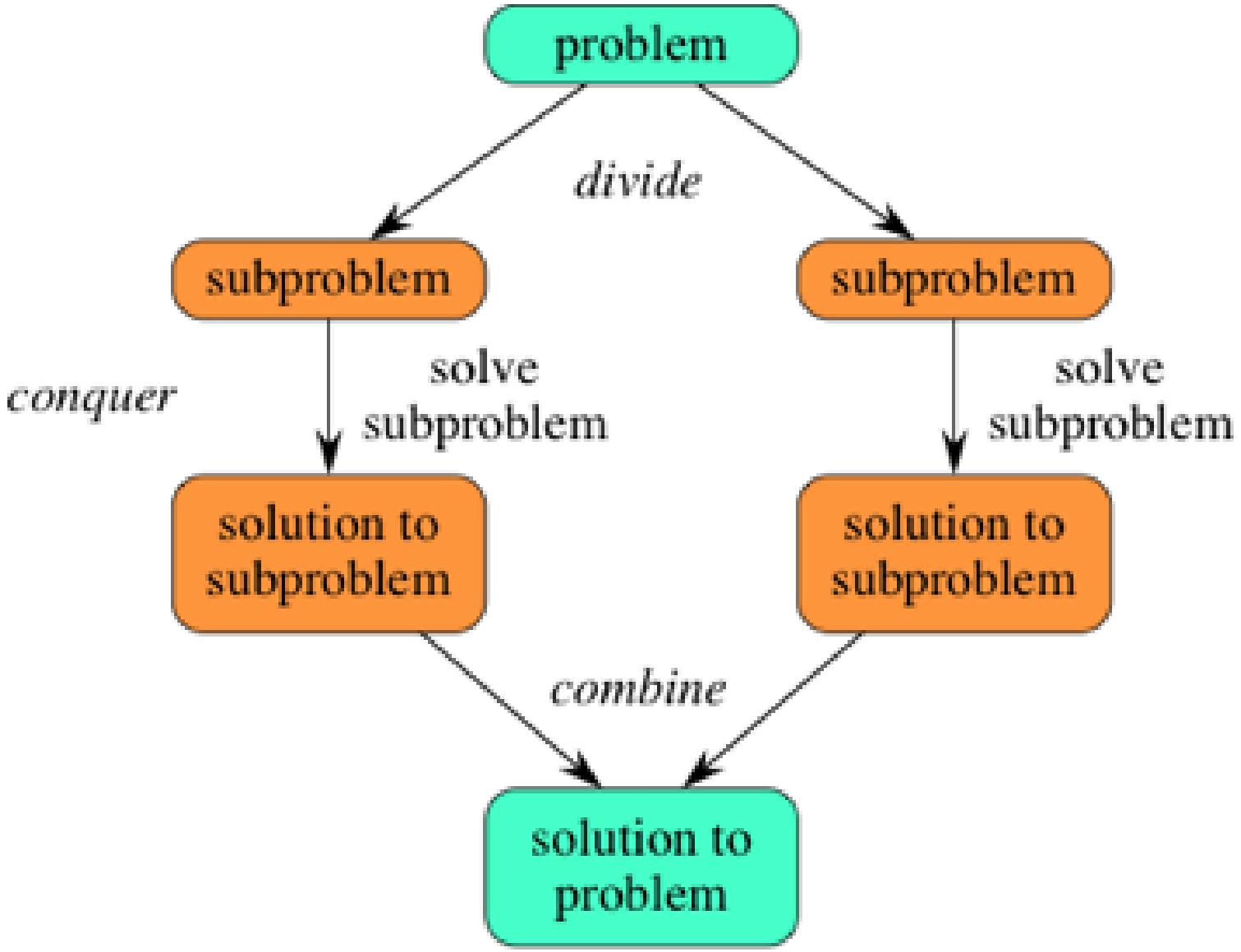
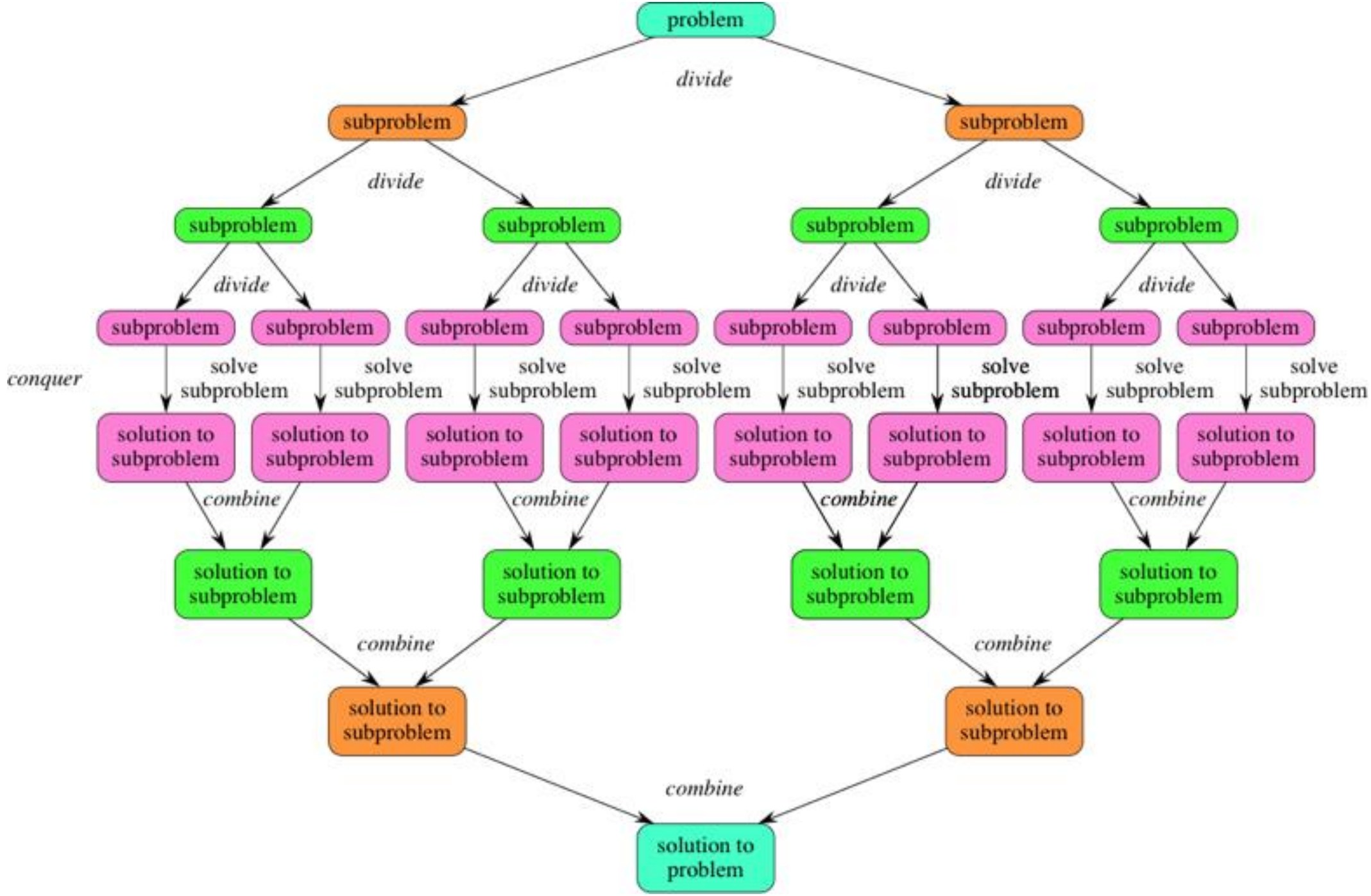
Divide Conquer Algorithm

