Data Structures and Algorithm Analysis

15

Dr. Syed Asim Jalal
Department of Computer Science
University of Peshawar

In this Lecture

- Recurrences
- Divide and Conquer Approach
  - Merge Sort
  - Merge Sort Analysis

Recurrences

- **A recurrence is an equation or inequality that describes itself in terms of its values on smaller inputs.**
- Or a recurrence is a function that is defined in terms of
  1. one or more base cases, *(stopping conditions)*
  2. itself with smaller arguments.
- We get recurrences from recursive algorithms.
- Recursive algorithms call itself again an again until some Base Case is reached.

How to do Analysis of Recursive Algorithms?

- From recursive algorithm we first obtain a recurrence relationship and then
- From the relation we find its solution or equations using one of the **Recurrence Solution methods**

- For example, for the following Recurrence Relation

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

- If we solve this recurrence, we will get the following running time.

$$T(n) = n$$

Some other examples of recurrence relations and their solutions.

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

Solution: $T(n) = n \lg n + n$.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 \end{cases}$$

Solution: $T(n) = \Theta(n \lg n)$.

Q. How we get the solutions?
A. By using one of the methods of solving recurrences.

Methods for Solving Recurrences

- Following are the methods to find out a solution or bounds for recurrence relations.

1. Recursion tree method
2. Iteration method
3. Substitution method
4. Master theorem method

"Divide and Conquer" strategy

- Recurrences are derived from Recursive algorithms which are based on recursion.
- Recursion usually follows "Divide and Conquer" strategy
  - In algorithms, it means to divide the problem of a large input into smaller pieces of input data
  - Recursively divide the input until certain *smaller size* is reached. This stops the division of the input.
  - Then solve the smaller problems and combine the *piecewise* results to get a *global* solution for the original large input

"Divide and Conquer" strategy

- **Divide** the problem into a number of sub-problems

- **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough (Base Case), just solve the sub-problems in a straightforward manner.

- **Combine** the solutions to the sub-problems into the solution for the original problem.

Merge Sort

- Merge sort is a sorting algorithm
- Merge sort follows the “divide and conquer” strategy and is a recursive algorithm
- It has better performance then the insertion sort, bubble sort and selection sort for larger data

