Merge Sort

Algorithm

Design paradigms

- **Paradigm**
  - "In science and philosophy, a paradigm is a distinct set of concepts or thought patterns, including theories, research methods, postulates, and standards for what constitutes legitimate contributions to a field"- Wikipedia

Types of Design Paradigms

- Incremental Approach
- Greedy approach
- Dynamic Programming
- Divide and Conquer

Incremental approach

- Example: Insertion sort
  - In the so far sorted subarray, insert a new single element into its proper place, resulting in the new sorted subarray
  - Example:

$$[\quad \cdots\cdot \quad] \uparrow [\quad \cdots\cdots\cdots \quad]$$

A[1 .. j-1] *A [j .. n]*

Key = A [j]

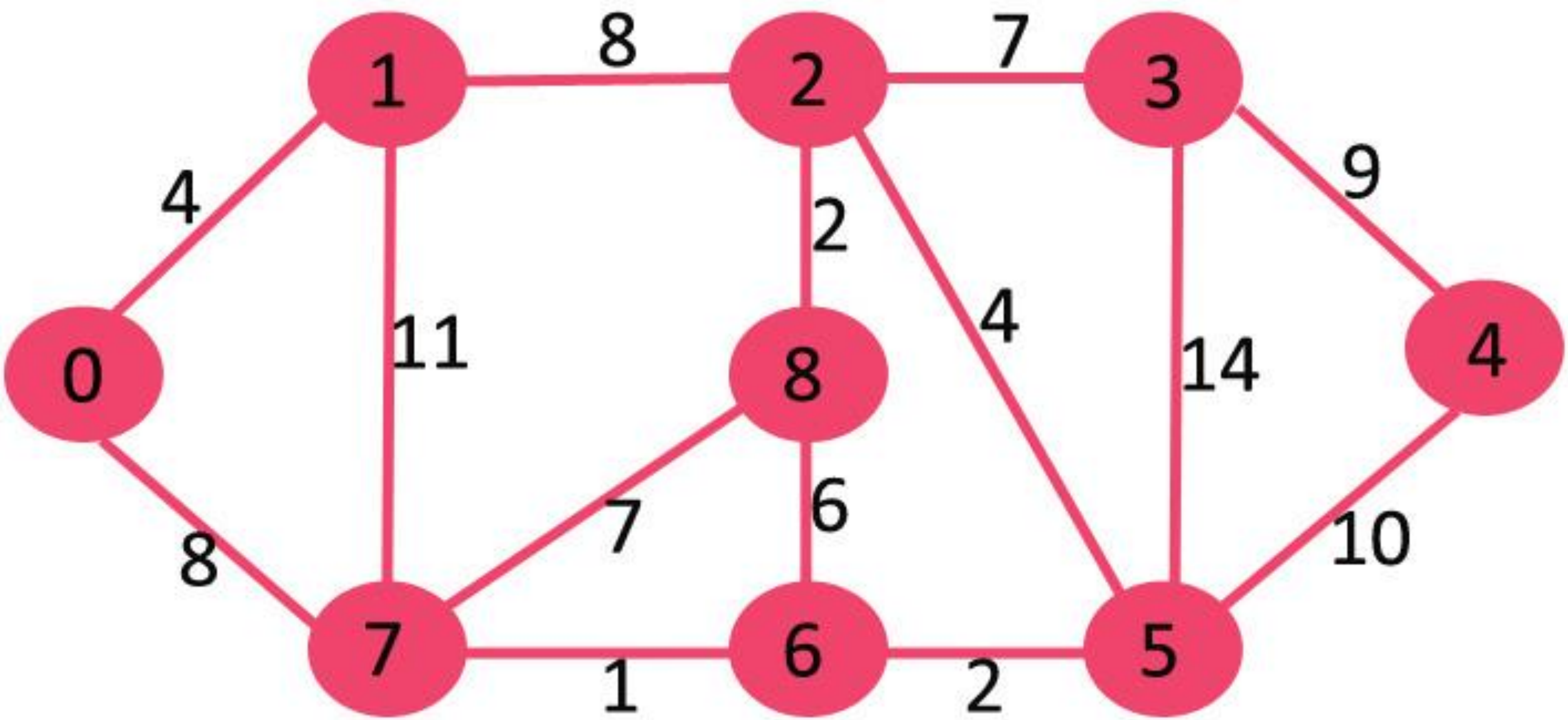
Types of Design Paradigms

- Incremental Approach
- Greedy approach
- Dynamic Programming
- Divide and Conquer

Greedy approach

- *builds up a solution piece by piece*
- *always choosing the next piece that offers the most obvious and immediate benefit*
- *choosing locally optimal also leads to a global solution in general*
- Example: Dijkstra’s shortest path algorithm

Dijkstra’s shortest path algorithm



Motivation for Divide and Conquer

Our life is frittered away by detail. Simplify, simplify.

— Henry David Thoreau

The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

Nothing is particularly hard if you divide it into small jobs.