Divide Conquer and Combine Algorithm

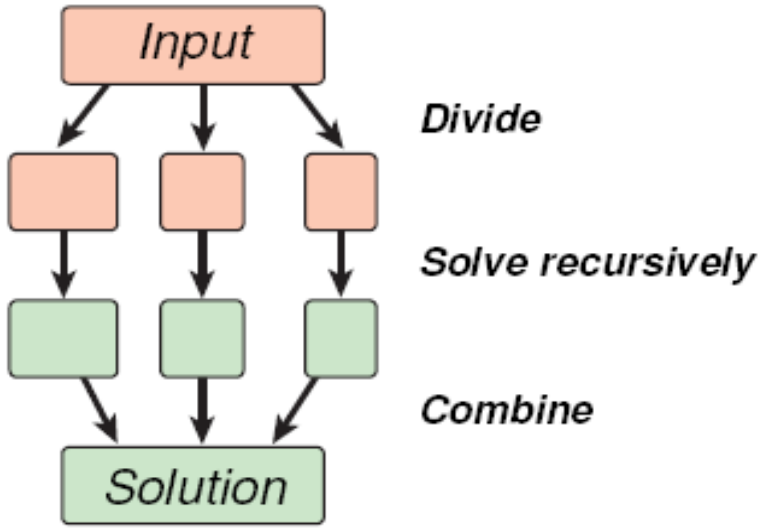



problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
combine
combine
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
solution to subproblem
solution to subproblem
combine
solution to problem

Divide Partition the input into two or more pieces, often of approximately equal size.

Conquer Recursively solve the problem on the pieces produced in the divide step.

Combine Combine the solutions obtained on the pieces to obtain a solution to the problem on the whole input.



Often, either the divide or the combine step is trivial.

Examples:

- ***Merge Sort:*** All the work is done in the combine step.
- ***Quicksort:*** All the work is done in the divide step.

Divide-and-Conquer: Applications

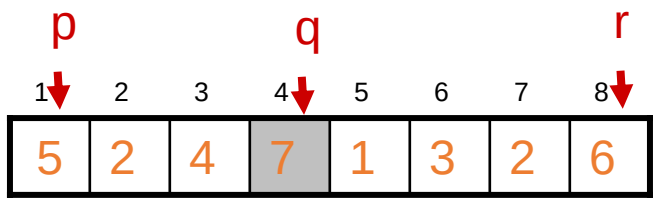
- **Merge Sort**
- **Quick Sort**

Merge Sort Approach

- To sort an array $A[p \,..\, r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Merge Sort

Alg.: MERGE-SORT(*A*, p, r)

