Recursion: A tool for Algorithm Designer

Atul Gupta

ES103: Course Summary

- Estimating algorithmic complexities
- Tools like Recursion, Iteration
- ADT
  - Linear data organizations like Arrays, Stacks, queues, Linked lists
  - Non-linear data organizations like hashing, trees (and its many variances), graphs (and many variances)
  - Examples
- Standard Problems: Searching and Sorting
- Standard Problem Solving Approaches like **"Divide and Conquer",** Greedy, Dynamic Programming, Branch & Bound, Backtracking

Recursion

- A programming tool for implementing “Divide and Conquer” Approach of Problem Solving
- Infinite computations by finite statements
- Recursive computations/functions call themselves
- A recursive function must have
  - Expression for sub division
  - Termination condition

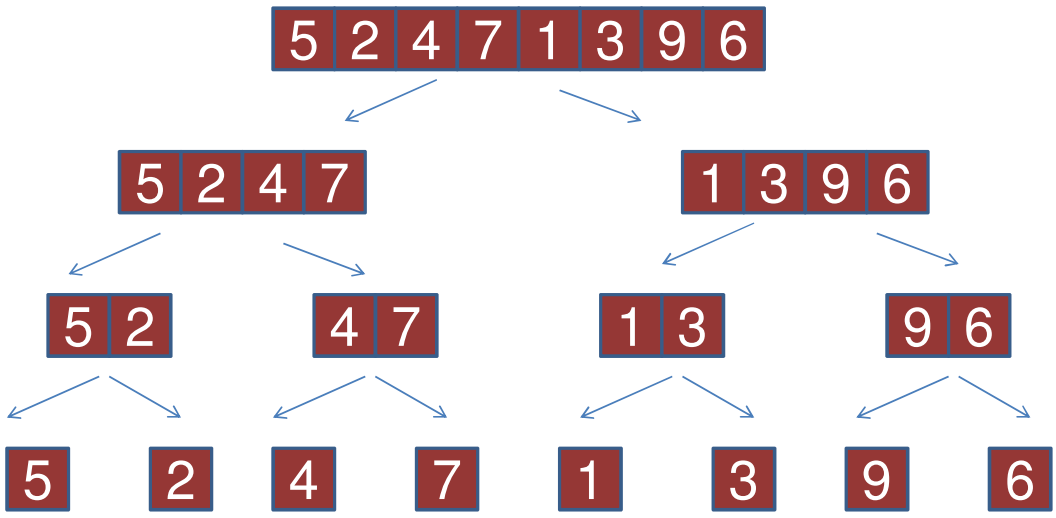
An Example: Factorial

```
int factorial(n)
  if n == 1,
      return 1
  else
      return (n × factorial(n-1)
end factorial
```

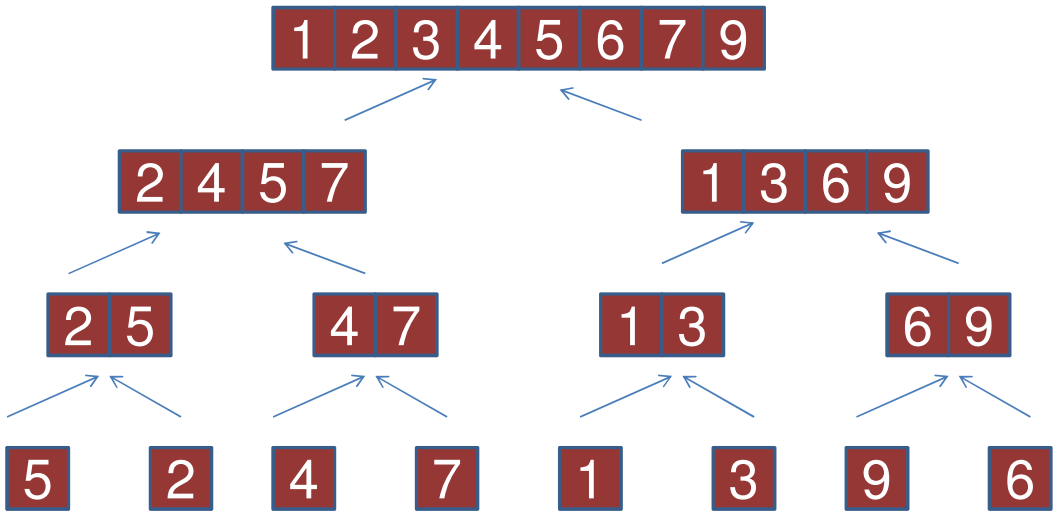
Divide and Conquer Approach

- Divide the problem into sub problems
- Conquer each sub problem autonomously
- Combine the solutions of the sub problems