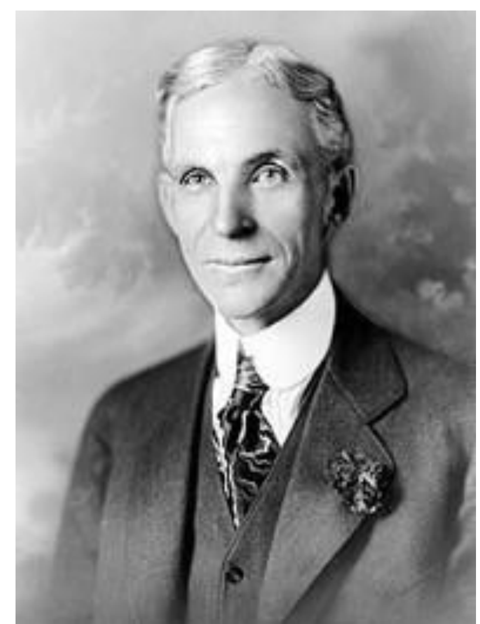
— Henry Ford

Henry Ford (July 30, 1863 - April 7, 1947) was an American captain of industry and a business magnate.

Founder of the Ford Motor company

Sponsor of the development of the ***assembly line technique of mass production***.

Breaks the manufacture of a good into steps that are completed in a pre-defined sequence



Henry Ford’s Assembly line

Divide and Conquer

- **Three crucial steps**
  - **Divide** the problem into smaller sub problems
  - **Conquer** the smaller subproblems recursively
  - **Combine** solutions of the subproblems to get the solution of the original problem

Divide and conquer - First step

- **Divide/Break** the problem into smaller sub problems
  - For example, Problem P is divided into subproblems P1 and P2
  - Also, P1 and P2 resemble the original problem and their input size is small

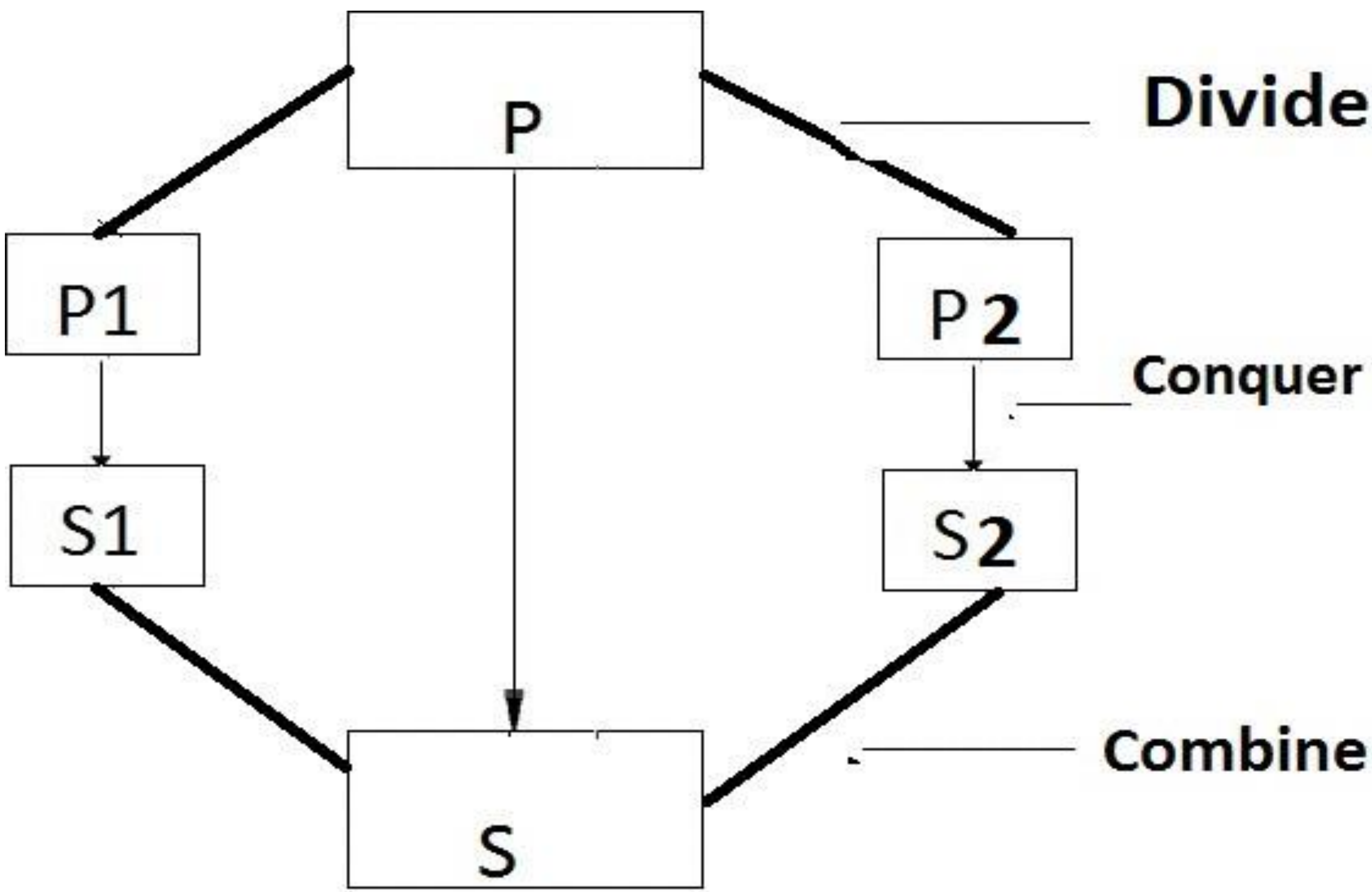
Divide and conquer - Second step

- **Conquer/Solve** the smaller subproblems recursively. If the input size of the sub problem is very small, solve them directly
  - P1 is solved to give solution S1, P2 is solved to give solution S2