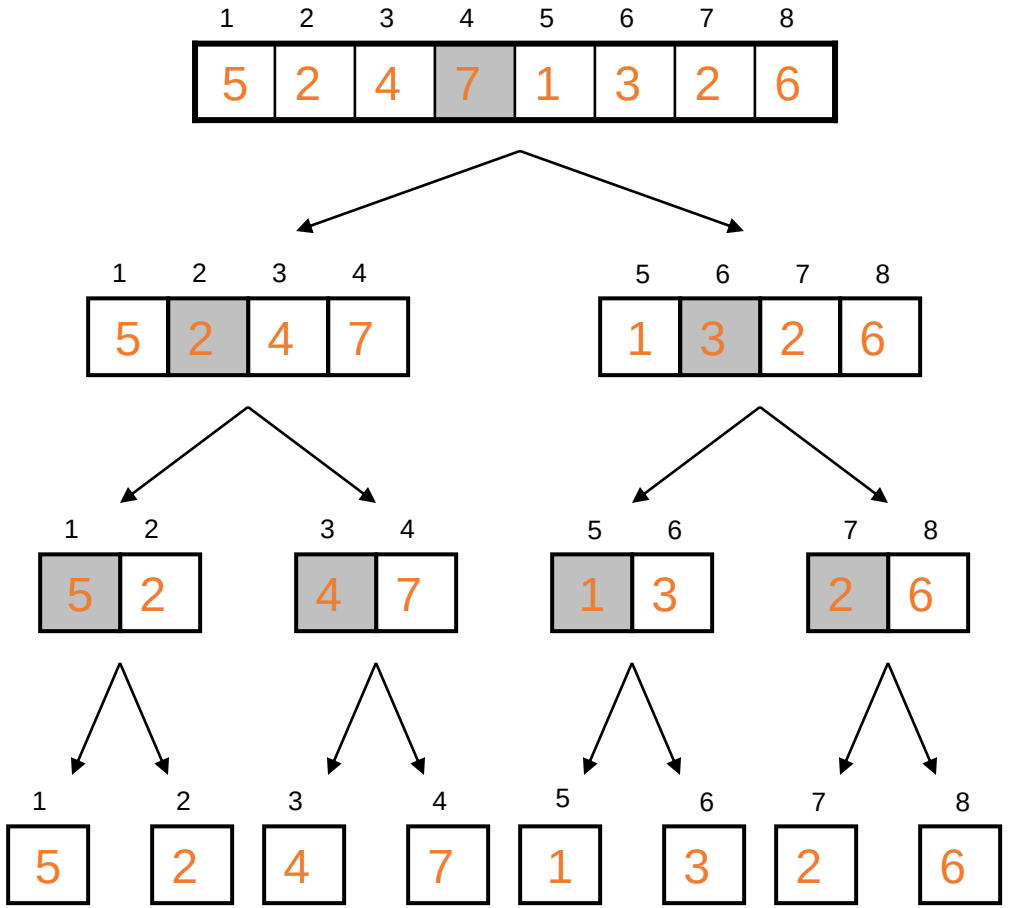



```
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```

- Initial call: MERGE-SORT(*A*, 1, n)

Example – **n** Power of 2

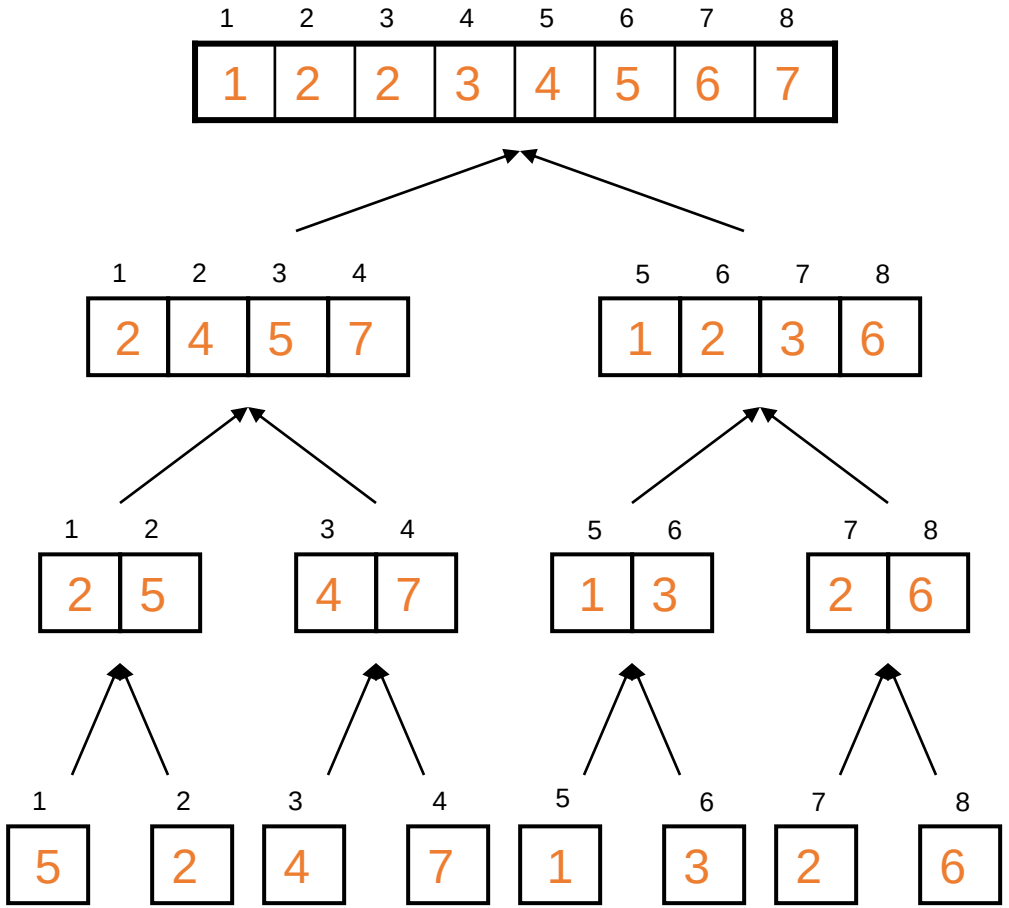
Divide



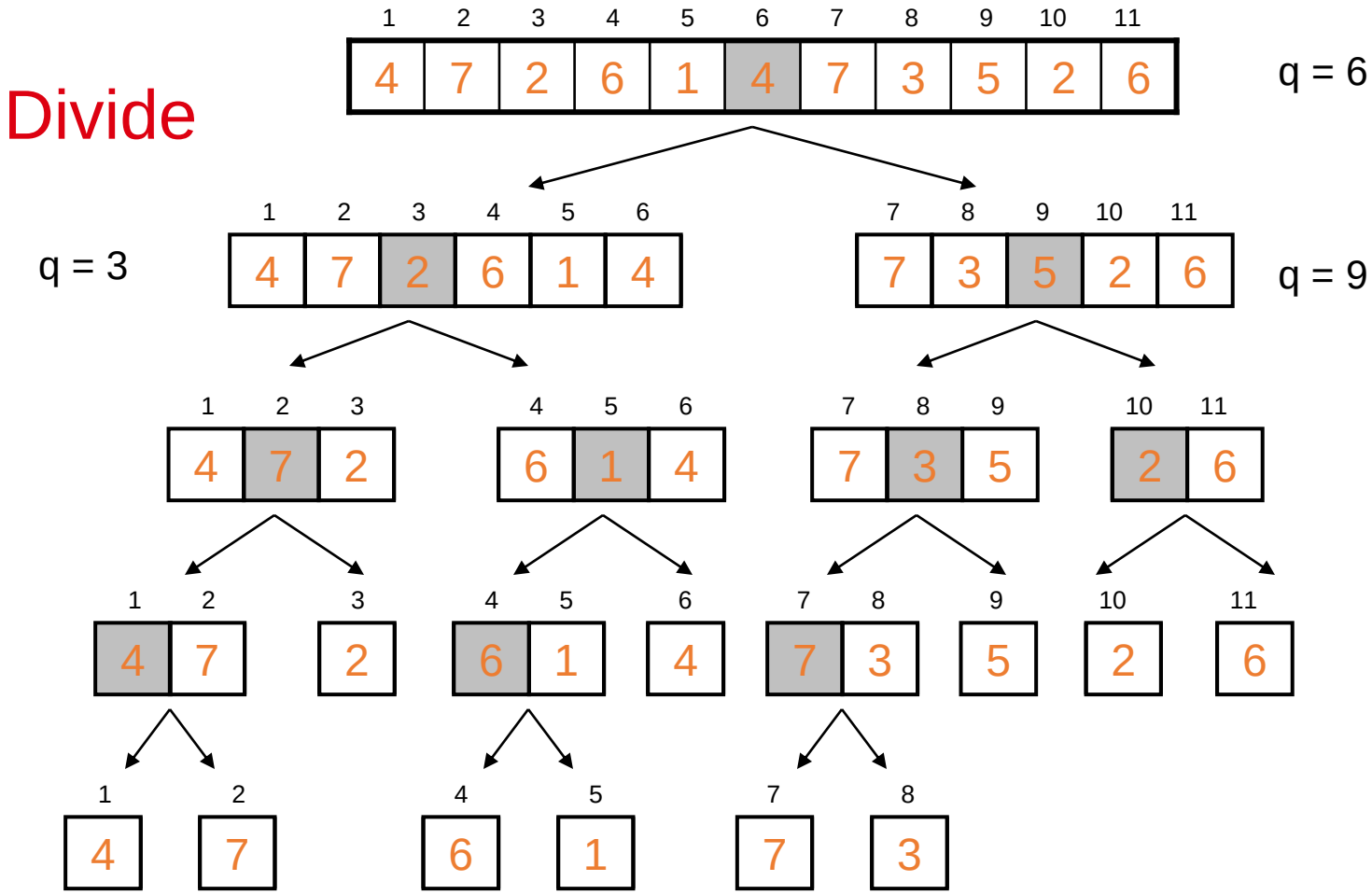
q = 4

Example – **n** Power of 2

Conquer and Merge

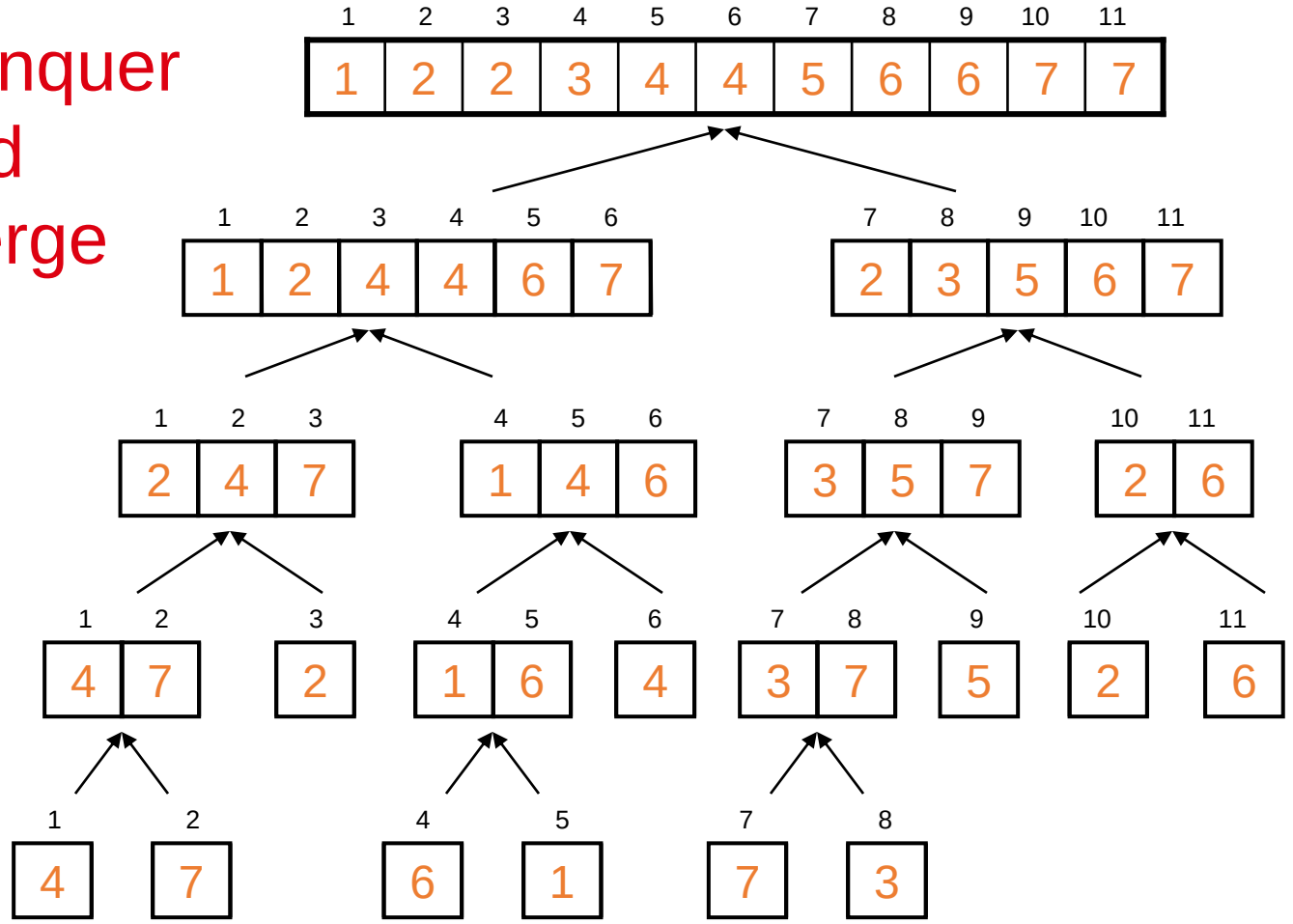


Example – **n** Not a Power of 2

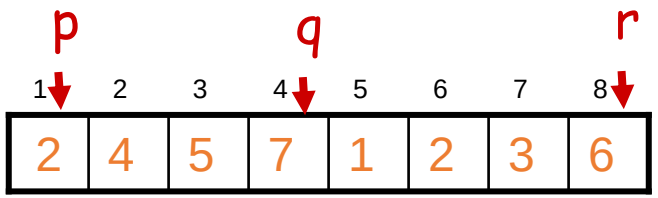