Merge Sort



Merge Sort



Algorithm: Merge Sort

merge-sort(*A, p, r)*

if *p < r*

then *q* ← *⌊(p + r)/2⌋*

merge-sort (*A, p, q)*

merge-sort (*A, q + 1, r)*

merge (*A, p, q, r)*

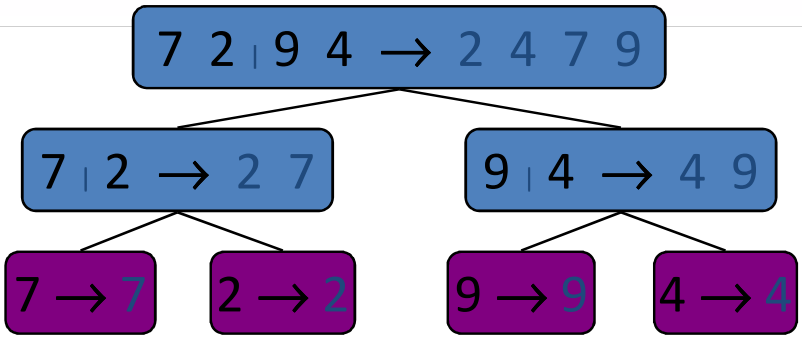
merge (*A, p, q, r)*

```
1   n1 ← q - p + 1
2   n2 ← r - q
3   create arrays L[n1] and R[n2]
4   for (i ← 1; i<= n1, i++)
5           L[i] ← A[p + i - 1]
6   for (j ← 1; i<= n2, j++)
7           R[j] ← A[q + j]
8    i ← 1
9    j ← 1
10  for (k ← p; k<= r ; k++) do
11          if L[i] ≤ R[j]
12                  then A[k] ← L[i]
13                  i ← i + 1
14          else A[k] ← R[j]
15                  j ← j + 1
```

Merge-Sort Tree

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

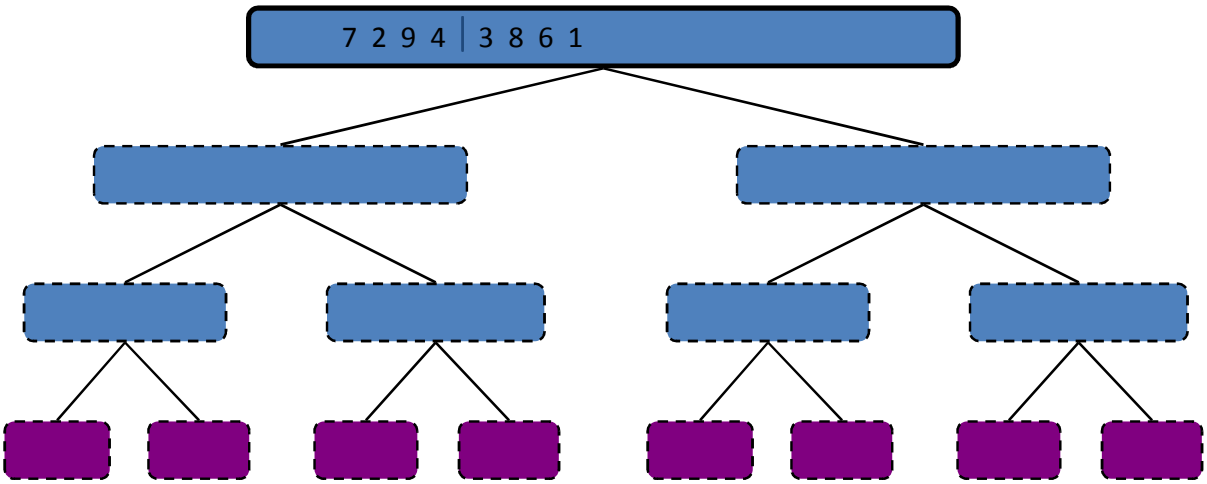
- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences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