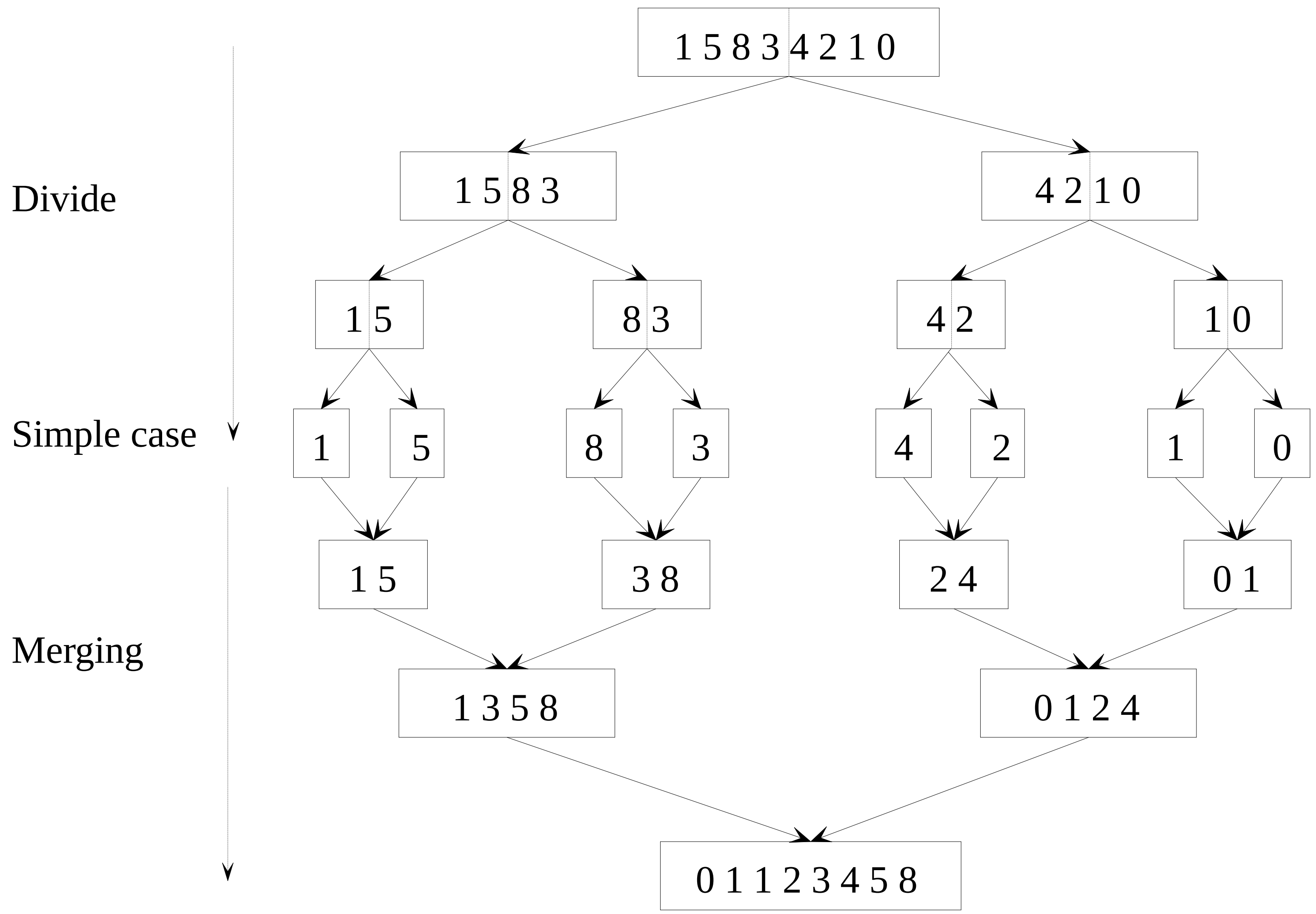
Divide & Conquer strategy in Merge Sort

- **Divide:**
  - Divide the *n*-elements list to be sorted into two subsequences of *n*/2 elements each
- **Conquer:**
  - Sort the two subsequences recursively using *Merge Sort*
- **Combine:**
  - Merge the two sorted subsequences to produce the sorted sequence

- The recursion stops when the sub-sequence to be sorted reaches the length of **1**. Sequence of length 1 is already in sorted order, and nothing in reality is done for sorting.

- The actual ***sorting related activity*** in the merge sort occurs during the ***merging process*** of the two sorted already ***sub-sequences***. i.e the combine step.

