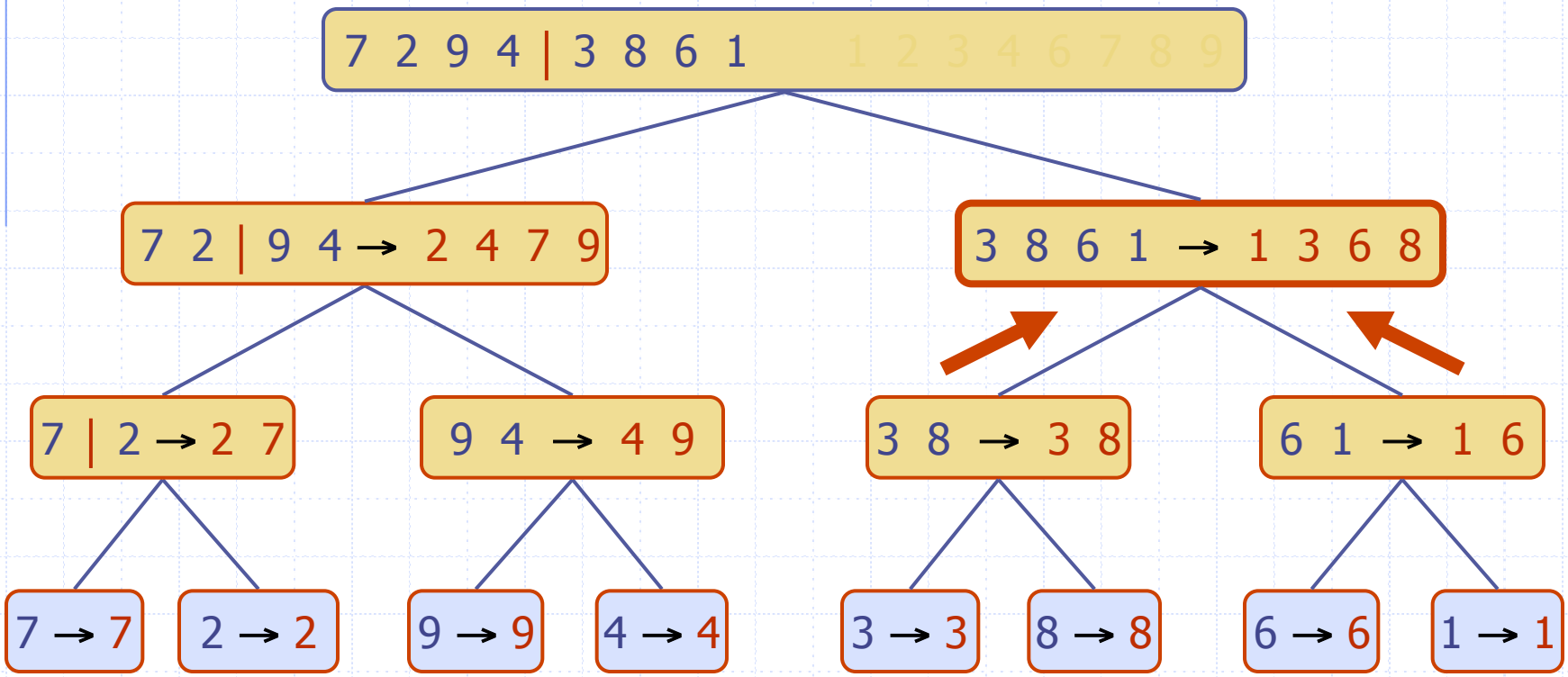
# Execution Example (cont.)

