Divide and Conquer (Revisiting Merge Sort and Quick sort)

Instructor: Dr. Tarunpreet Bhatia

Assistant Professor, CSED

Thapar Institute of Engineering and Technology

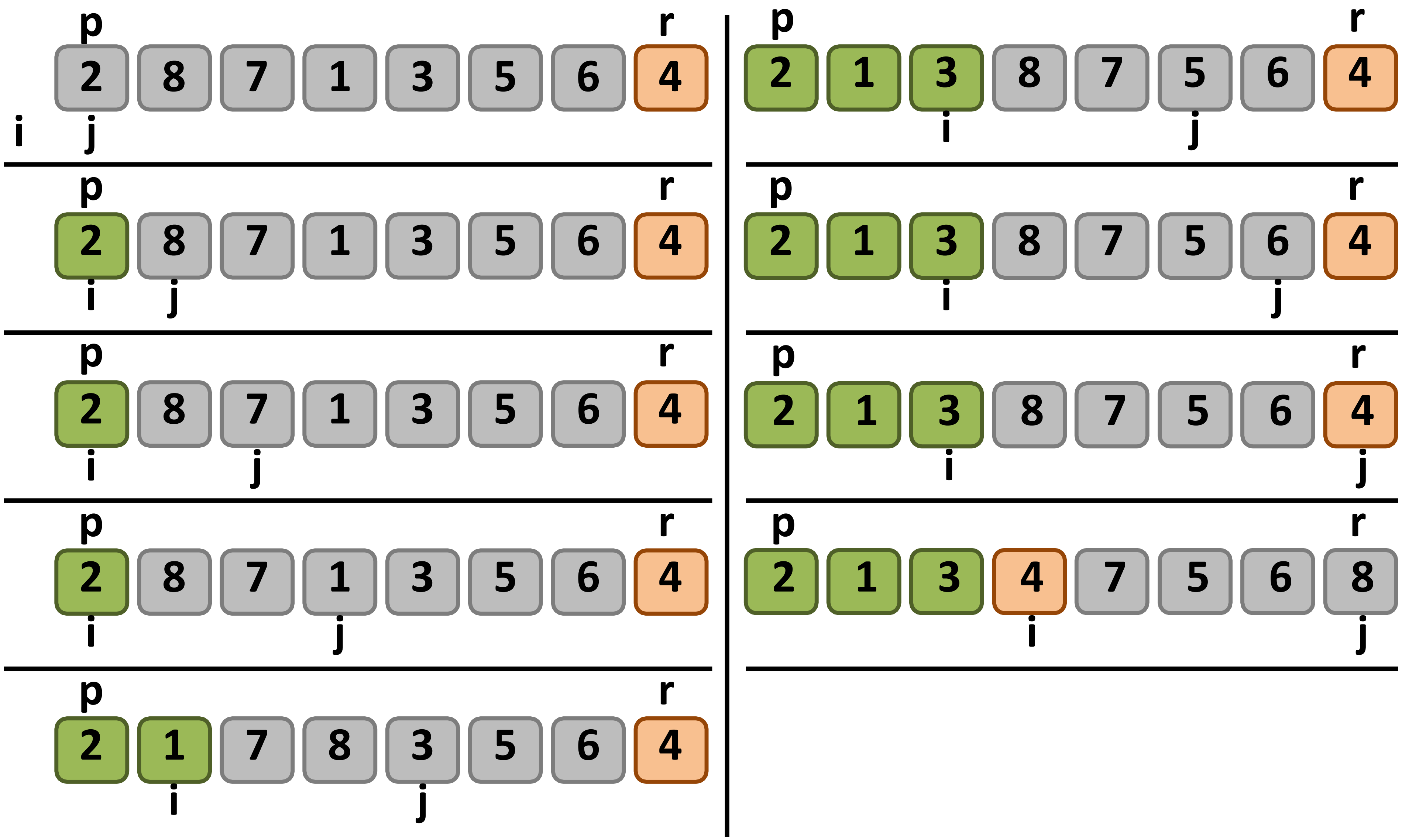
Quick Sort

- Divide and Conquer algorithm. In-place algorithm.
- Picks an element as pivot and partitions the given array around the picked pivot, such that
  - The pivot is placed at its correct position
  - All elements smaller than the pivot are placed before the pivot.
  - All elements greater than the pivot are placed after the pivot.
- Several ways to pick a pivot.
  - The first element.
  - The last element.
  - Any random element.
  - The median.