Merge sort example

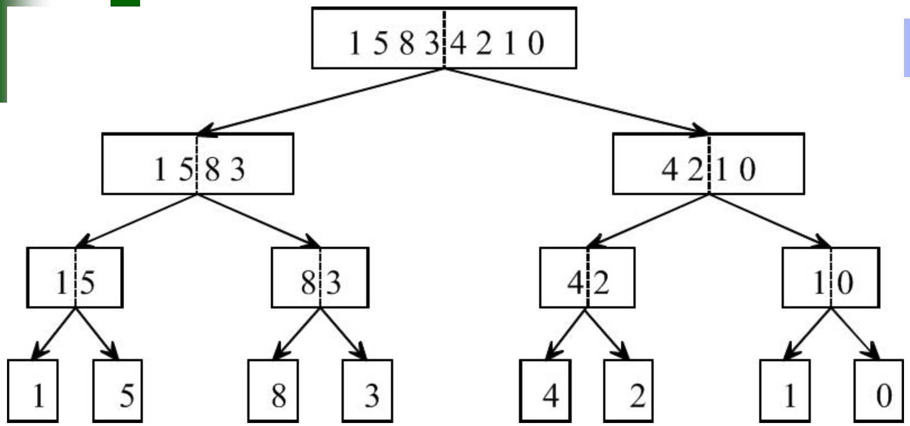


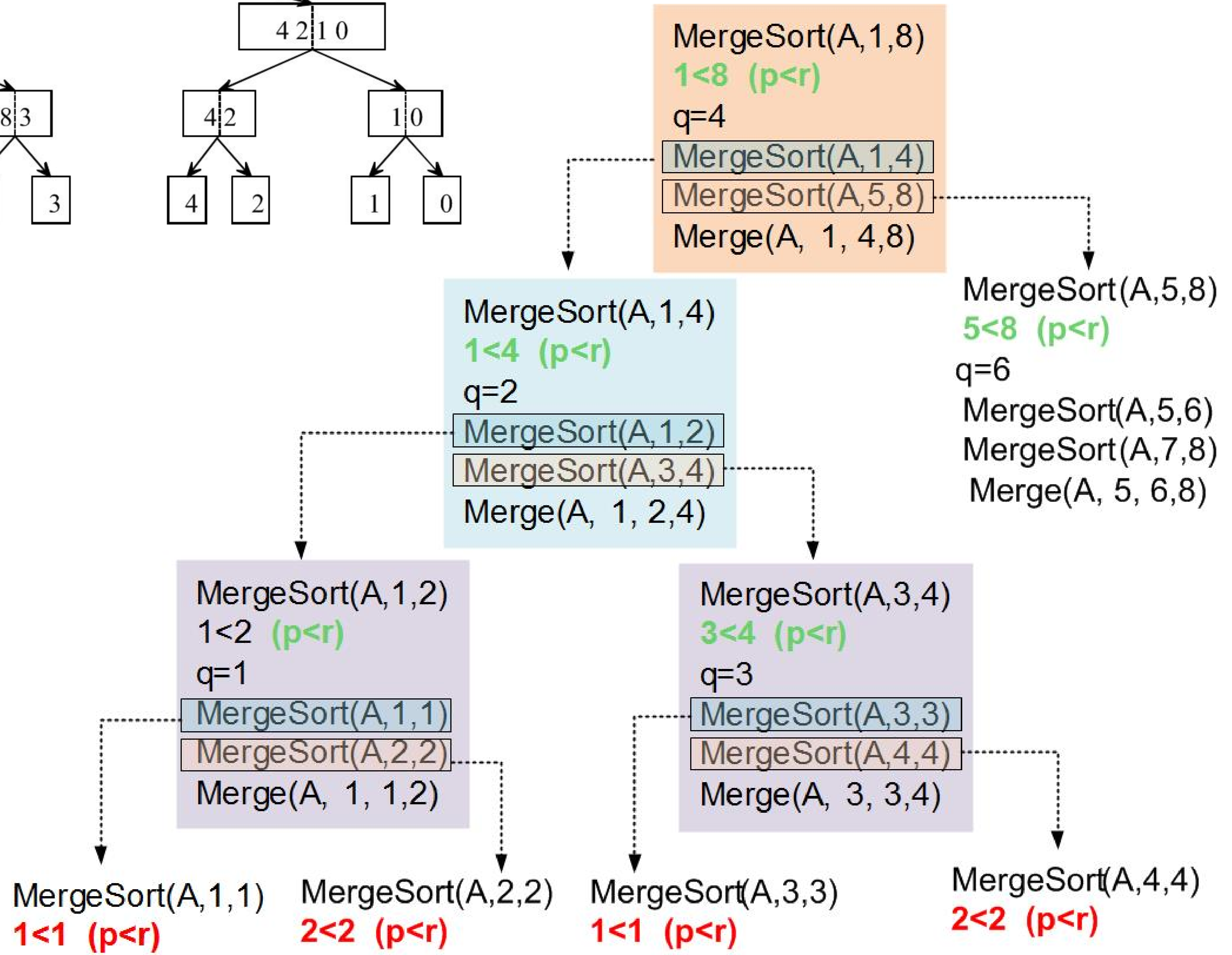
Merge sort Algorithm

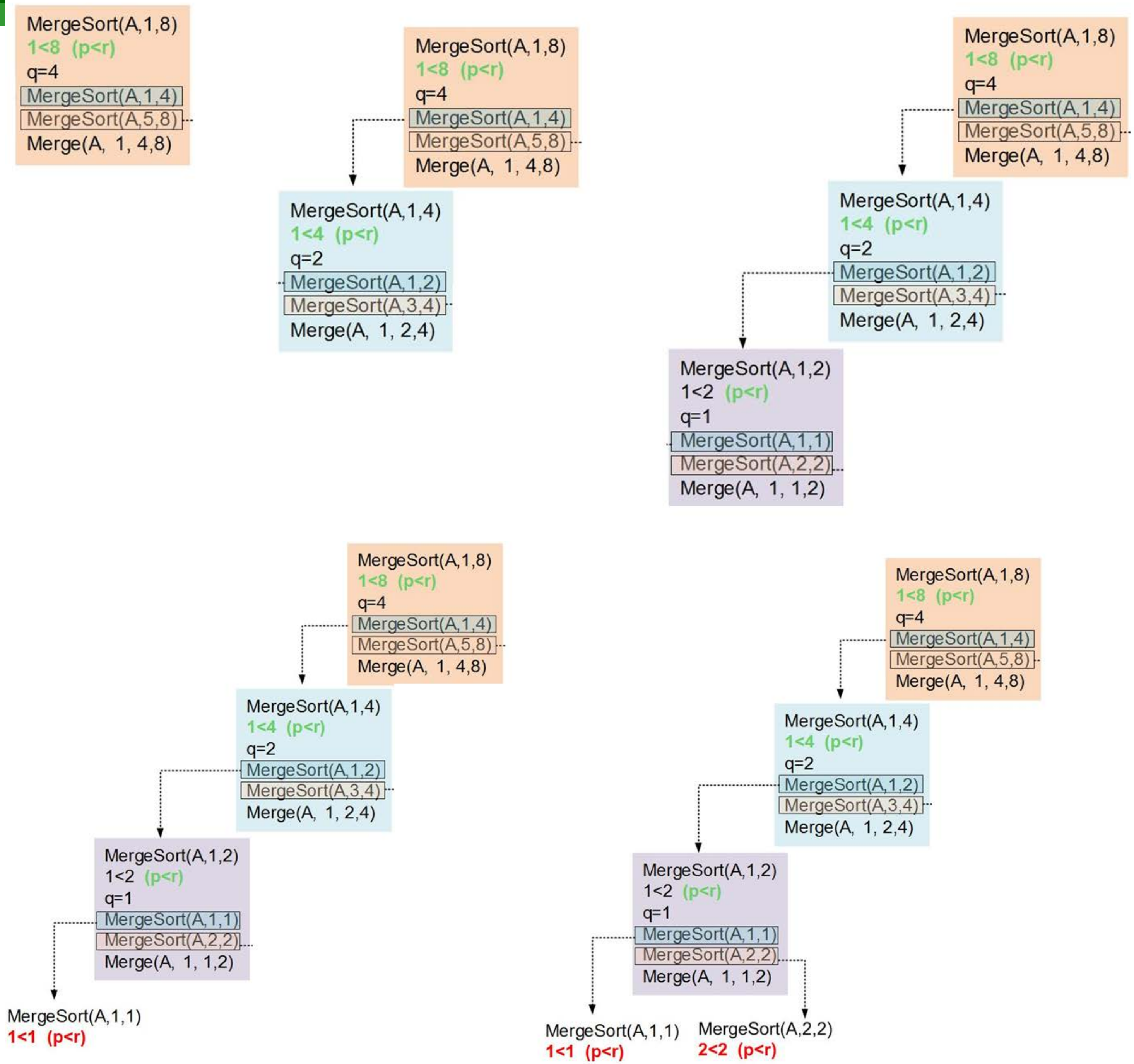
$\text{MERGE-SORT}(A, p, r)$

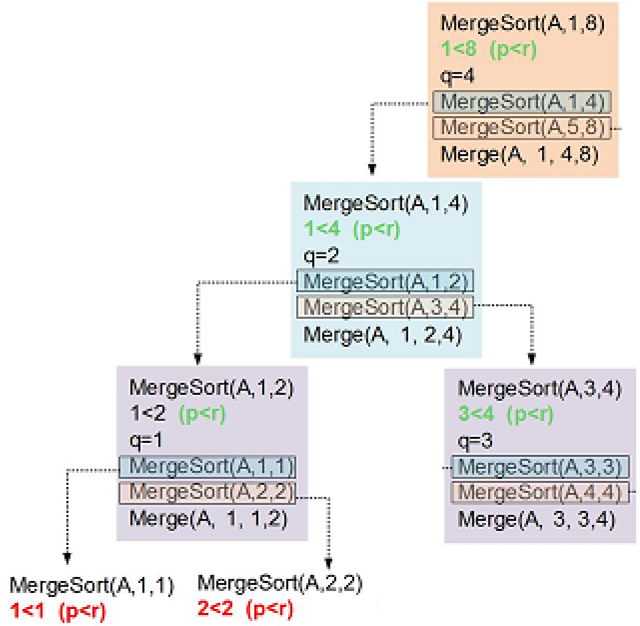
if $p < r$	$\triangleright$ Check for base case
then $q \leftarrow \lfloor (p + r)/2 \rfloor$	$\triangleright$ Divide
$\text{MERGE-SORT}(A, p, q)$	$\triangleright$ Conquer
$\text{MERGE-SORT}(A, q + 1, r)$	$\triangleright$ Conquer