- Merge



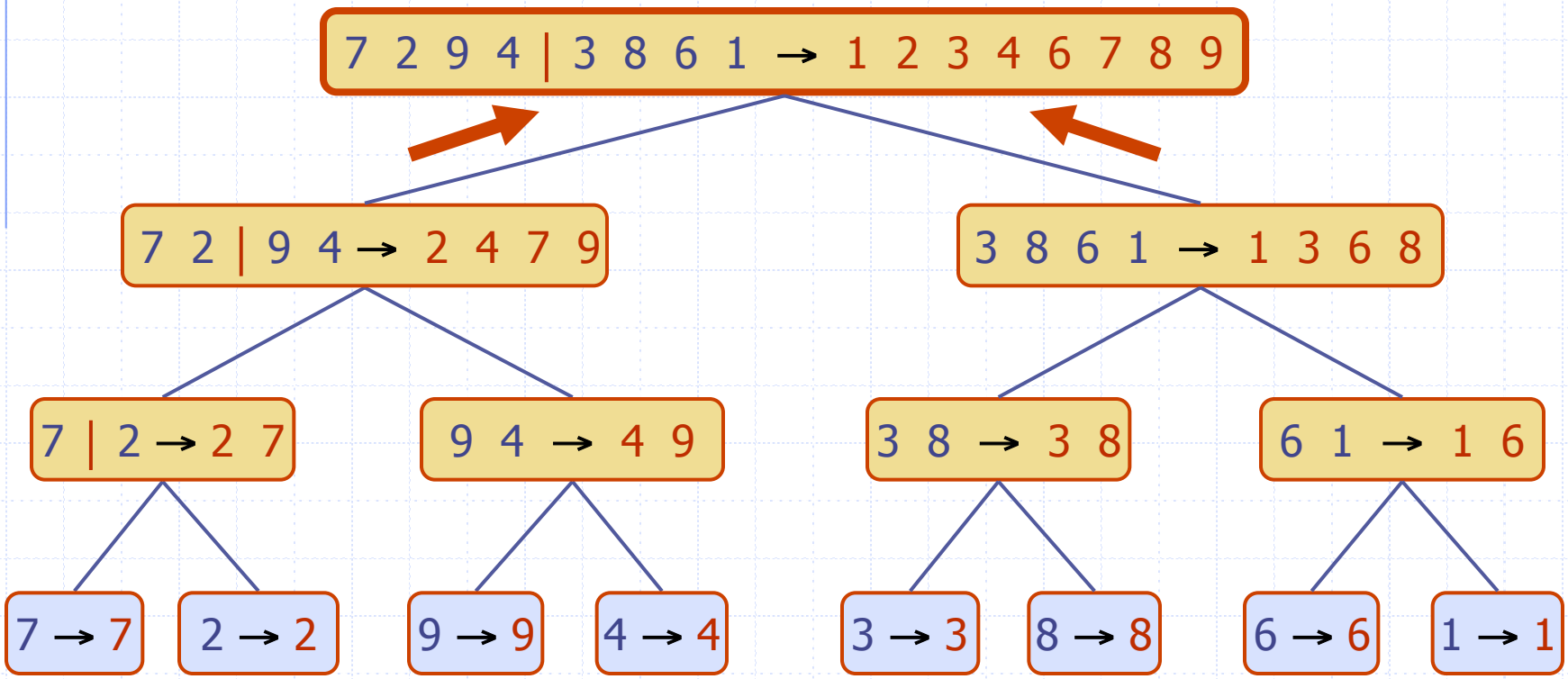
# Execution Example (cont.)

- Recursive call, ..., merge, merge



# Execution Example (cont.)

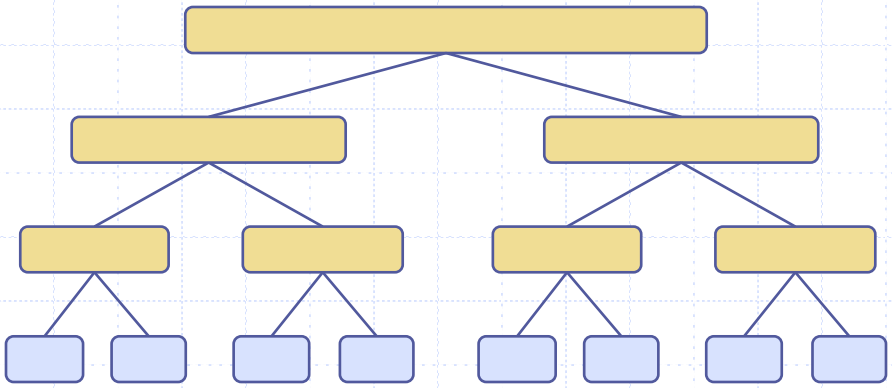
- Merge



# Analysis of Merge-Sort

- The height $h$ of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth $i$ is $O(n)$
  - we partition and merge $2^i$ sequences of size $n/2^i$
  - we make $2^{i+1}$ recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

| depth | #seqs | size |
|---|---|---|
| 0 | 1 | $n$ |
| 1 | 2 | $n/2$ |
| $i$ | $2^i$ | $n/2^i$ |
| … | … | … |