Algorithm


```
PARTITION(A, p, r)
x = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤ x
        i = i + 1
        Exchange A[i] with A[j]
Exchange A[i + 1] with A[r]
return i + 1
```

Partition Procedure



Algorithm

- QUICKSORT(A, p, r)

```
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q – 1)
    QUICKSORT(A, q + 1, r)
```

To sort an array A with n elements, the first call to QUICKSORT is made with $p = 0$ and $r = n – 1$.

```
PARTITION(A, p, r)
x = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤ x
        i = i + 1
        Exchange A[i] with A[j]
Exchange A[i + 1] with A[r]
return i + 1
```

Call sequence for QUICKSORT

2 8 7 1 3 5 6 4

- QUICKSORT (A, 0, 7) — Partition point at index 3 — 2 1 3 4 7 5 6 8
  - QUICKSORT (A, 0, 2) — Partition point at index 2 — 2 1 3 4 7 5 6 8
    - QUICKSORT (A, 0, 1) — Partition point at index 0 — 1 2 3 4 7 5 6 8
      - QUICKSORT (A, 0, -1)
      - QUICKSORT (A, 1, 1)
    - QUICKSORT (A, 3, 2)
  - QUICKSORT (A, 4, 7) — Partition point at index 7 — 1 2 3 4 7 5 6 8
    - QUICKSORT (A, 4, 6) — Partition point at index 5 — 1 2 3 4 5 6 7 8
      - QUICKSORT (A, 4, 4)
      - QUICKSORT (A, 6, 6)
    - QUICKSORT (A, 8, 7)

Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is perfectly balanced
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

Time Complexity – Quick Sort

- Best Time Complexity: **Ω** $(n\ logn)$
- Average Time Complexity: **Θ** $(n\ logn)$
- Worst Time Complexity: **O** (n^2)

Analyzing Quicksort

- In the worst case:

  $T(1) = \Theta(1)$

  $T(n) = T(n - 1) + \Theta(n)$

- Works out to

  $T(n) = \Theta(n^2)$

Analyzing Quicksort

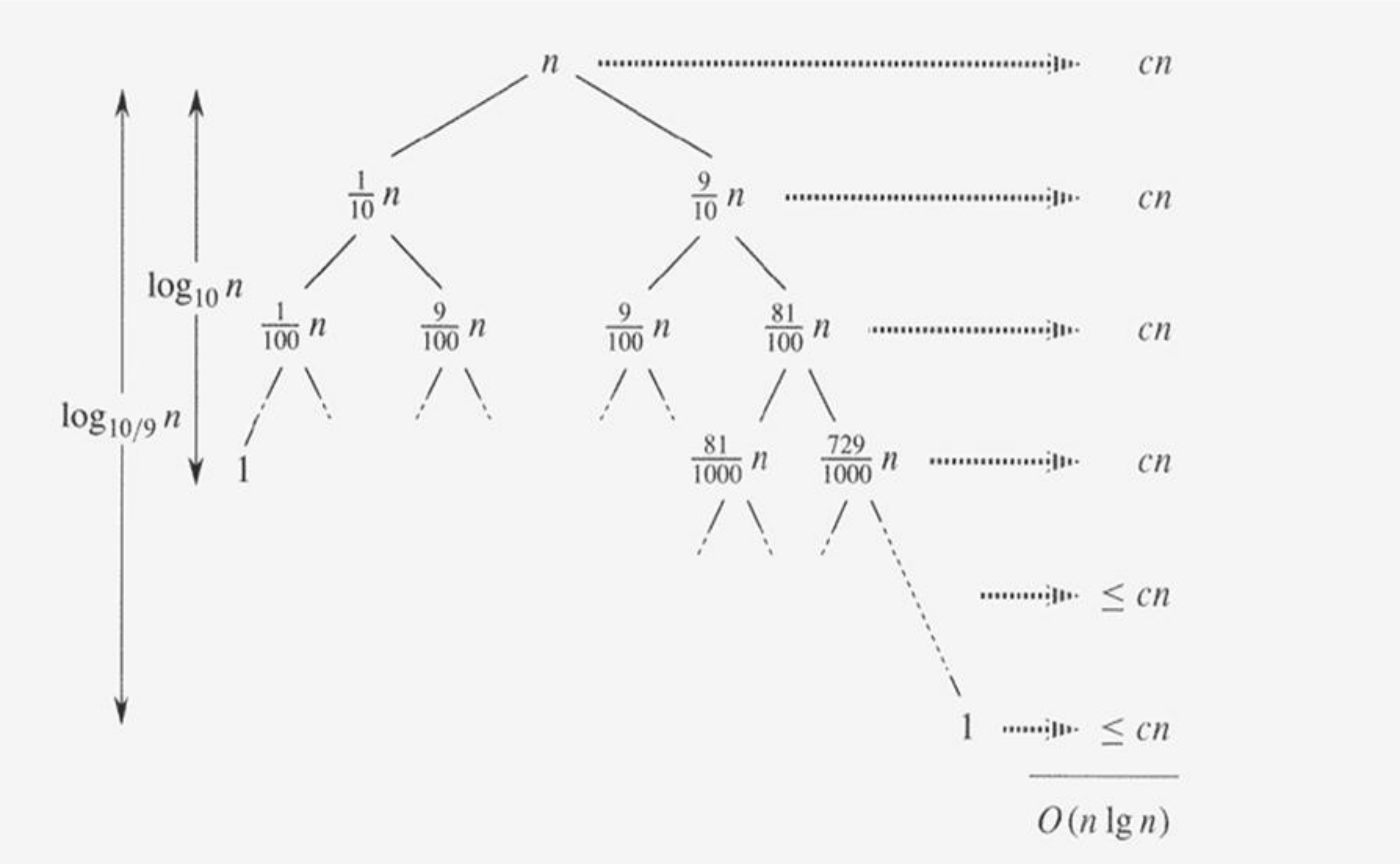
- In the best case:

  $T(n) = 2T(n/2) + \Theta(n)$

- What does this work out to?

  $T(n) = \Theta(n \lg n)$

Average case



Improving Quicksort

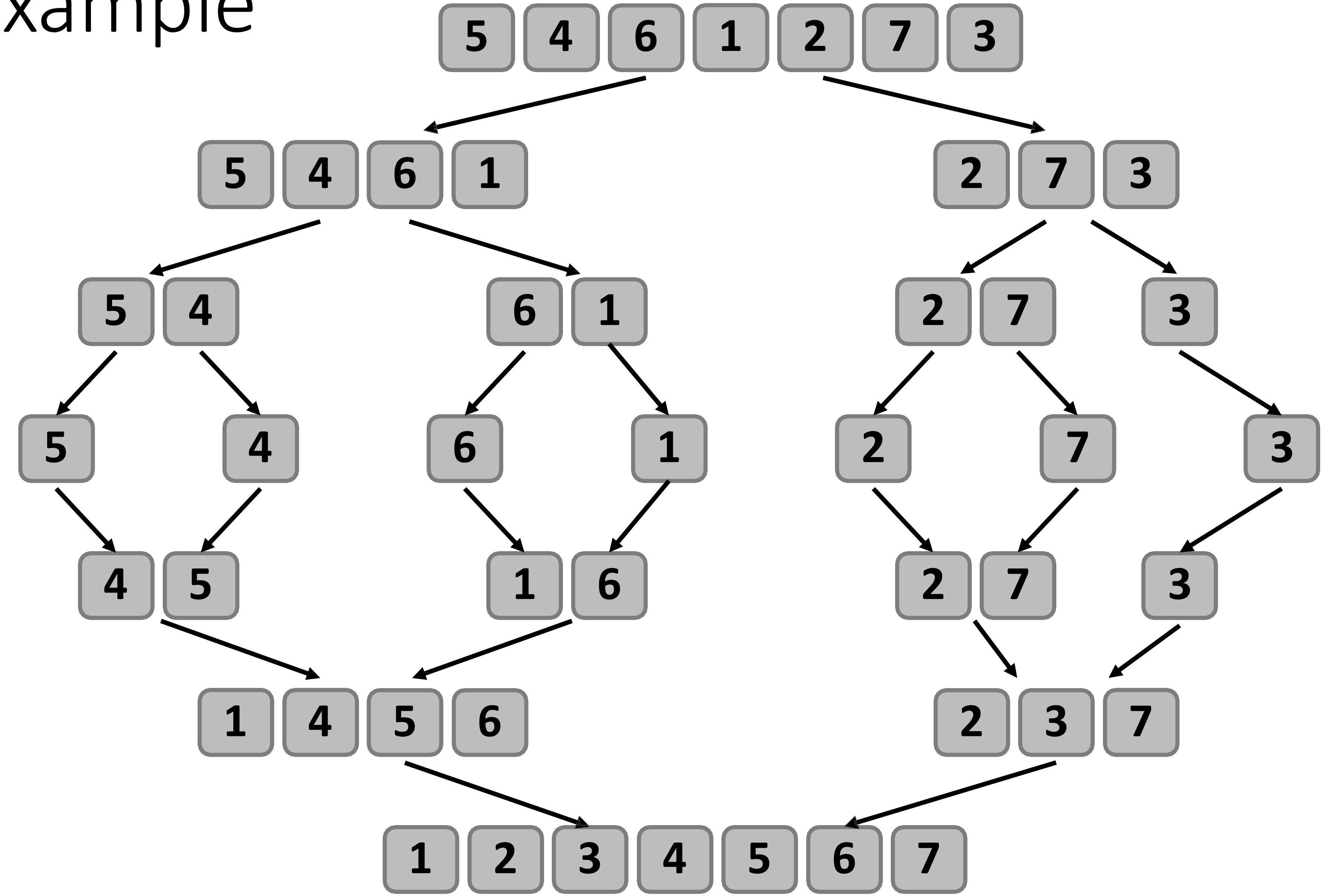
- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Two solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element*
- *How will these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Merge Sort