$\text{MERGE}(A, p, q, r)$	$\triangleright$ Combine

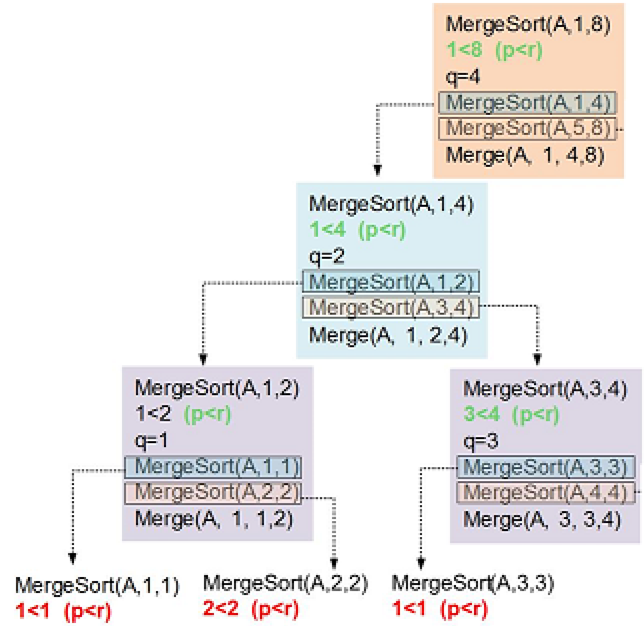
- The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure ***MERGE(A, p, q, r),*** where A is an array and p, q, and r are indices numbering elements of the array such that p ≤ q < r.
- The procedure assumes that the subarrays **A[p : q]** and **A[q+ 1: r]** are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray A[p: r].