Execution Example

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

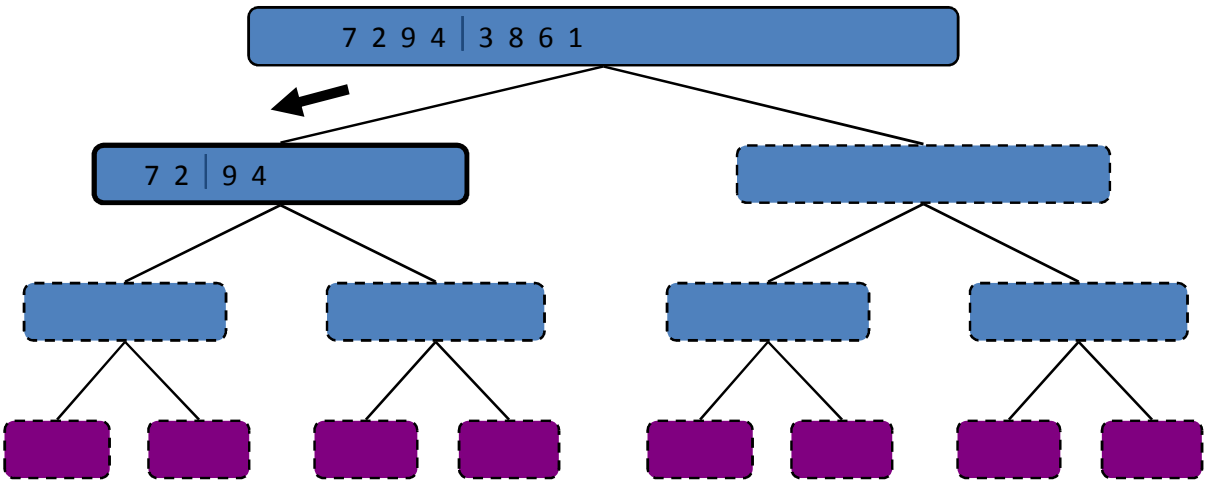
- Partition



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

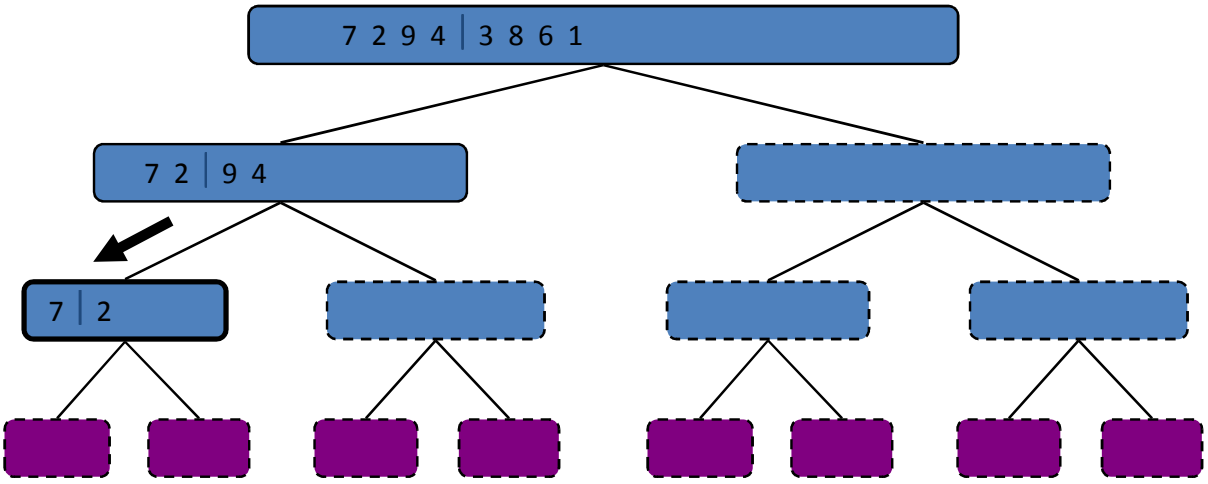
- Recursive call, partition



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

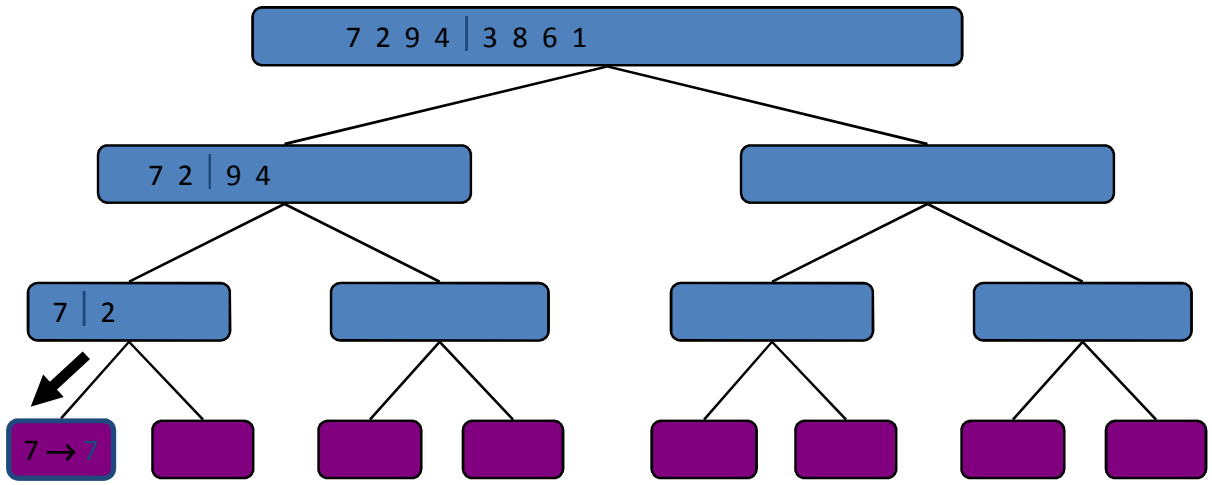
- Recursive call, partition



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

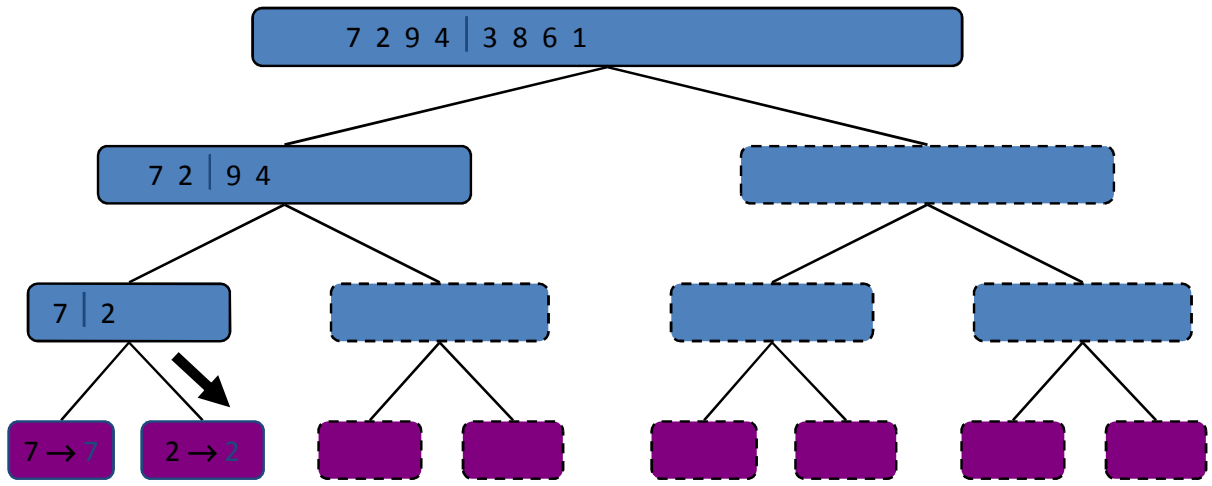
- Recursive call, base case



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

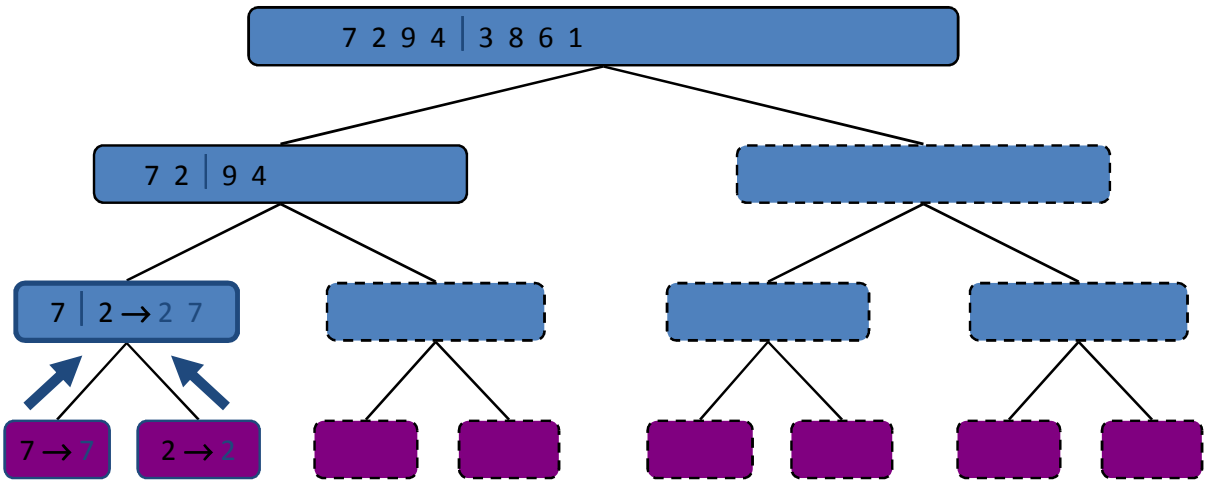
- Recursive call, base case



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