- Based on the divide-and-conquer paradigm.
- To sort an array A[p .. r], (initially p = 0 and r = n-1)

1. Divide Step
   – If a given array A has zero or one element, then return as it is already sorted.
   – Otherwise, split A[p…r] into two subarrays A[p…q] and A[q + 1... r], each containing about half of the elements of A[p…r]. That is, q is the halfway point of A[p...r].
1. Conquer Step
   – Recursively sort the two subarrays A[p…q] and A[q + 1…r].
2. Combine Step
   – Combine the elements back in A[p…r] by merging the two sorted subarrays A[p…q] and A[q + 1…r] into a sorted sequence.

Example



Merge Two Sorted Arrays

n1 - #Elements in L
n2 - #Elements in R

L: | 1 | 4 | 5 | 6 |
i

A: | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
k

R: | 2 | 3 | 7 |
j

```
i = 0, j = 0, and k = p.
while i < n1 and j < n2
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
        k++
while i < n1
        A[k] = L[i]
        i++
        k++
while j < n2
        A[k] = R[j]
        j++
        k++
```

Algorithm

- MERGE-SORT (A, p, r)

1. if p < r
2. q = FLOOR[(p + r)/2]
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q + 1, r)
5. MERGE (A, p, q, r)

- To sort an array A with n elements, the first call to MERGE-SORT is made with $p = 0$ and $r = n – 1$.

Contd…

- Algorithm MERGE (A, p, q, r)
- Input: Array A and indices p, q, r such that p ≤ q ≤ r. Subarrays A[p…q] and A[q + 1…r] are sorted.
- Output: The two subarrays are merged into a single sorted subarray in A[p .. r].

```
1.  n1 = q − p + 1
2.  n2 = r − q
3.  Create arrays L[n1] and R[n2]
4.  for i = 0 to n1 – 1
5.          L[i] = A[p + i]
6.  for j = 0 to n2 – 1
7.          R[j] = A[q + 1 + j ]
```

Contd…

```
8.  i = 0, j = 0, and k = p.
9.  while i < n1 and j < n2
10.         if L[i] ≤ R[j]
11.                 A[k] = L[i]
12.                 i = i + 1
13.         else
14.                 A[k] = R[j]
15.                 j = j + 1
16.         k++
17. while i < n1
18.         A[k] = L[i]
19.         i++
20.         k++
21. while j  < n2
22.         A[k] = R[j];
23.         j++;
24.         k++;
```