Divide and conquer - Third step

- Merge/Combine these solutions to create a solution to the original problem
  - S1 and S2 are combined to give the solution S for the original problem P

Pictorial Representation: Divide and Conquer



Divide and Conquer (D & C)

- Most of the algorithms designed using D & C are recursive in nature
- **Recursive algorithms:** Call themselves recursively to solve the closely related subproblems
- **Examples**
  - Towers of Hanoi
  - Binary search
  - Merge Sort
  - Quick sort

Towers of Hanoi

Merge Sort

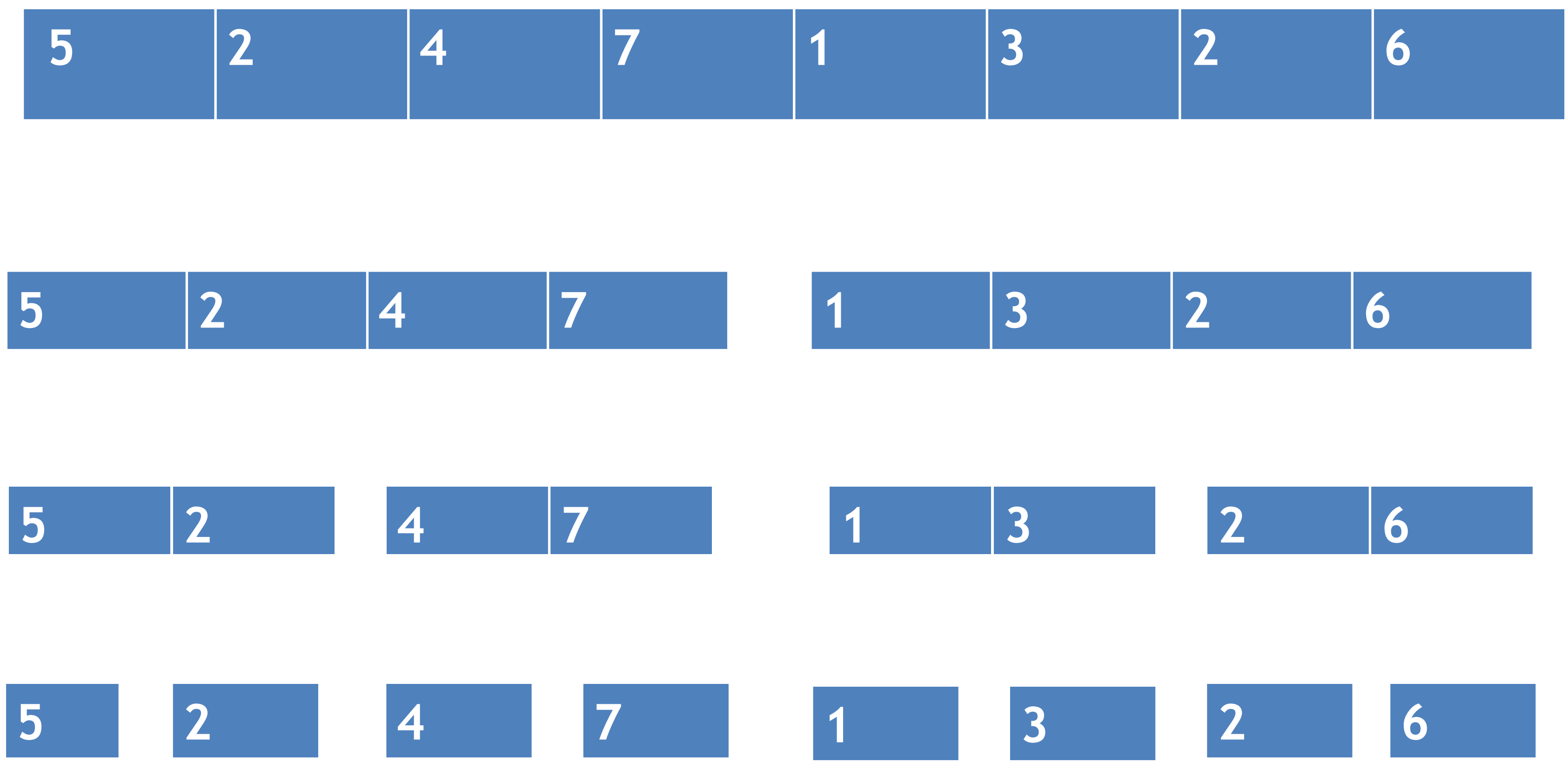
- Follows D & C paradigm

- Divide: Divide the n-element array into two subarrays of size n/2
- Conquer: Sort the two subarrays recursively
- Combine: Merge the two sorted subarrays to produce the sorted array