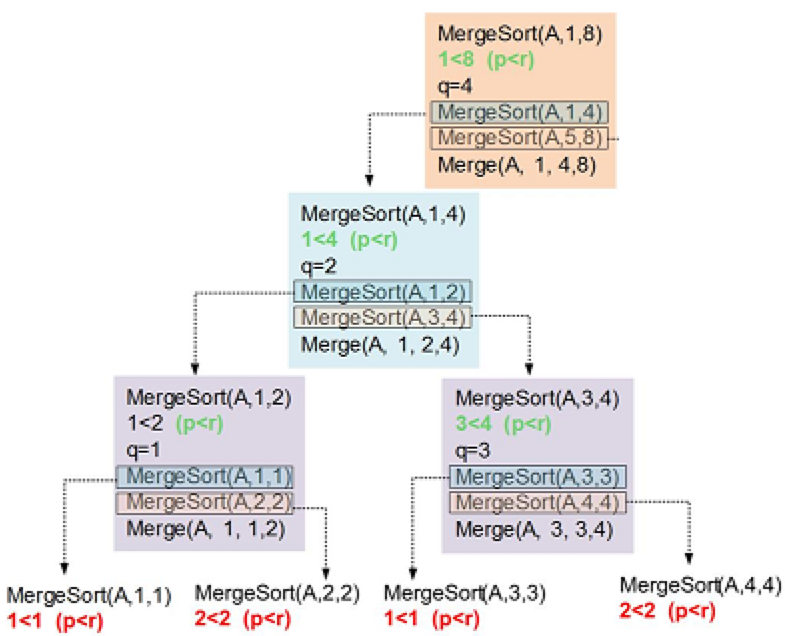

1 5 8 3 4 2 1 0
1 5 8 3
4 2 1 0
1 5
8 3
4 2
1 0
1
5
8
3
4
2
1
0

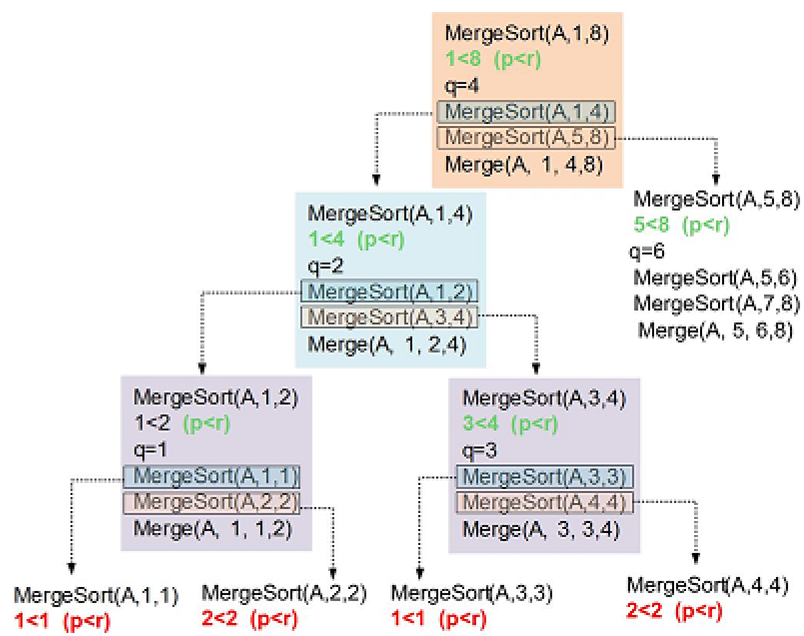

MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,5,8)
5<8 (p<r)
q=6
MergeSort(A,5,6)
MergeSort(A,7,8)
Merge(A, 5, 6,8)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,3,4)
3<4 (p<r)
q=3
MergeSort(A,3,3)
MergeSort(A,4,4)
Merge(A, 3, 3,4)
MergeSort(A,1,1)
1<1 (p<r)
MergeSort(A,2,2)
2<2 (p<r)
MergeSort(A,3,3)
1<1 (p<r)
MergeSort(A,4,4)
2<2 (p<r)


MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,1,1)
1<1 (p<r)
MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,1,1)
1<1 (p<r)
MergeSort(A,2,2)
2<2 (p<r)


MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,3,4)
3<4 (p<r)
q=3
MergeSort(A,3,3)
MergeSort(A,4,4)
Merge(A, 3, 3,4)
MergeSort(A,1,1)
1<1 (p<r)
MergeSort(A,2,2)
2<2 (p<r)


MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,3,4)
3<4 (p<r)
q=3
MergeSort(A,3,3)
MergeSort(A,4,4)
Merge(A, 3, 3,4)
MergeSort(A,1,1)
1<1 (p<r)
MergeSort(A,2,2)
2<2 (p<r)
MergeSort(A,3,3)
1<1 (p<r)


MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,3,4)
3<4 (p<r)
q=3
MergeSort(A,3,3)
MergeSort(A,4,4)
Merge(A, 3, 3,4)
MergeSort(A,1,1)
1<1 (p<r)
MergeSort(A,2,2)
2<2 (p<r)
MergeSort(A,3,3)
1<1 (p<r)
MergeSort(A,4,4)
2<2 (p<r)


MergeSort(A,1,8)
1<8 (p<r)
q=4
MergeSort(A,1,4)
MergeSort(A,5,8)
Merge(A, 1, 4,8)
MergeSort(A,5,8)
5<8 (p<r)
q=6
MergeSort(A,5,6)
MergeSort(A,7,8)
Merge(A, 5, 6,8)
MergeSort(A,1,4)
1<4 (p<r)
q=2
MergeSort(A,1,2)
MergeSort(A,3,4)
Merge(A, 1, 2,4)
MergeSort(A,1,2)
1<2 (p<r)
q=1
MergeSort(A,1,1)
MergeSort(A,2,2)
Merge(A, 1, 1,2)
MergeSort(A,3,4)
3<4 (p<r)
q=3
MergeSort(A,3,3)
MergeSort(A,4,4)
Merge(A, 3, 3,4)
MergeSort(A,1,1)
1<1 (p<r)
MergeSort(A,2,2)
2<2 (p<r)
MergeSort(A,3,3)
1<1 (p<r)
MergeSort(A,4,4)
2<2 (p<r)

Pseudocode:

$\text{MERGE}(A, p, q, r)$

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$

for $i \leftarrow 1$ **to** n_1

  do $L[i] \leftarrow A[p + i - 1]$

for $j \leftarrow 1$ **to** n_2

  do $R[j] \leftarrow A[q + j]$

$L[n_1 + 1] \leftarrow \infty$

$R[n_2 + 1] \leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

for $k \leftarrow p$ **to** r

  do if $L[i] \leq R[j]$

    then $A[k] \leftarrow L[i]$

      $i \leftarrow i + 1$

    else $A[k] \leftarrow R[j]$

      $j \leftarrow j + 1$

n_1: calculate the size of left sorted array

n_2: calculate the size of left sorted array

Create two temp arrays

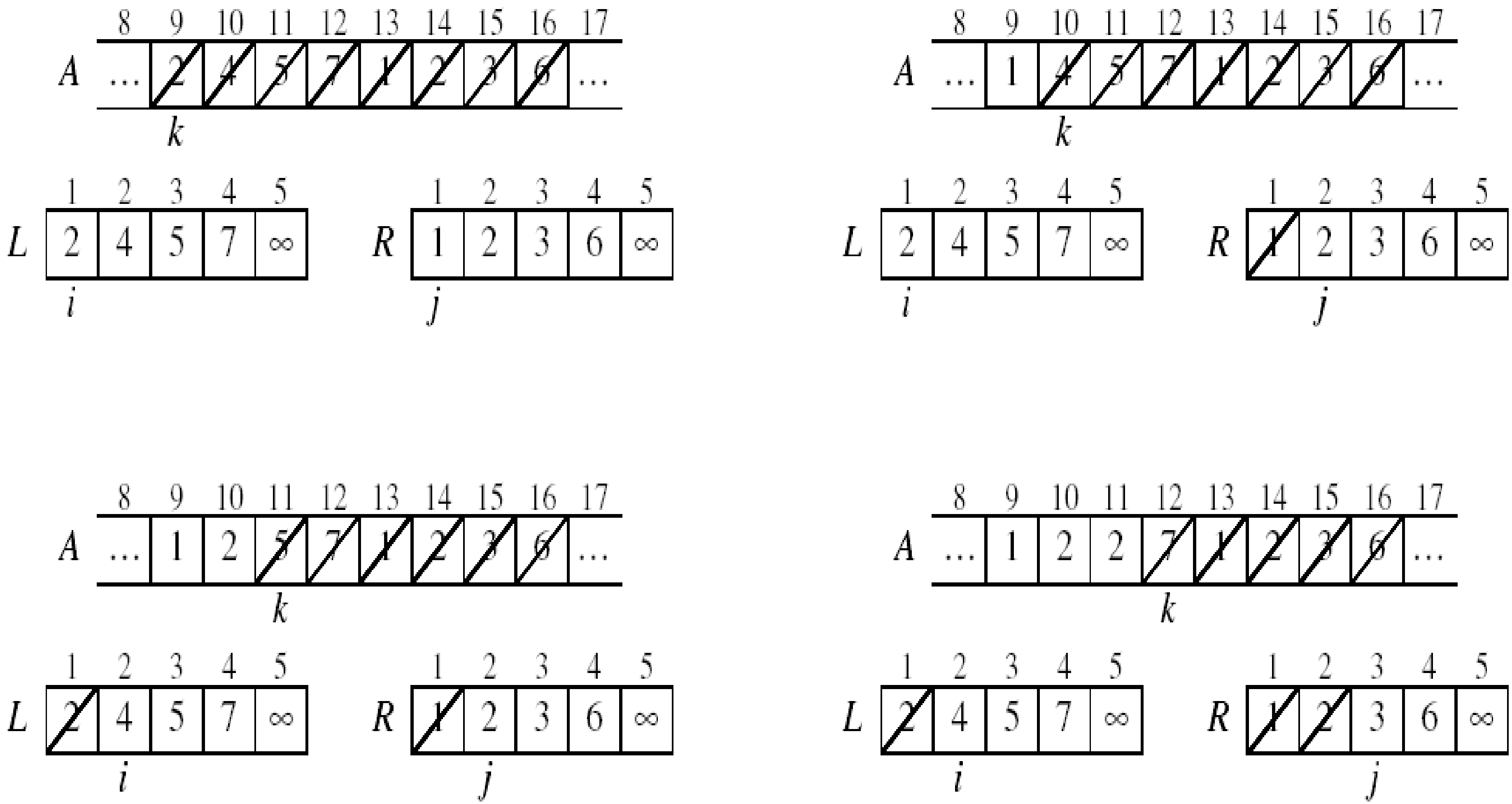
Copy left sorted array

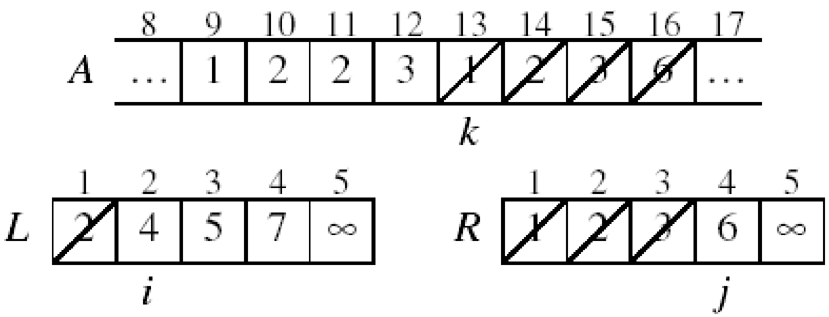
Copy second sorted array

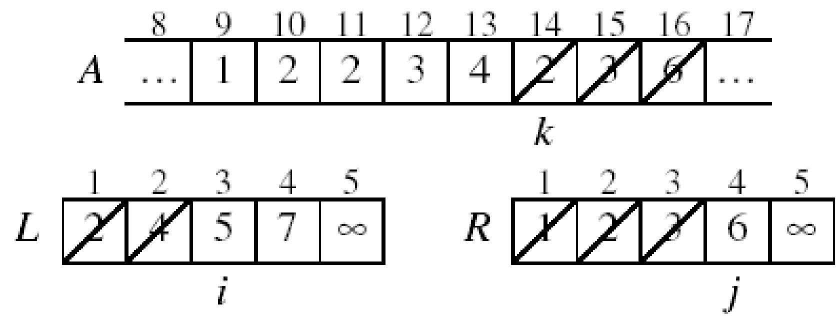
Assign very large values at both array's last locations.

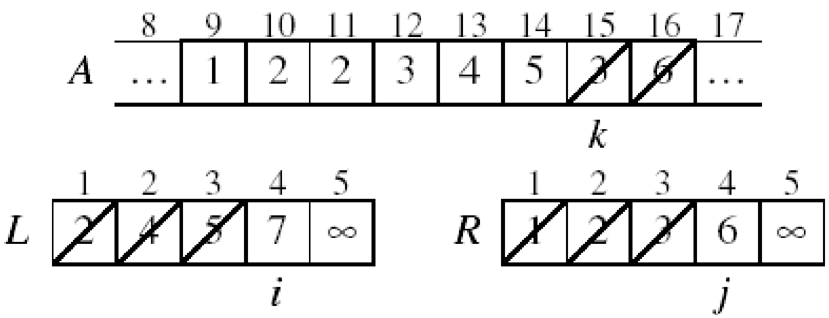
Merge and copy two sorted arrays while comparing values