Call sequence for an array with size 8

- MERGE-SORT (A, 0, 7)
  - MERGE-SORT (A, 0, 3)
    - MERGE-SORT (A, 0, 1)
      - MERGE-SORT (A, 0, 0)
      - MERGE-SORT (A, 1, 1)
      - **MERGE (A, 0, 0, 1)**
    - MERGE-SORT (A, 2, 3)
      - MERGE-SORT (A, 2, 2)
      - MERGE-SORT (A, 3, 3)
      - **MERGE (A, 2, 2, 3)**
    - **MERGE (A, 0, 1, 3)**
  - MERGE-SORT (A, 4, 7)
    - MERGE-SORT (A, 4, 5)
      - MERGE-SORT (A, 4, 4)
      - MERGE-SORT (A, 5, 5)
      - **MERGE (A, 4, 4, 5)**
    - MERGE-SORT (A, 6, 7)
      - MERGE-SORT (A, 6, 6)
      - MERGE-SORT (A, 7, 7)
      - **MERGE (A, 6, 6, 7)**
    - **MERGE (A, 4, 5, 7)**
  - **MERGE (A, 0, 3, 7)**

5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	1	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	3	7	8
1	2	3	4	5	6	7	8

Time Complexity – Merge Sort

- Best Time Complexity: **Ω** $(n\ logn)$
- Average Time Complexity: **Θ** $(n\ logn)$
- Worst Time Complexity: **O** $(n\ logn)$

Analysis of Merge Sort

- Running time $T(n)$ of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

$\Rightarrow T(n) = \Theta(n \lg n)$

Recursion Tree – Example

- Running time of Merge Sort:

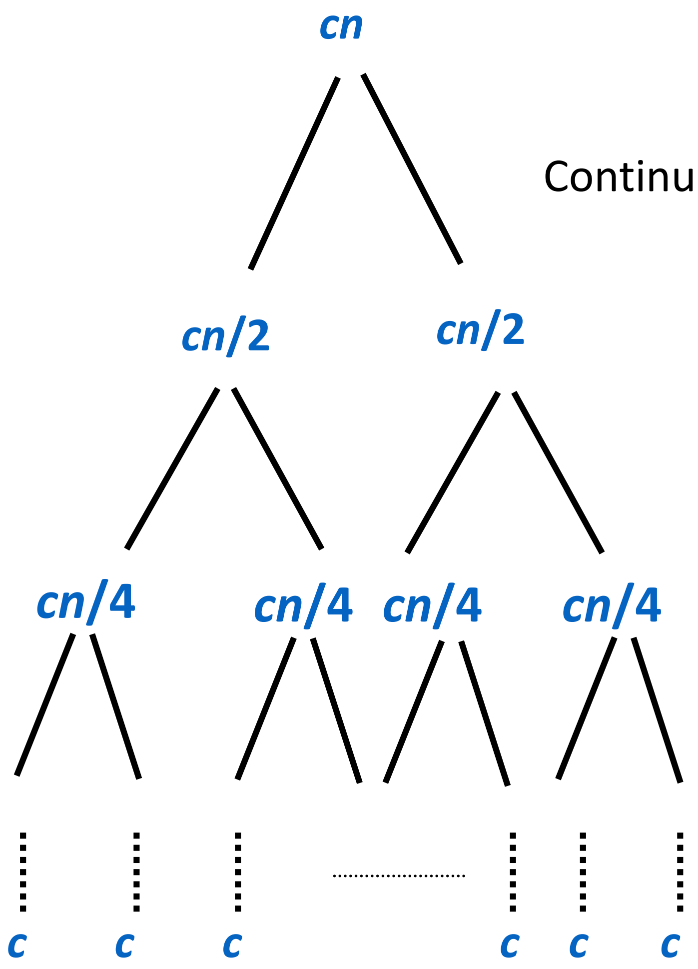
$$T(n) = \Theta(1) \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

- Rewrite the recurrence as

$$T(n) = \boldsymbol{c} \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \boldsymbol{cn} \qquad \text{if } n > 1$$

$\boldsymbol{c > 0}$: Running time for the base case and time per array element for the divide and combine steps.

Recursion Tree for Merge Sort



Continue expanding until the problem size reduces to 1.

- Each level has total cost ***cn***.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves $\Rightarrow$ *cost per level remains the same*.
- There are $\lg n + 1$ levels, height is $\lg n$. (Assuming n is a power of 2.)
- Total cost = sum of costs at each level = $(\lg n + 1)cn = cn\lg n + cn = \Theta(n \lg n)$.