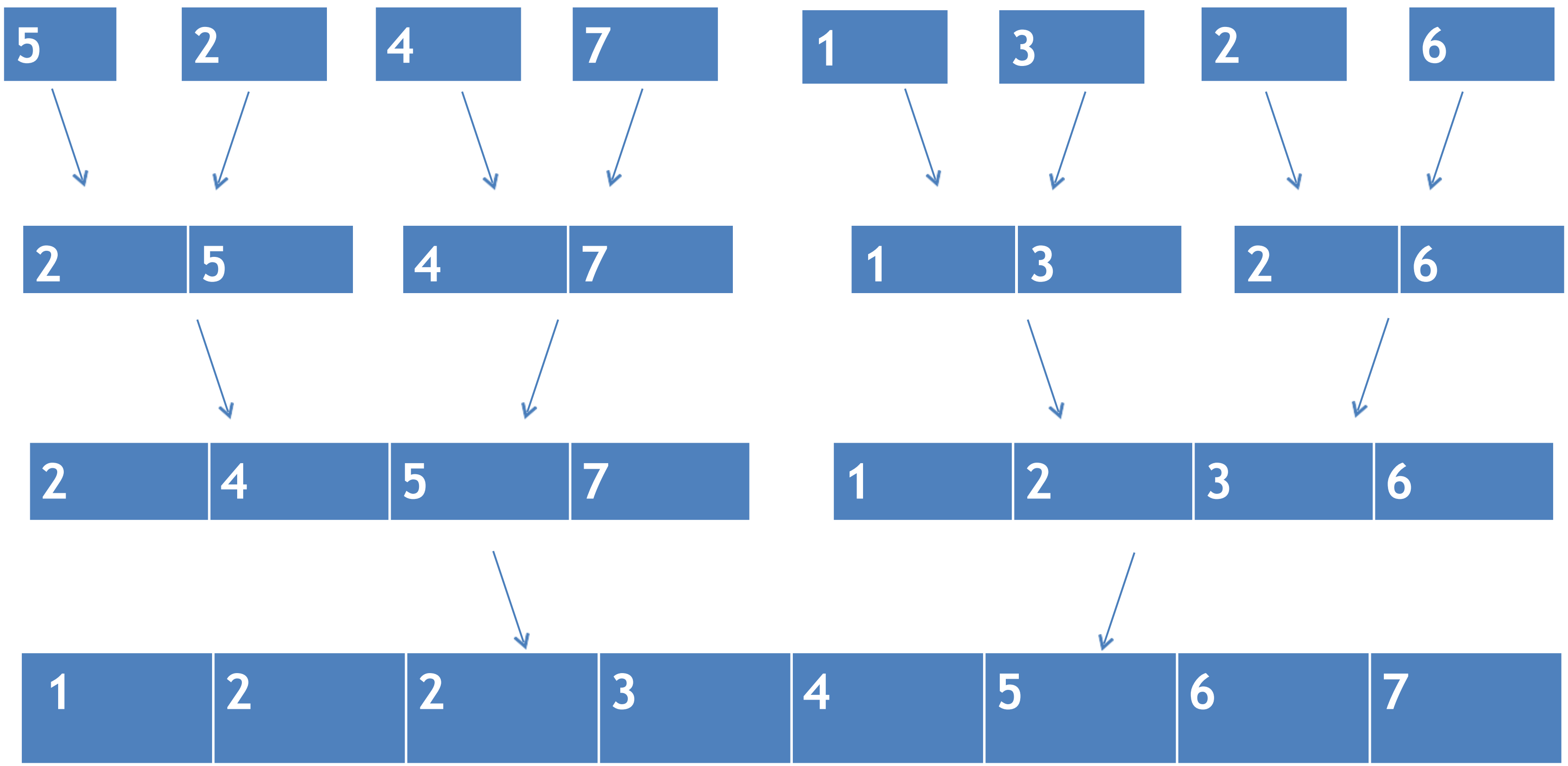
Merge Sort - Example

The operation of merge sort on the array

A= {5, 2, 4, 7, 1, 3, 2, 6}



Merging of sorted subarrays



Merge Sort – Recursive Algorithm

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Ref: CLRS Book

MERGE-SORT

- If p >= r , the subarray has at most one element and is therefore already sorted.

- Otherwise, the divide step, computes an index q that partitions A[p...r] into two subarrays A[p…q] and A[q+1…r] containing (n/2) elements

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Function call:

MERGE-SORT(A, 1, A.length)

Merge sort – Recursive algorithm

- **Base case:**
  - When the size of the subproblem is 1, we don't need to do any further
  - Its already sorted
- **Key operation:** Merging of two sorted arrays in the combine step
- Merge is done by calling another function Merge (A,p,q,r)

How to Merge?

Merge function

Merge is done by calling another function **Merge (A,p,q,r)**

A- Array, p,q,r are indices such that p <= q < r

Assumption: A[p…q] and A[q+1 … r] are in sorted order

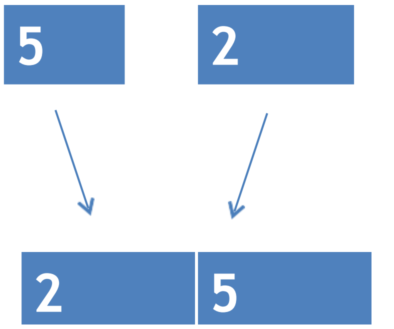
Input: Array A, indices p, q, and r

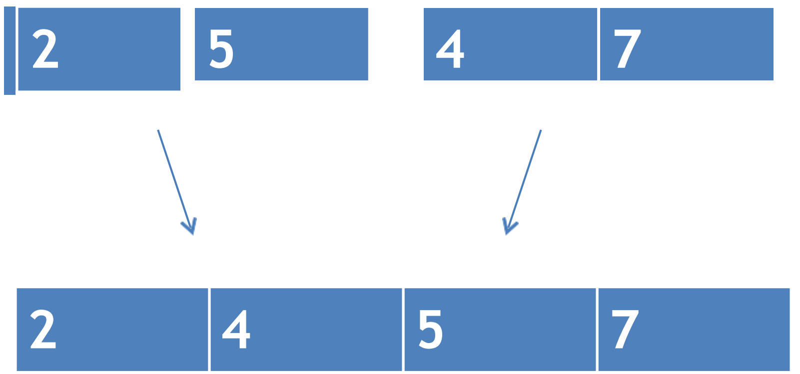
Output: Merges A[p…q] and A[q+1 … r] and produce a single sorted subarray A[p…r]