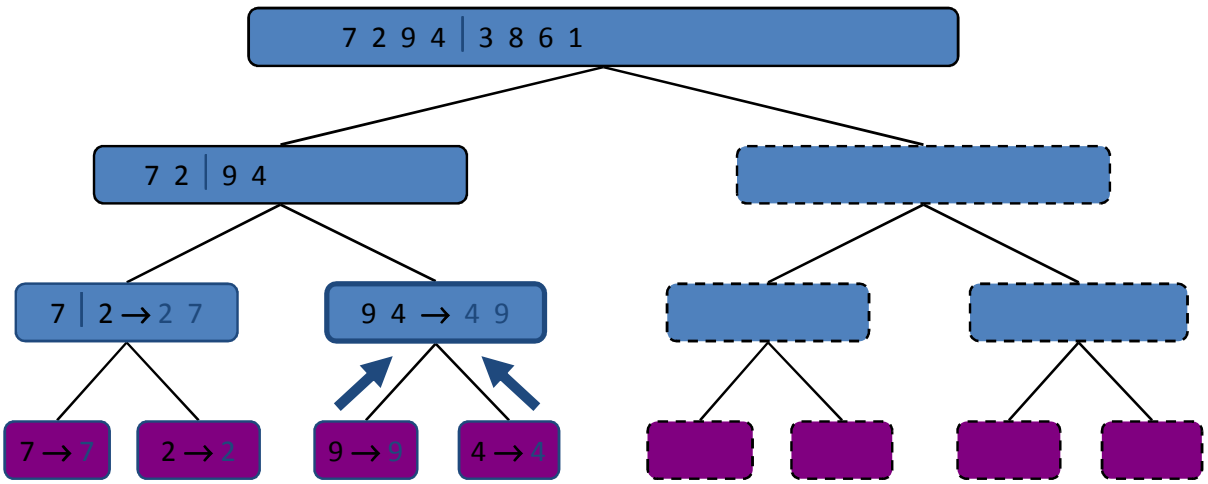
- Merge



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

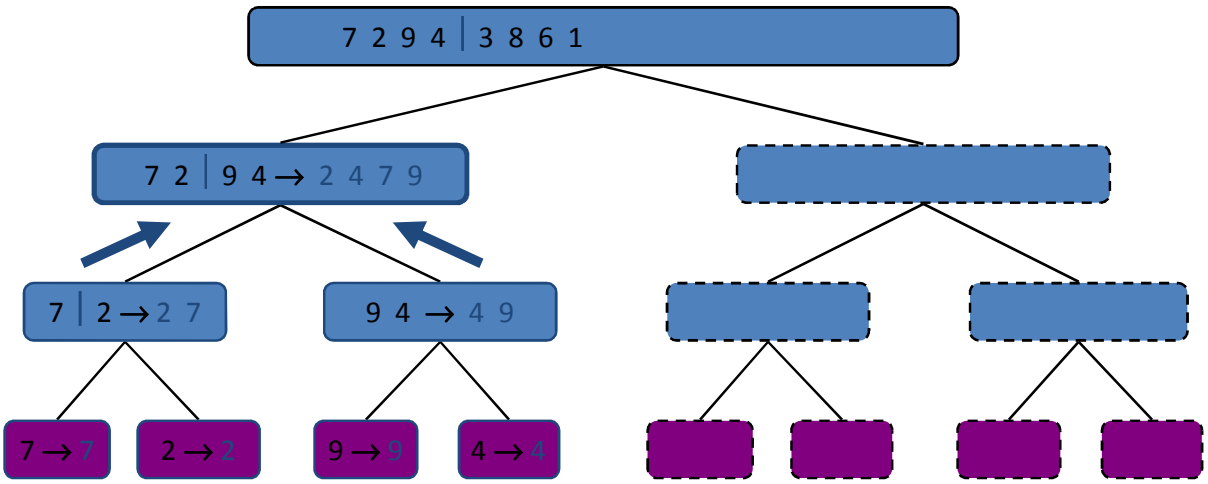
- Recursive call, …, base case, merge



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

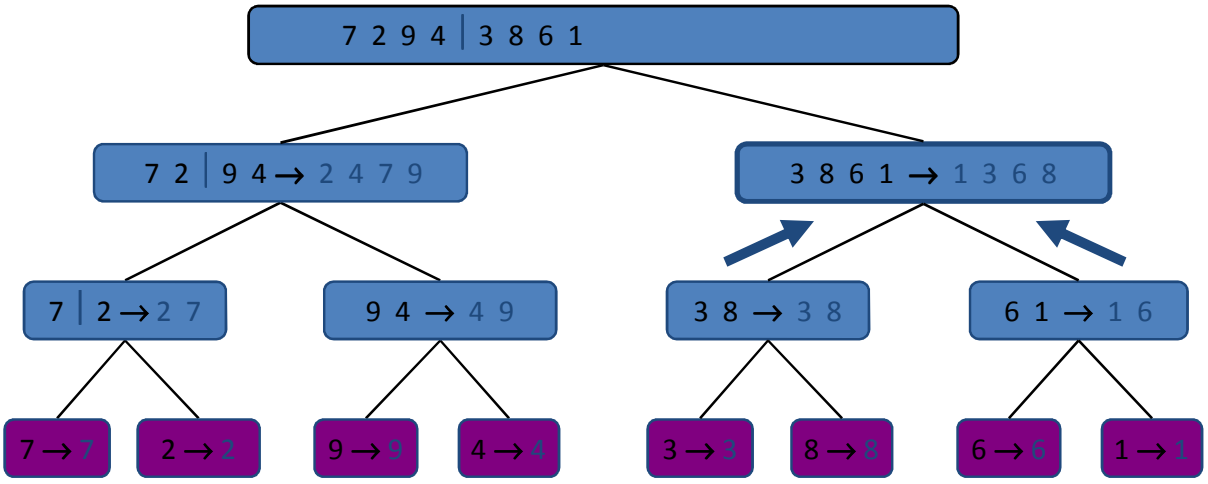
- Merge



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

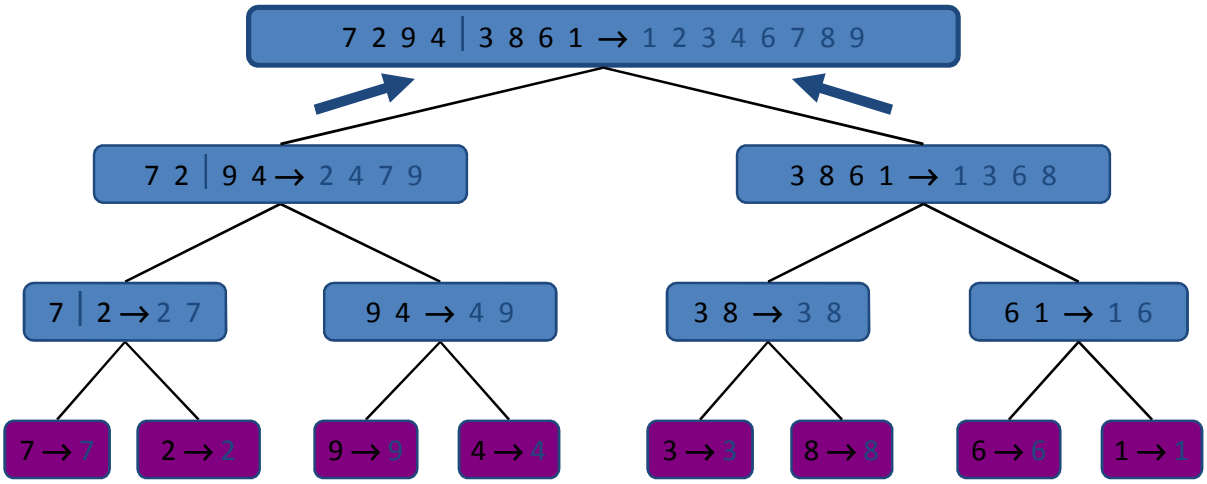
- Recursive call, …, merge, merge



Execution (cont.)

```
merge-sort(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
    merge-sort (A, p, q)
    merge-sort (A, q + 1, r)
    merge (A, p, q, r)
```

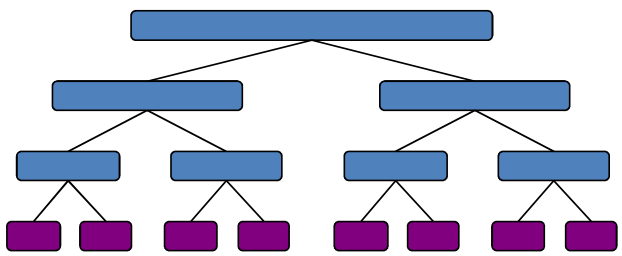
- Merge



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	$n/2^i$
…	…	…