# Summary of Sorting Algorithms

| Algorithm | Time | Notes |
|---|---|---|
| selection-sort | $O(n^2)$ | ◆ slow<br>◆ in-place<br>◆ for small data sets (< 1K) |
| insertion-sort | $O(n^2)$ | ◆ slow<br>◆ in-place<br>◆ for small data sets (< 1K) |
| heap-sort | $O(n \log n)$ | ◆ fast<br>◆ in-place<br>◆ for large data sets (1K — 1M) |
| merge-sort | $O(n \log n)$ | ◆ fast<br>◆ sequential data access<br>◆ for huge data sets (> 1M) |

# Nonrecursive Merge-Sort

```
public static void mergeSort(Object[] orig, Comparator c) { // nonrecursive
    Object[] in = new Object[orig.length]; // make a new temporary array
    System.arraycopy(orig,0,in,0,in.length); // copy the input
    Object[] out = new Object[in.length]; // output array
    Object[] temp; // temp array reference used for swapping
    int n = in.length;
    for (int i=1; i < n; i*=2) { // each iteration sorts all length-2*i runs
      for (int j=0; j < n; j+=2*i)  // each iteration merges two length-i pairs
        merge(in,out,c,j,i); // merge from in to out two length-i runs at j
      temp = in; in = out; out = temp; // swap arrays for next iteration
    }
    // the "in" array contains the sorted array, so re-copy it
    System.arraycopy(in,0,orig,0,in.length);
  }
```

merge runs of length 2, then 4, then 8, and so on

# Nonrecursive Merge-Sort

```
protected static void merge(Object[] in, Object[] out, Comparator c, int
    start,
    int inc) { // merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
  int x = start; // index into run #1
  int end1 = Math.min(start+inc, in.length); // boundary for run #1
  int end2 = Math.min(start+2*inc, in.length); // boundary for run #2
  int y = start+inc; // index into run #2 (could be beyond array boundary)
  int z = start; // index into the out array
  while ((x < end1) && (y < end2))
    if (c.compare(in[x],in[y]) <= 0) out[z++] = in[x++];
    else out[z++] = in[y++];
  if (x < end1) // first run didn't finish
    System.arraycopy(in, x, out, z, end1 - x);
  else if (y < end2) // second run didn't finish
    System.arraycopy(in, y, out, z, end2 - y);
}
```

merge two runs in the in array to the out array

# Lower bound for sorting,

Taken from notes by Long Quan

# Lower Bound for Sorting

- Mergesort and QuickSort
  - worst-case running time is O(N log N)
- Are there better algorithms?
- Goal: Prove that any sorting algorithm based on only comparisons takes $\Omega$(N log N) comparisons in the worst case (worse-case input) to sort N elements.