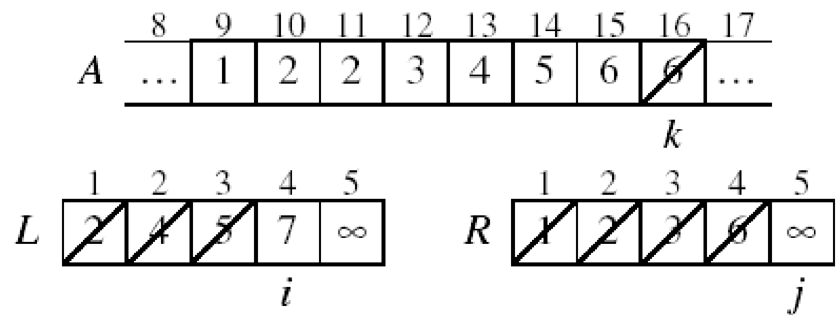
Example: A call of MERGE(9, 12, 16)

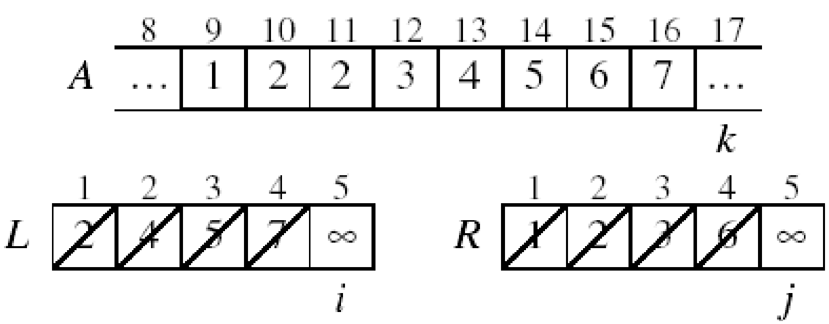












What if n is odd??

Analysis of Merge Sort

- Merge Sort is a Recursive Algorithm
- In order to analyze any recursive Algorithm we need to
  1. First find the **recurrence relation** for the algorithm
  2. Then **solve the recurrence** relation to find running time.

How to find a Recurrence Relation???

Finding a Recurrence Relation from divide-and-conquer Algorithm

- In a Divide and conquer algorithms

  $T(n)$ = running time on a problem of size n.

- If the problem size is small enough (say, $n \le c$ for some constant c), we have a ***base case***.
  - In divide & conquer the solution of base case is always constant time: $\Theta(1)$

- Otherwise, we divide problem into *'a' sub-problems*, each $1/b$ the size of the original.
  - In Merge Sort, a=2, b = 2.

- '*a*' sub-problems would take ***a T (n/b)*** time
  - There are '*a*' sub-problems to solve, each of size '*n/b*'.
  - *T(n)* is the time entire problem of size *n*.
  - Therefore sub-problem of size *n/b* would take *T (n/b)* time.
  - Therefore '*a*' sub-problems take *a T (n/b)* time.
- Let the time to divide a size-*n* problem be ***D(n)***
- Time to combine solutions be ***C(n).***
- We obtain the following relation for the recurrence.

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

Analysis of Merge Sort

- We now apply analysis procedure of divide and conquer on Merge Sort Algorithm
- For simplicity, assume that n is a power of 2
  - Each divide-step yields two sub-problems, both of size exactly $n/2$.
- The base case occurs when $n = 1$.
- When $n \geq 2$, Merge Sort steps are followed.

Recurrence Relation for Merge Sort

- **Divide:**
  - Divide is computing value of q as the average of p and r
  - It takes constant time $D(n) = \Theta(1)$
- **Conquer:**
  - Recursively solve 2 sub-problems, each of size $n/2$
  - $2T(n/2)$.
- **Combine:**
  - MERGE n-element sub-array takes (n) time
  - $C(n) = \Theta(n)$.

Cost of Combine: *Merge ()*

Running time of *MERGE(A, p, q, r)* procedure

The first two ***for*** loops take Θ(n1 +n2) = Θ(n) time.

The last ***for*** loop makes at most *n* iterations, each taking constant time, for Θ(n) time.

- T(n) = Θ(n1 + n2) + Θ(n)
- T(n) = Θ(n) + Θ(n)
- T(n) = Θ(n) + Θ(n)

- Therefore, cost of Combine is

T(n) = Θ(n)

Pseudocode:

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```

- Since *D(n)* = **Θ(1)** and *C(n)* = **Θ(n)**, summed together they give a function that is linear in *n*: **Θ(n)**
  - ✓ *D(n)* + *C(n)* = **Θ(n)**
- Hence recurrence for merge sort running time is

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

we will next apply a Recurrence Relation solving technique to get the running time for Merge Sort.