Example – **n** Not a Power of 2

Conquer and Merge

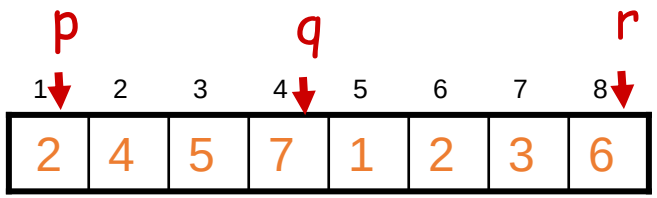


Merging

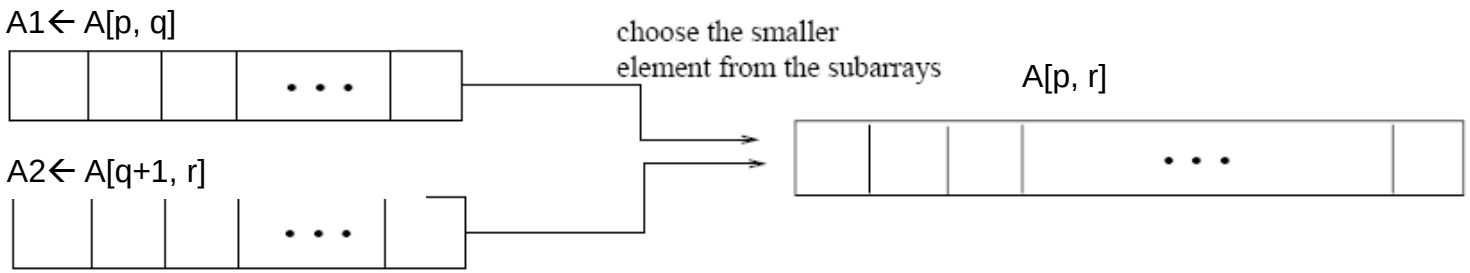


- **Input:** Array A and indices p, q, r such that $p \le q < r$
  - Subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are sorted
- **Output:** One single sorted subarray $A[p\,..\,r]$

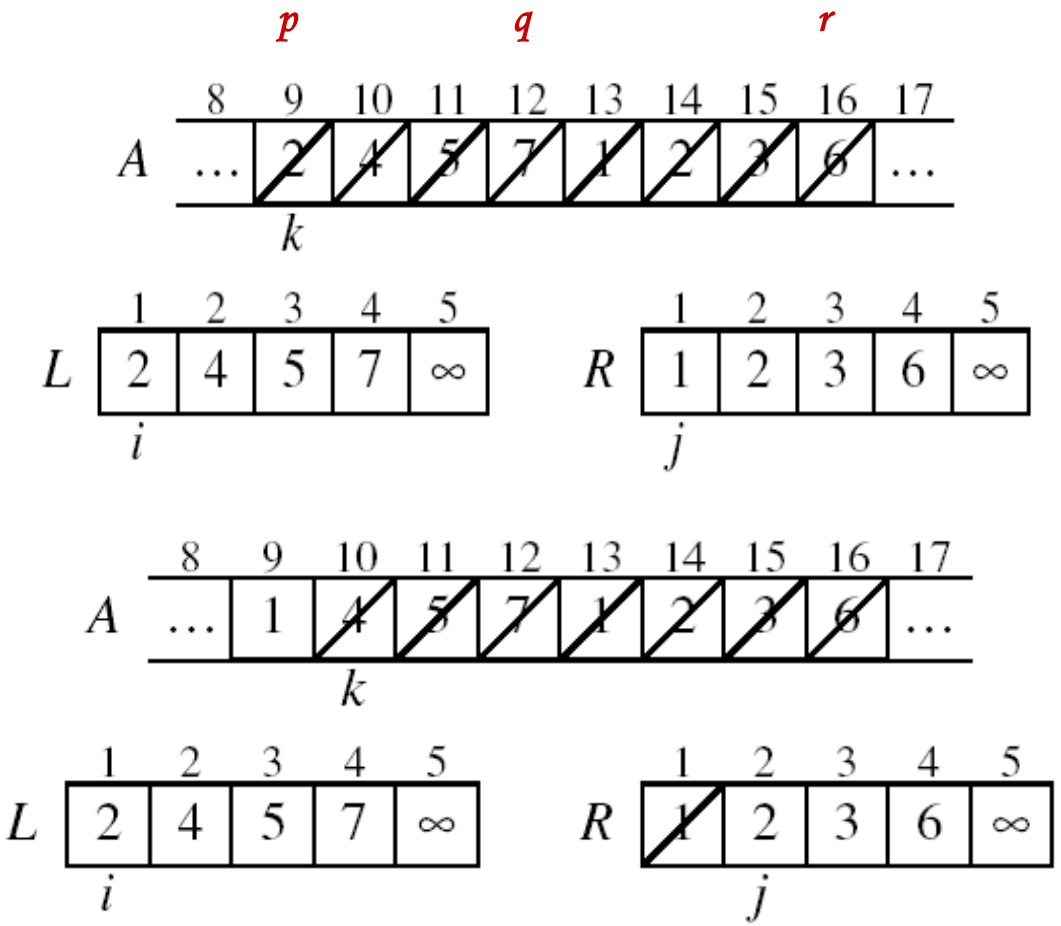
Merging