Merge function

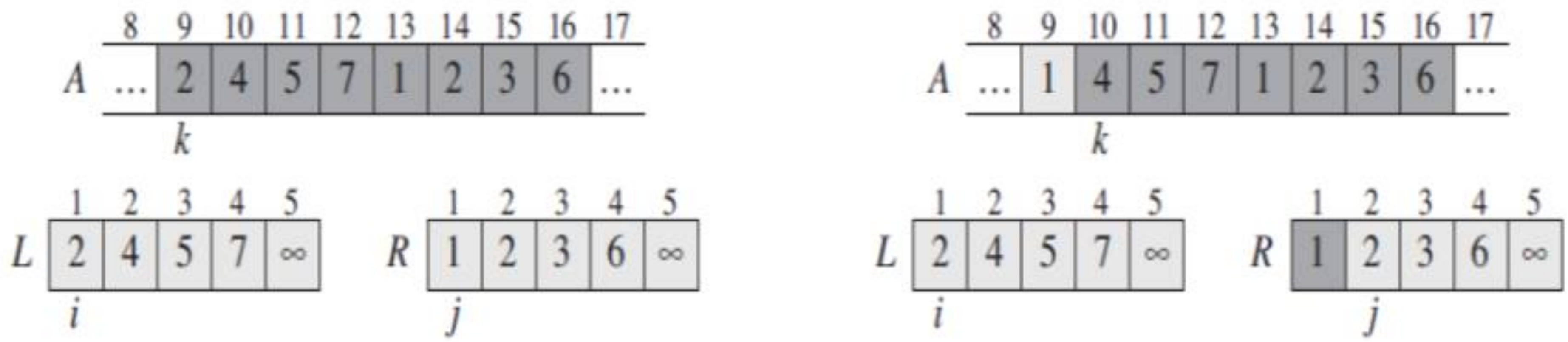
```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Sentinel - a special value





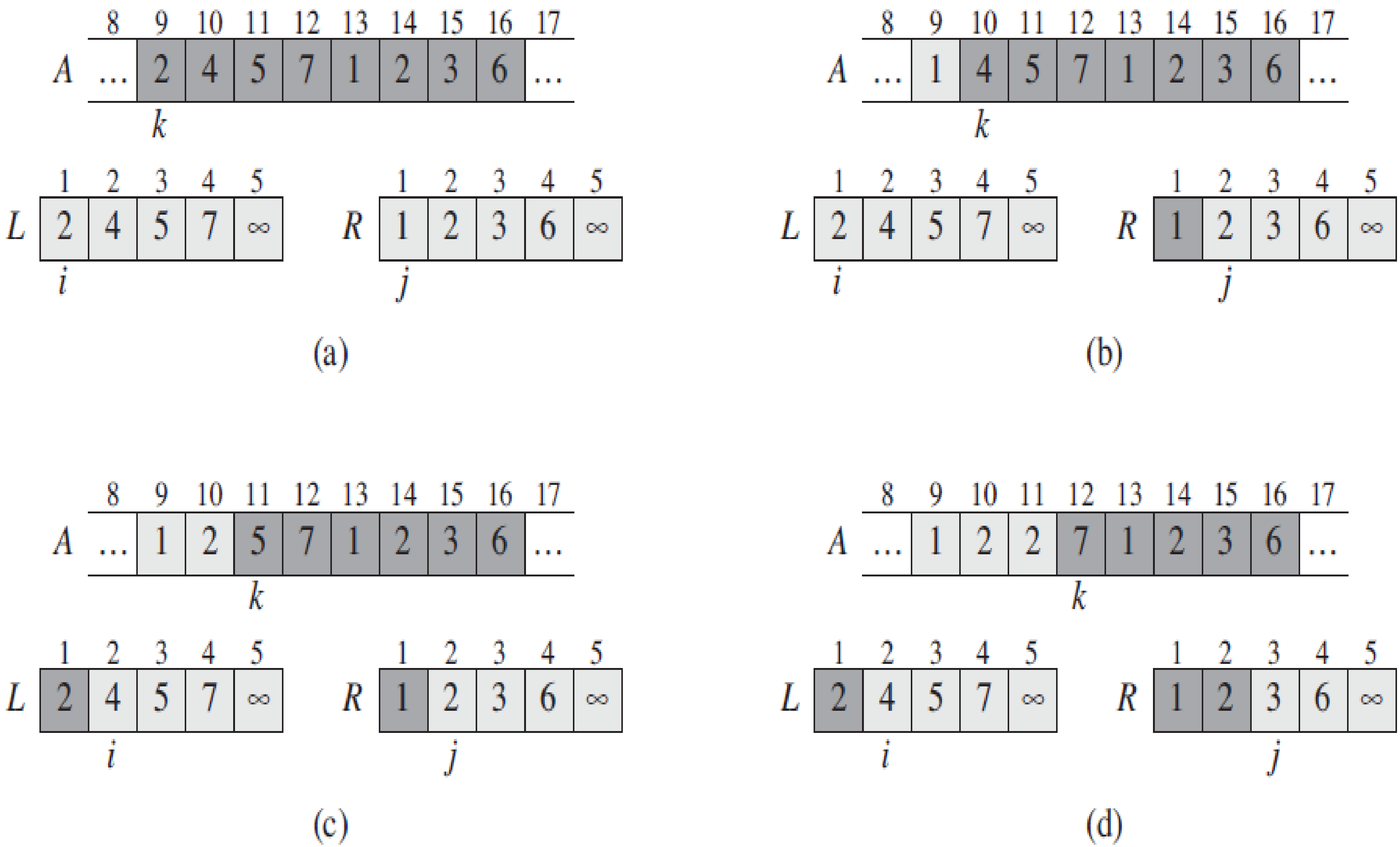
Ref: CLRS Book



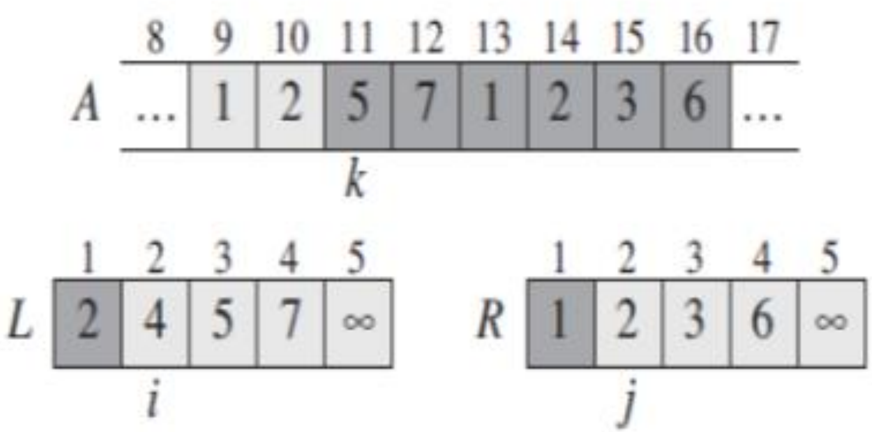
(a) (b)