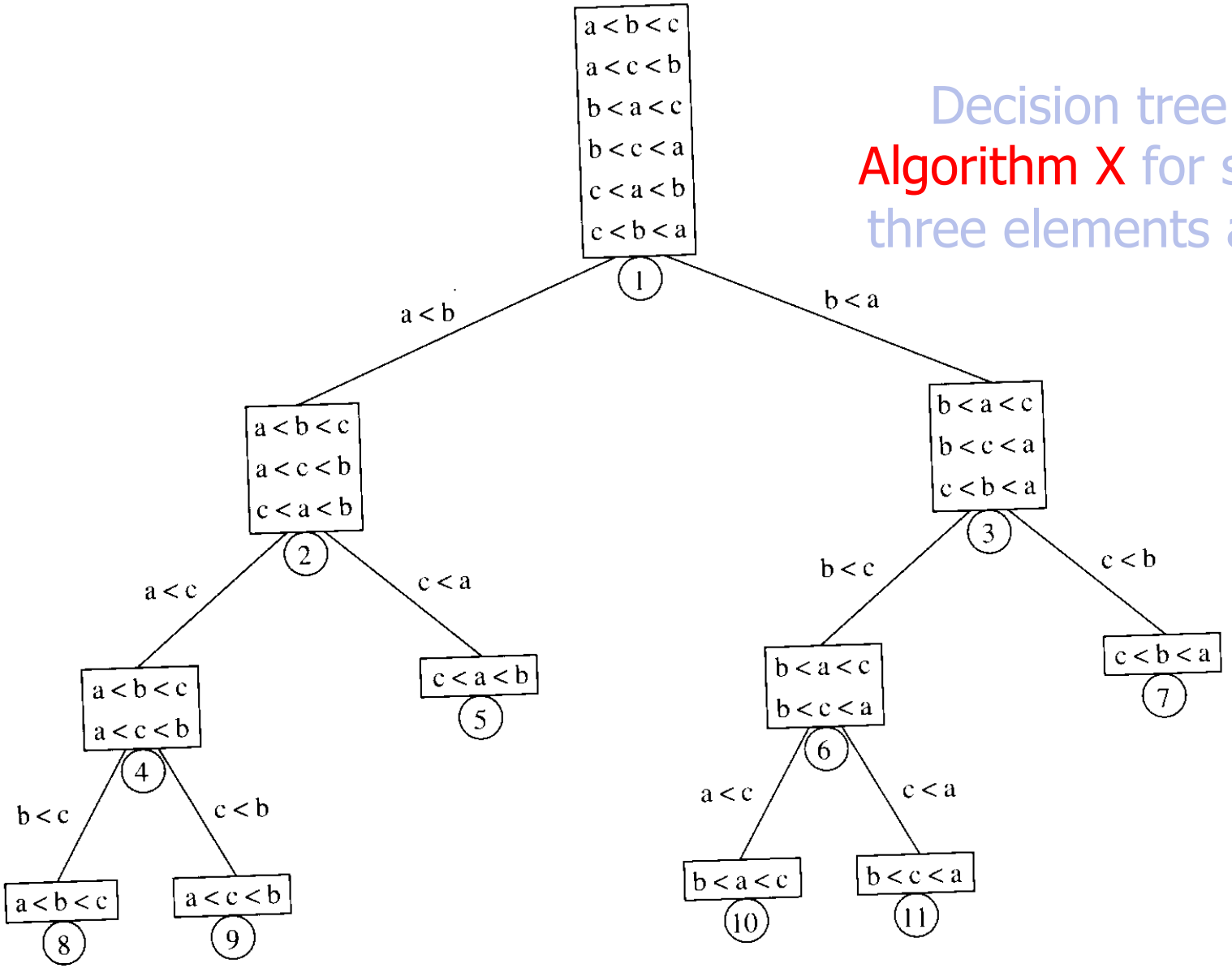
# Lower Bound for Sorting

- Suppose we want to sort N distinct elements
- How many possible orderings do we have for N elements?

# Lower Bound for Sorting

- Any comparison-based sorting process can be represented as a binary decision tree.
  - Each node represents a set of possible orderings, consistent with all the comparisons that have been made
  - The tree edges are results of the comparisons

# Decision tree for Algorithm X for sorting three elements a, b, c



# Lower Bound for Sorting

- The worst-case number of comparisons used by the sorting algorithm is equal to the height of the tree
- A decision tree to sort N elements must have N! leaves
  - the decision tree with N! leaves must have depth at least $\lceil \log_2 (N!) \rceil$
- Therefore, any sorting algorithm based on only comparisons between elements requires $\lceil \log_2(N!) \rceil$ comparisons in the worst case.

# Lower Bound for Sorting

$$
\begin{aligned}
\log_2(N!) &= \log(N(N-1)(N-2)\cdots(2)(1)) \\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\
&\geq \frac{N}{2}\log\frac{N}{2} \\
&= \frac{N}{2}\log N - \frac{N}{2} \\
&= \Omega(N \log N)
\end{aligned}
$$

Thus, any sorting algorithm based on comparisons between elements requires $\Omega$(N log N) comparisons in the worst case.

# Beating n log(n): Bucket Sort

- Constraint:
  - Input numbers are positive integers < M
  - Example S = { 1  5  2  4  3  2  1  7  10  9}
  - Count = Array [10] contains frequency of each element
  - Count = {2  2  1  1  1  0  1  0  1  1 }
  - Scan count to sort the list

# Bucket Sorting

- Algorithm:
  - Initialize array count of size M to zeros
  - For each element e
    - Count(e) ++;
  - For each cell in count print the corresponding elements

# Bucket Sort

- Running Time:
  - O(m +n)
  - If m is O(n) -→ It is O(n)
- In general, special input properties can be used to obtain complexity less than O(nlogn)

# Stable sorting and Radix Sort

- Suppose we want to alphabetize words
- Sort by letters in reverse order
- Sort must be *stable* -- preserves order of equal items
- O(p(M+N))