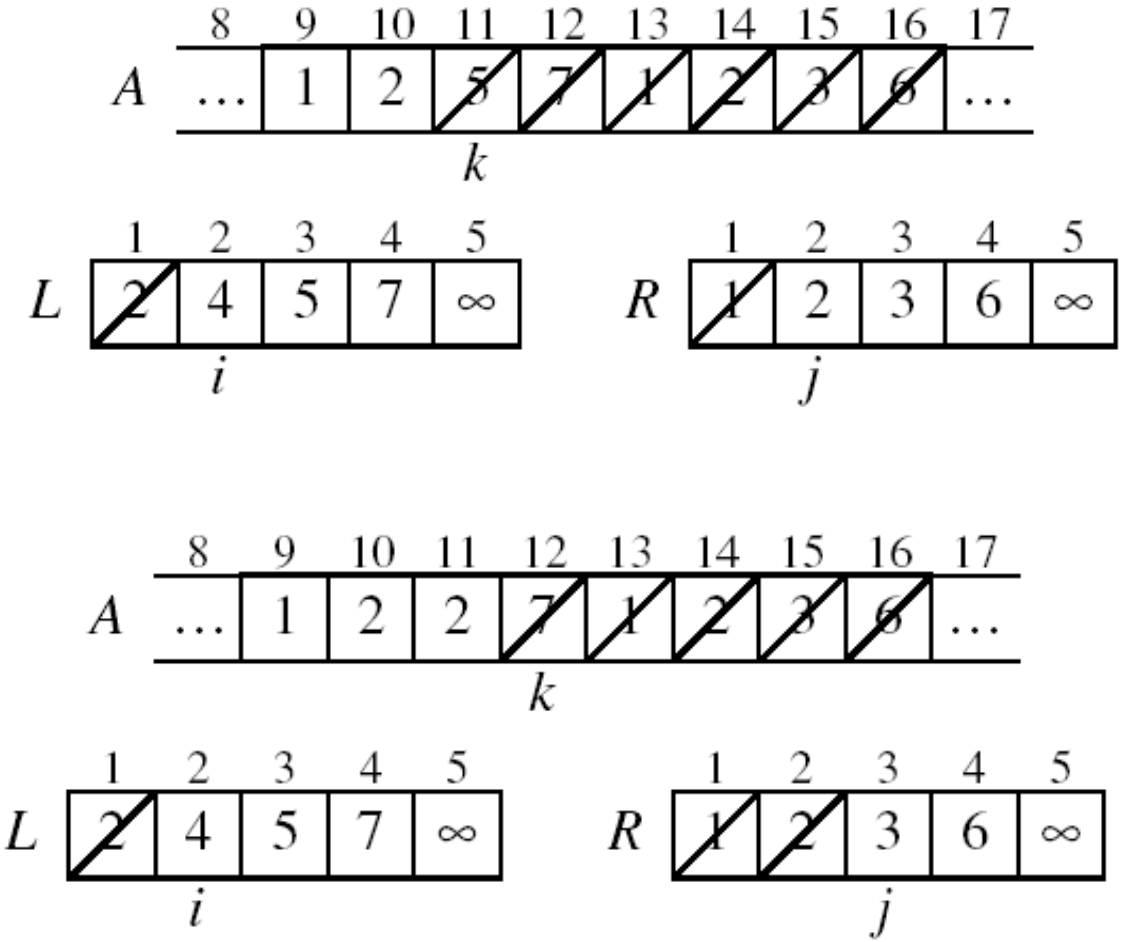
- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile



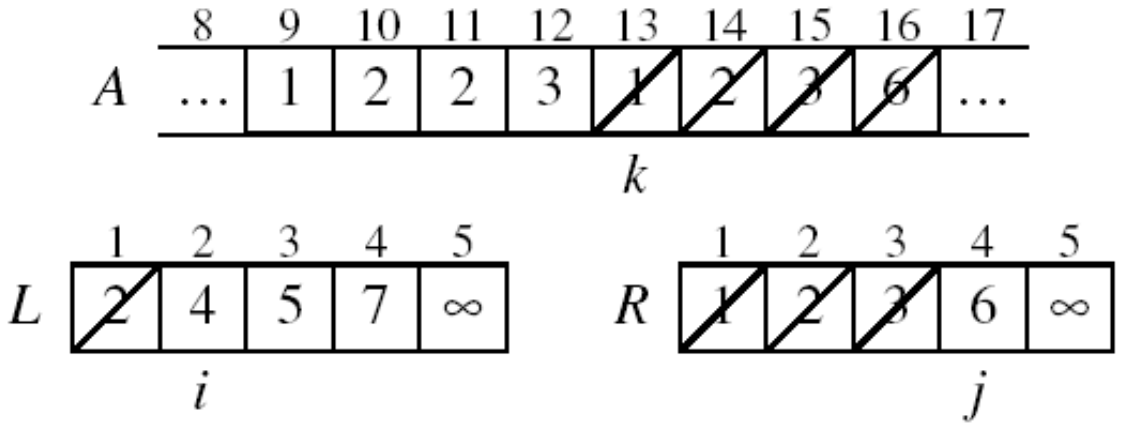
Example: MERGE(A, 9, 12, 16)

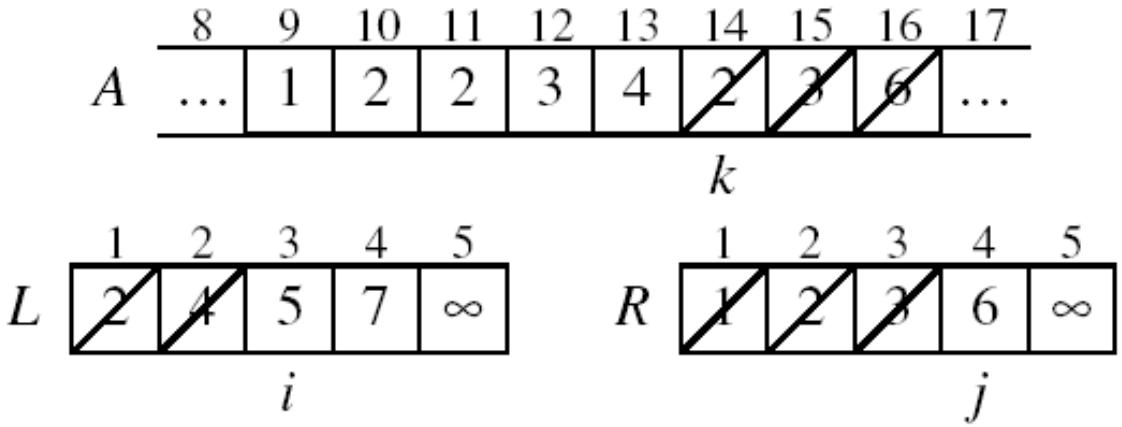


Example: MERGE(A, 9, 12, 16)

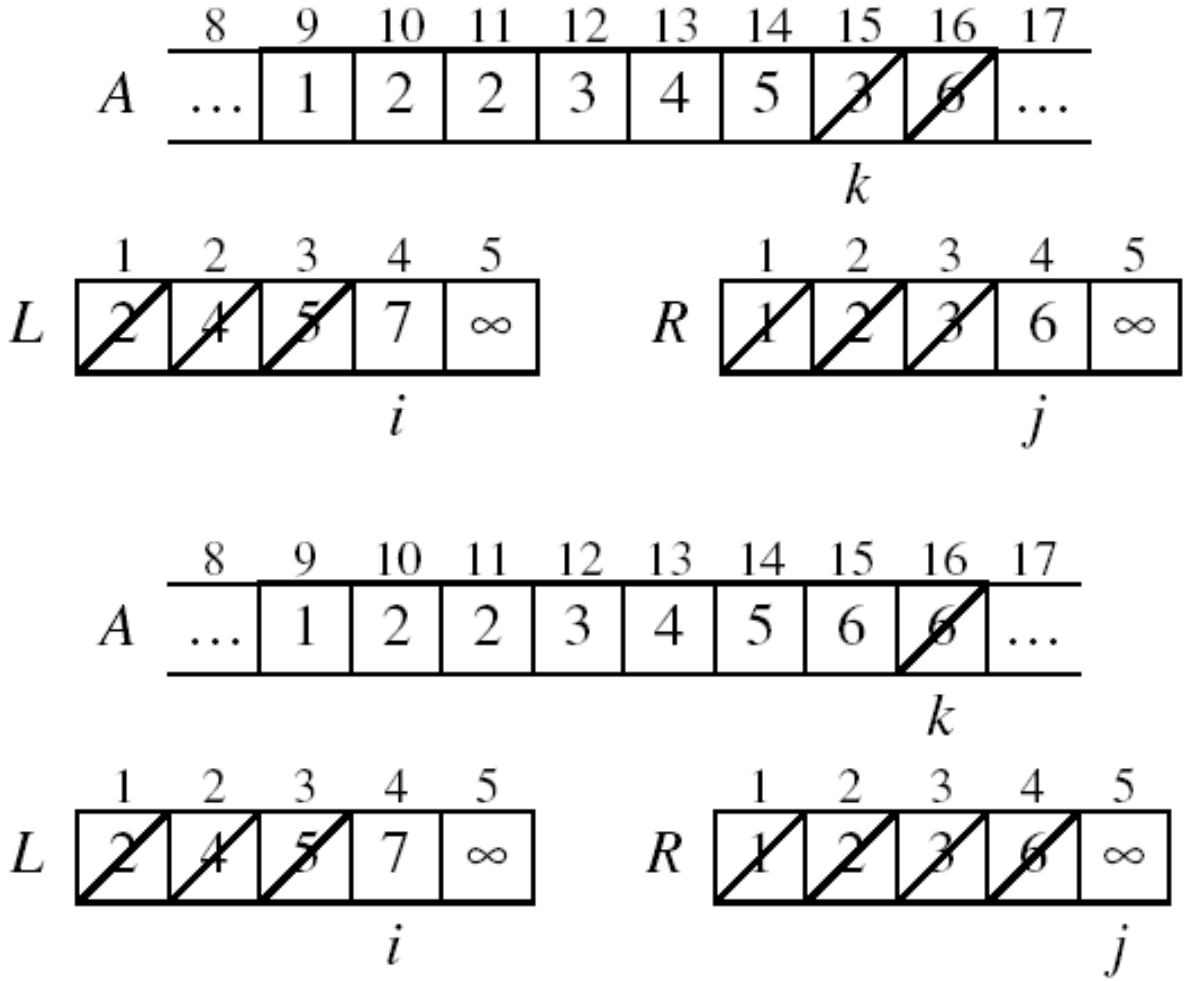


Example (cont.)

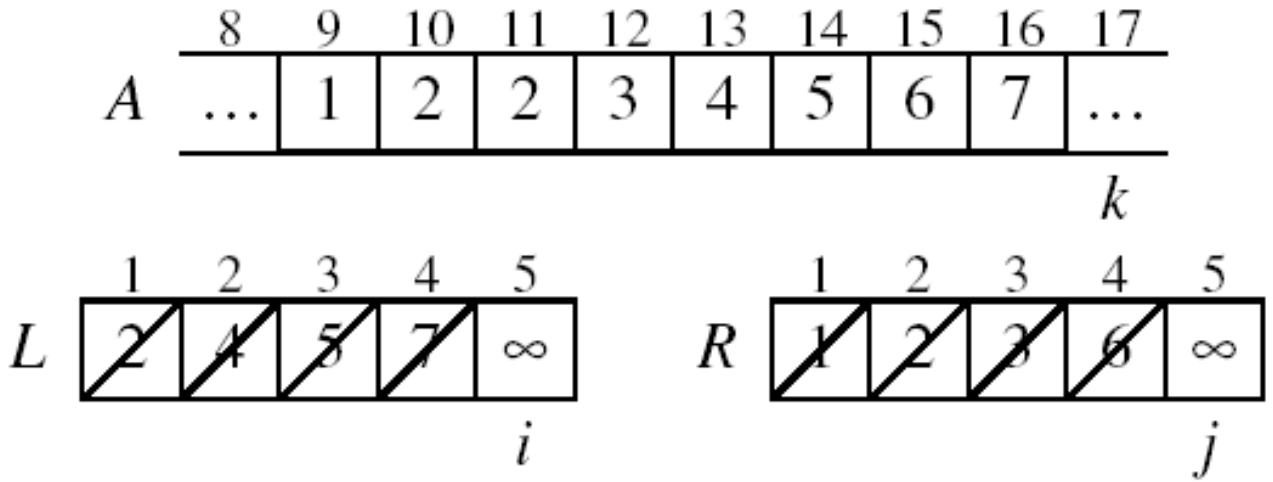




Example (cont.)



Example (cont.)



Done!

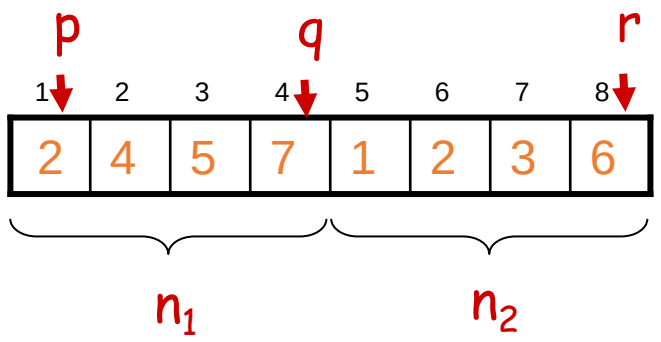