```
MERGE(A, p, q, r)
n₁ = q − p + 1
n₂ = r − q
let L[1 .. n₁ + 1] and R[1 .. n₂ + 1] be new arrays
for i = 1 to n₁
      L[i] = A[p + i − 1]
for j = 1 to n₂
      R[j] = A[q + j]
L[n₁ + 1] = ∞
R[n₂ + 1] = ∞
```

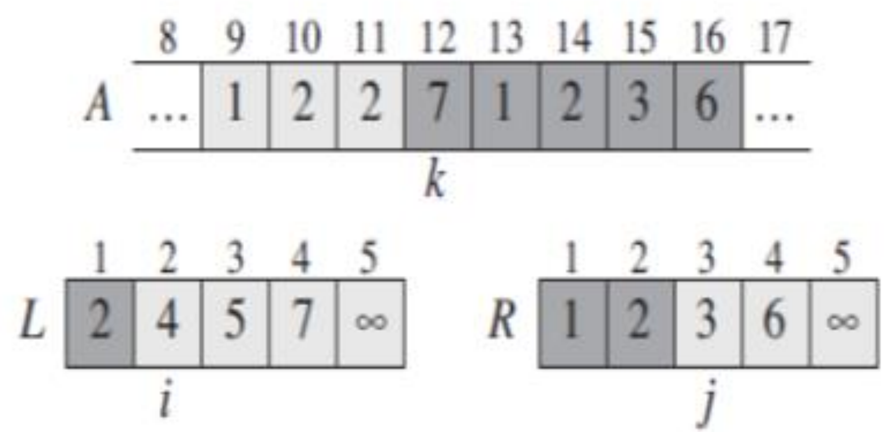
Working of Merge function



```
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



(c)



(d)

Merge :Running time

- MERGE procedure runs in Θ(n)-time, n=r-p+1
- Line 1-3 and 8-11 takes constant time
- Line 4-7 take $\Theta(n_1 + n_2)$-time
- Line 12-17 : n iterations of the for loop
  - Each statement within for loop takes constant time

```
MERGE(A, p, q, r)
1  n1 = q - p + 1
2  n2 = r - q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i - 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

Analysis of D & C algorithms

- Running time using recurrence equation or recurrence
- Recurrence describes the overall running time on a problem of size n in terms of the running time on smaller inputs
- Use mathematical tools to solve recurrence

Merge Sort – Recursive Algorithm

```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

Ref: CLRS Book

Recurrence for D & C algorithms

- Suppose the division of the problem yields **a** sub problems, each of which is **1/b the size** of the original.
- **T (n /b)-time** to solve one sub problem of **size n/b**
- **aT(n/b)- time** to solve **a** of them
- **D(n)-time** to divide the problem into sub problems and **C(n)-time** to combine the solutions to the sub problems into the solution to the original problem

Recurrence:

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

Merge Sort – Recursive Algorithm

MERGE-SORT(A, p, r)

```
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Ref: CLRS Book

Analysis of Merge sort

- Recurrence-based analysis is simplified if we assume that original problem size is a power of 2.
- Each divide step then yields two subsequences of size exactly n/2.
- Let T(n) be the worst-case running time of merge sort on n numbers

Analysis of Merge sort

- Merge sort on just one element takes constant time.
- When $n > 1$ elements, break down the running time as follows.
- **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n)=\Theta(1)$
- **Conquer:** Solve two sub problems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine:** MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n)= \Theta(n)$

Analysis of Merge sort

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

We can solve the above recurrence using **Master's theorem**.

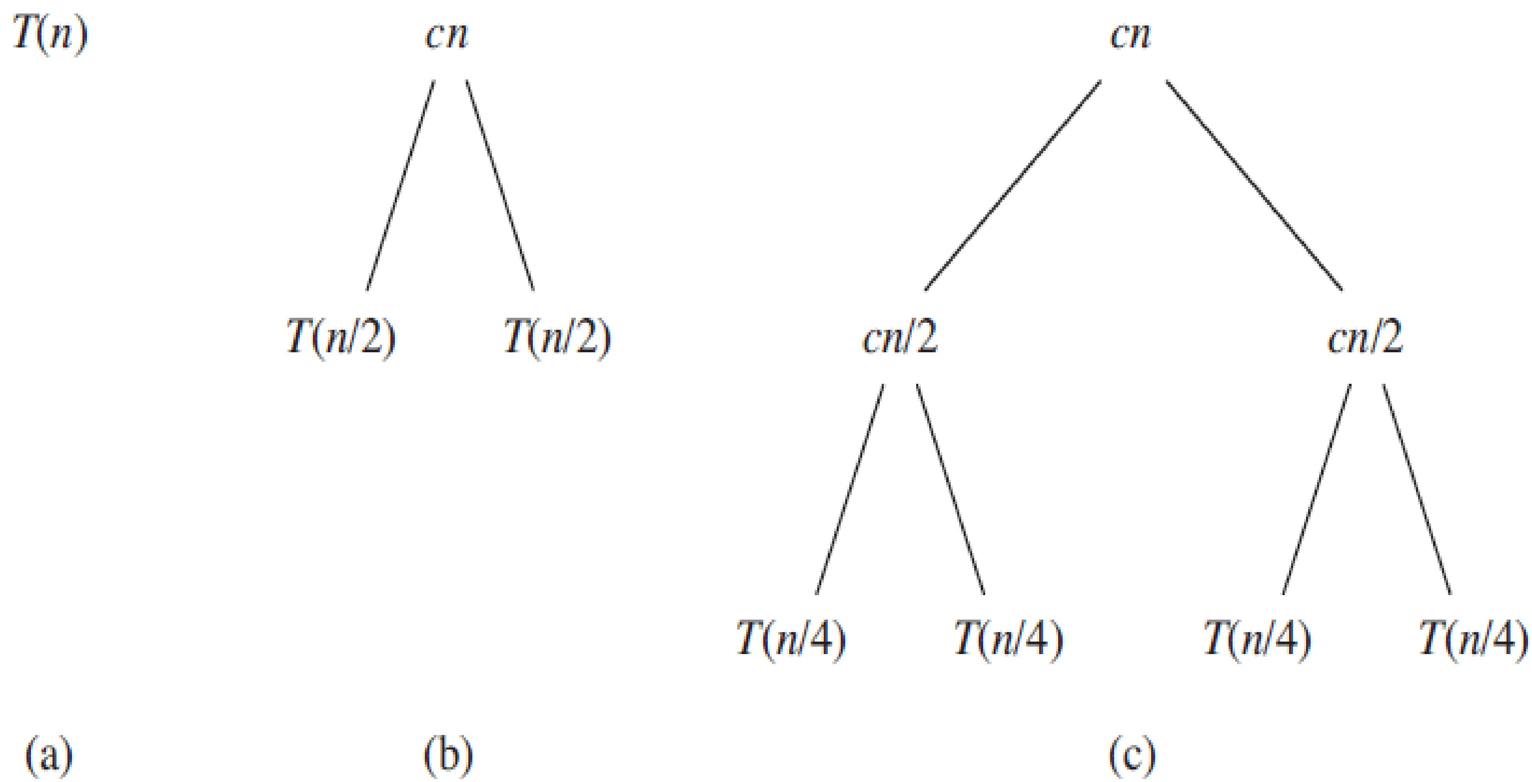
Now, we rewrite the recurrence as follows to solve it using **recursion tree** technique.

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

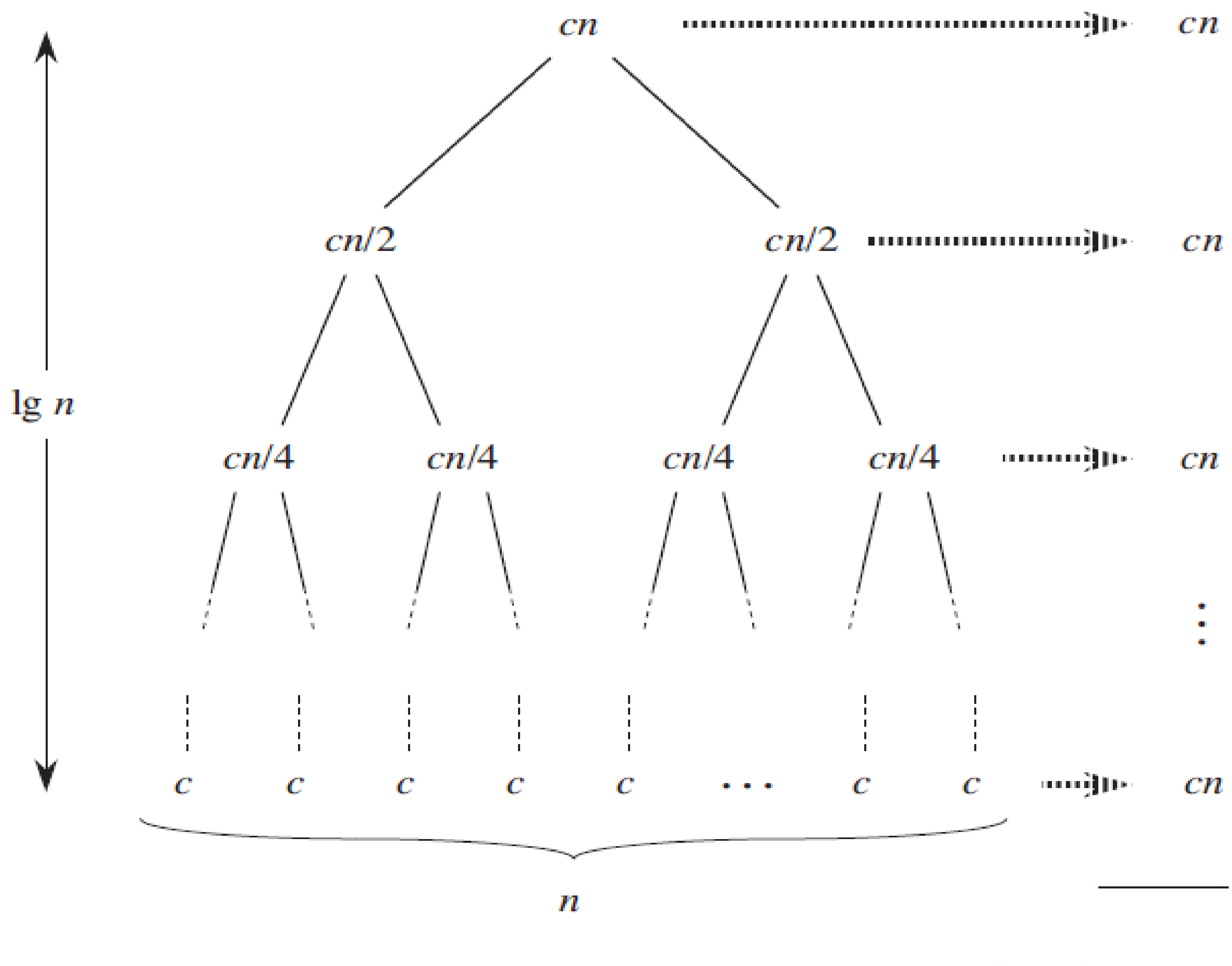
Constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps

Recursion tree

- Assume that **n is an exact power of 2**.
- The **cn** term is the root (the cost incurred at the top level of recursion)
- Two subtrees of the root are the **two smaller recurrences T (n/2)**



Continue expanding each node in the tree as determined by the recurrence, until the **problem sizes get down to 1, each with a cost of c.**



(d)

- **Total cost of the recursion tree = Cost of each level * total number of levels**

Costs across each level of the tree:

- The top level (root) has total cost **cn**
- Next level down has total cost

  $c(n/2)+ c(n/2)= cn$
- Level after that has total cost

  $c(n/4)+ c(n/4)+ c(n/4)+ c(n/4) = cn$, and so on
- **Level i has 2^i nodes**, each contributing a cost of **$c(n/2^i)$**, total cost **$2^i * c(n/2^i) = cn$**
- **Bottom level has n nodes**, each contributing a cost of **c**, for a **total cost of cn**

Compute the total cost

- Cost of each level * height of the tree

= cn * (lg n)

= cn lg n

= Θ (n lg n)

Hence, Merge sort running time is Θ (n lg n).

Correctness of Merge

Correctness of Merge