# Quick Sort

Shamelessly copied from M. B. Fayek Cairo University

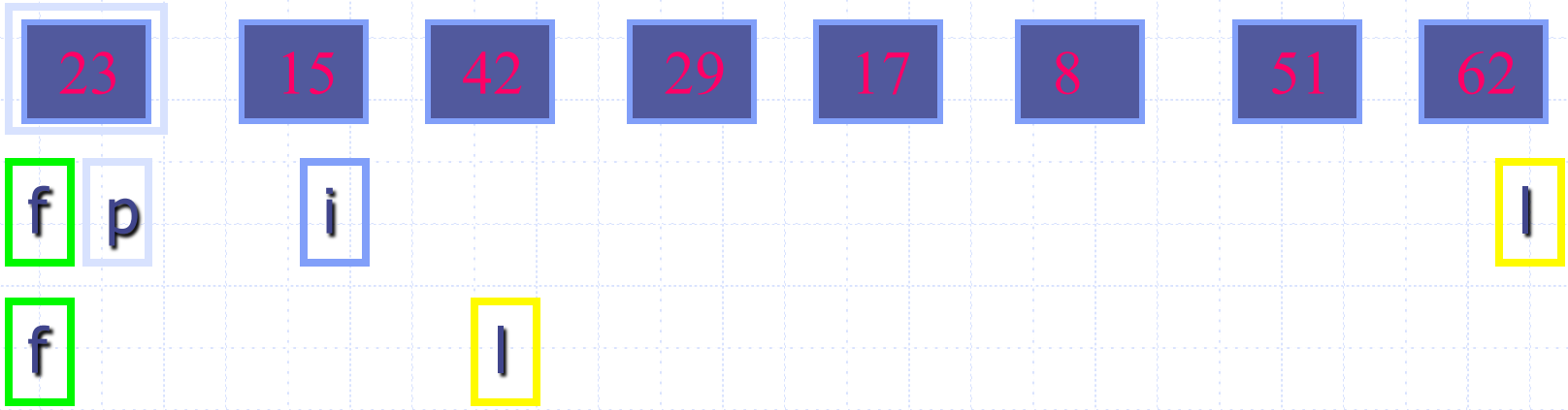
# Quick Sort

- Most popular sorting algorithm, **fastest** sorting algorithms ever
- The **basic strategy** is
  - Focus on one record (referred to as pivot element)
  - Move larger records to the right and smaller records to left (partition)
  - Sort left and right partitions (recurse)
  - Concatenate the results

# Partition Strategy

PivotList( list, first, last )

QuickSort(list, first, pivot-1)



# Quiz

- What is the running time for
  - Sorted list
  - Reversed order list ?

# Picking The pivot Element

- First Element (What happens if the list if sorted)
- Randomly
- Median of 3 random elements.

# Median routine

```
private static <AnyType extends Comparable<? super AnyType>>
    AnyType median3( AnyType [ ] a, int left, int right )
    {
        int center = ( left + right ) / 2;
        if( a[ center ].compareTo( a[ left ] ) < 0 )
            swapReferences( a, left, center );
        if( a[ right ].compareTo( a[ left ] ) < 0 )
            swapReferences( a, left, right );
        if( a[ right ].compareTo( a[ center ] ) < 0 )
            swapReferences( a, center, right );

            // Place pivot at position right - 1
        swapReferences( a, center, right - 1 );
        return a[ right - 1 ];
    }
```

```
private static <AnyType extends Comparable<? super AnyType>>
    void quicksort( AnyType [ ] a, int left, int right )
    {
        if( left + CUTOFF <= right )
        {
            AnyType pivot = median3( a, left, right );

              int i = left, j = right - 1;
            for( ; ; )
            {
                while( a[ ++i ].compareTo( pivot ) < 0 ) { }
                while( a[ --j ].compareTo( pivot ) > 0 ) { }
                if( i < j )
                    swapReferences( a, i, j );
                else
                    break;
            }

            swapReferences( a, i, right - 1 );   // Restore pivot

            quicksort( a, left, i - 1 );    // Sort small elements
            quicksort( a, i + 1, right );   // Sort large elements
        }
        else  // Do an insertion sort on the subarray
            insertionSort( a, left, right );
    }
```

# Quick Sort Analysis

- Best Case: When list is always divided at mid
  - → Complexity = N * logN
- Worst Case: When list is sorted
  - → Complexity =

$$\sum_{i=2}^{N}(i-1) = \frac{N(N-1)}{2}$$

# Average Case

- Assume pivot leads to random (uniformly) sizes
- Use this to compute average case analysis

# Quick Select

- Note we can use the same idea to find the median (or any other statistics)
- Just recurse on the appropriate side

# Sorting Summary

- Basic sorting methods and their complexities, advantages, disadvantages
  - Insertion, Heap, Merge, Quick, Bucket/Radix
- Lower bound for worst-case sorting behavior