Merge - Pseudocode

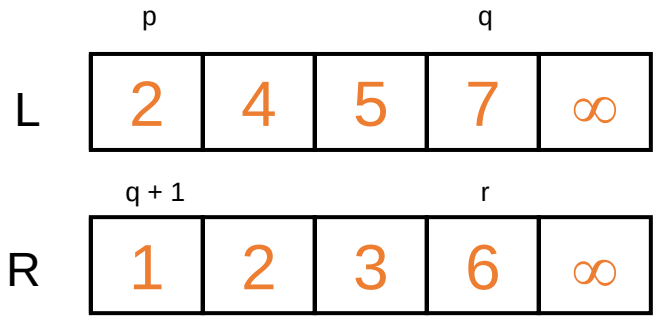
Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into

$L[1 \,..\, n_1 + 1]$ and the next n_2 elements into $R[1 \,..\, n_2 + 1]$

1. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
2. $i \leftarrow 1$; $j \leftarrow 1$
3. **for** $k \leftarrow p$ **to** r
4. **do if** $L[\, i \,] \leq R[\, j \,]$
5. **then** $A[k] \leftarrow L[\, i \,]$
6. $i \leftarrow i + 1$
7. **else** $A[k] \leftarrow R[\, j \,]$
8. $j \leftarrow j + 1$





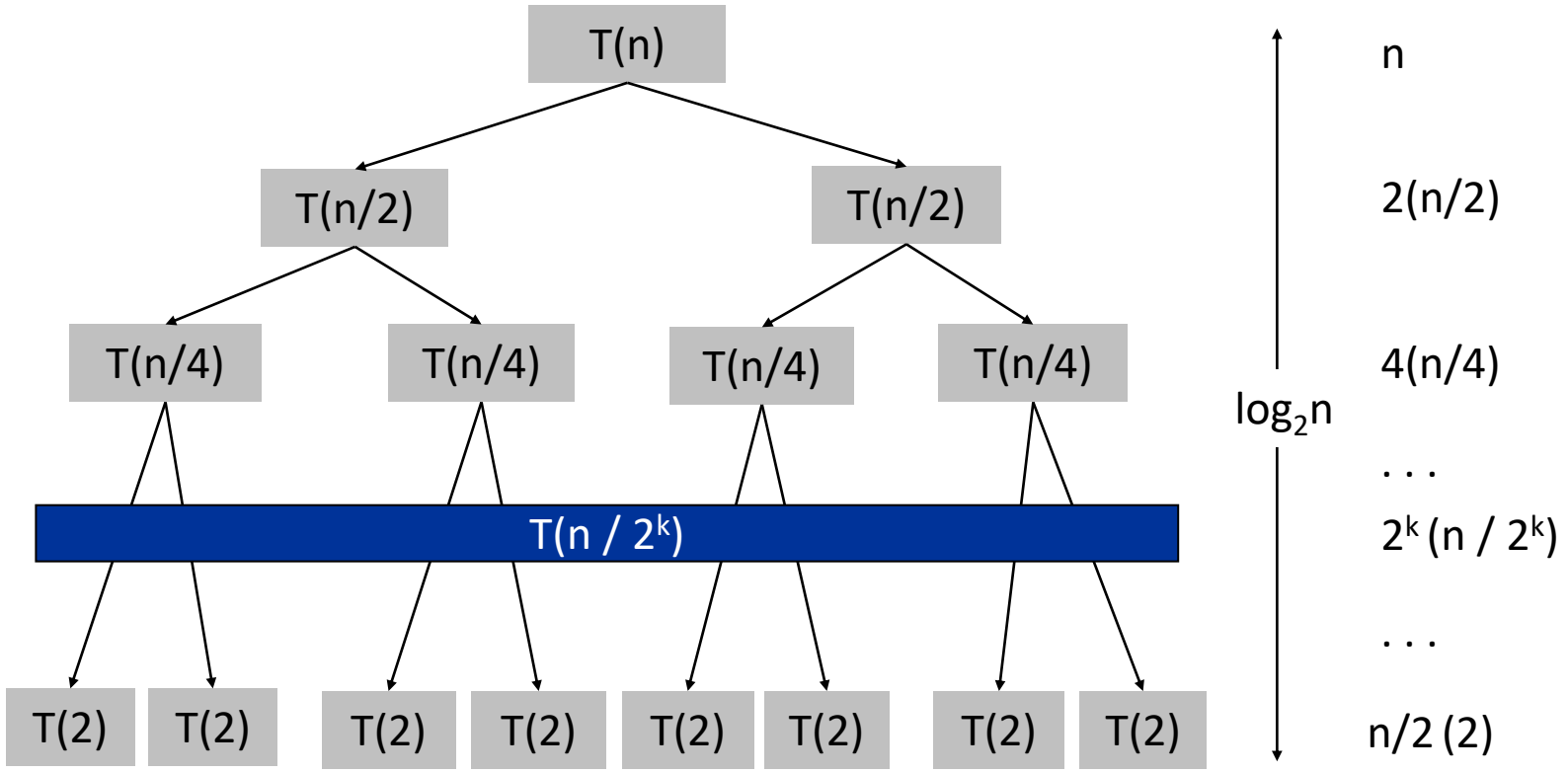
Analysis of Merge Sort

Statement	Effort
`MergeSort(A, P, r) {`	`T(n)`
`  if (p < r) {`	`Θ(1)`
`    q = floor(( p + r) / 2);`	`Θ(1)`
`    MergeSort(A, p, q);`	`T(n/2)`
`    MergeSort(A, q+1, r);`	`T(n/2)`
`    Merge(A, p, q, r);`	`Θ(n)`
`  }`	
`}`	

- So T(n) = $\Theta(1)$ when n = 1, and