Recursive Algorithms: Analysis

- Recurrence (or Recurrence Equation)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n <= c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- Parts
  - Solution for smallest subdivision (Terminating condition)
  - Subdivision relation (aT(n/b))
  - Division efforts (D(n))
  - Combining effort (C(n))

Merge Sort: Algorithmic Analysis

- Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{for } n > 1 \end{cases}$$

Solving Recurrence

- Substitution Method
- Recursion-tree Method
- Master Method

Binary Search

```
BinarySearchIterative(x, A) {
    left = 0;
    right = length(A) – 1;
    while (left <= right) {
            mid = (left + right) / 2
            if (A[mid] == x)
                    return mid;
            else if (A[mid] > value)
                    right = mid - 1
            else
                    left = mid + 1
    }
return (-1) // not found
}
```

```
BinarySearchRecursive(x, A,left,right) {
    If (left > right)
            return (-1) // not found
    else {
        mid = (left + right) / 2
        if (A[mid] == x)
            return mid;
        else if (A[mid] > value)
            return BinarySearchRecursive(x,A,left,mid-1);
        else
            return BinarySearchRecursive(x,A,mid+1,right);
    }
}
```

Recursion vs. iteration

- Recursion is inefficient
  - "function-call overhead“
- Recursion can be replaced by iteration
- Problems whose solutions are inherently recursive

Tail Recursion vs. Augmented Recursion

Tail Recursion

- Functions ending in a recursive call
- Does not build deferred operations
- Similar to iteration

```
gcd(x, y) {
if (y == 0)
        return x;
else
return gcd(y, x % y); }
```

Augmented Recursion

- builds up deferred operations
- Inefficient
- Iteration is preferred

```
int fact(int n) {
if (n == 1)
        return 1;
else
        return n * fact(n - 1); }
```

Summary

- Recursion is a tool to solve problems using “Divide and Conquer” approach
  - Divide the problem into sub problems,
  - Solving sub problems recursively, and
  - combining the solutions
- Tail Recursion is a preferred form of Recursion