```
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

- **Loop invariant:** At the start of each iteration of the **for** loop of lines 12–17, the subarray A[p … k-1] contains the k - p smallest elements of L[1.. n_1 + 1] and R[1.. n_2 + 1], in sorted order.
- Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Maintaining the loop invariant

- Show that loop invariant holds **prior to the first iteration** of the for loop of lines 12–17
- **Each iteration of the loop** maintains the invariant
- Show correctness when **the loop terminates.**

```
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Initialization

```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

- Prior to the first iteration of the loop, we have k = p
- Subarray A[p … k- 1] is empty.
- This empty subarray contains the k – p = 0 smallest elements of L and R, and since i = j = 1
- Both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Maintenance: Each iteration maintains the loop invariant

```
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

- First, **suppose that L[i] <= R[j]**
- L[i] is the smallest element not yet copied back into A.
- Because A[p … k- 1] contains k - p smallest elements, after line 14 copies L[i] into A[k]
- Subarray A[p … k] will contain k-p+1 smallest elements
- Incrementing k (in the **for** loop) and i (in line 15) reestablishes the loop invariant for the next iteration
- **Suppose if L[i] > R[j],** lines 16 – 17 perform appropriate action to maintain loop invariant

Termination

- At termination, $k = r + 1$. By the loop invariant, the subarray $A[p \ldots k-1]$, which is $A[p \ldots r]$, contains $k - p = r - p + 1$ smallest elements of $L[1.. n_1 + 1]$ and $R[1.. n_2 + 1]$, in sorted order.
- The arrays L and R together contain

  $$n_1 + n_2 + 2 = r - p + 3 \text{ elements.}$$
- All but the two largest have been copied back into A, and these two largest elements are the sentinels.

Conclusion

- What is the best case, worst case and average case input of Merge sort algorithm?
- What about Bubble sort algorithm?
- Merge sort’s worst case running time is much less than that of insertion sort
- Divide-and-conquer algorithms running times are easily determined by solving recurrence relations

Thank You