  $2T(n/2) + \Theta(n)$ when n > 1

Analysis of Merge Sort



$\log_2 n$

n

2(n/2)

4(n/4)

. . .

$2^k (n / 2^k)$

. . .

n/2 (2)

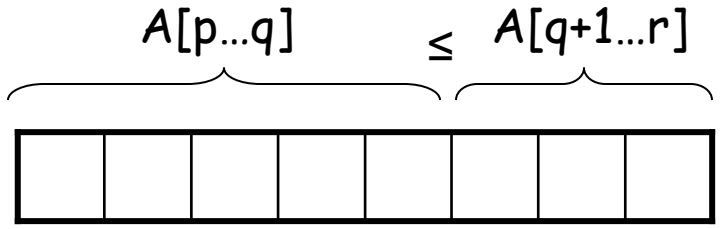
$n \log_2 n$

Analysis of Merge Sort

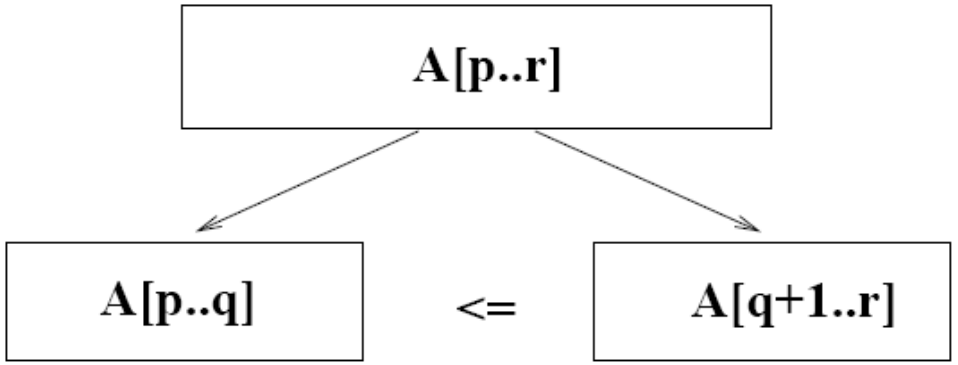
- Running time insensitive of the input
- Advantages:
  - Guaranteed to run in $\Theta(n \lg n)$
- Disadvantage
  - Requires extra space $\approx N$

Quick sort

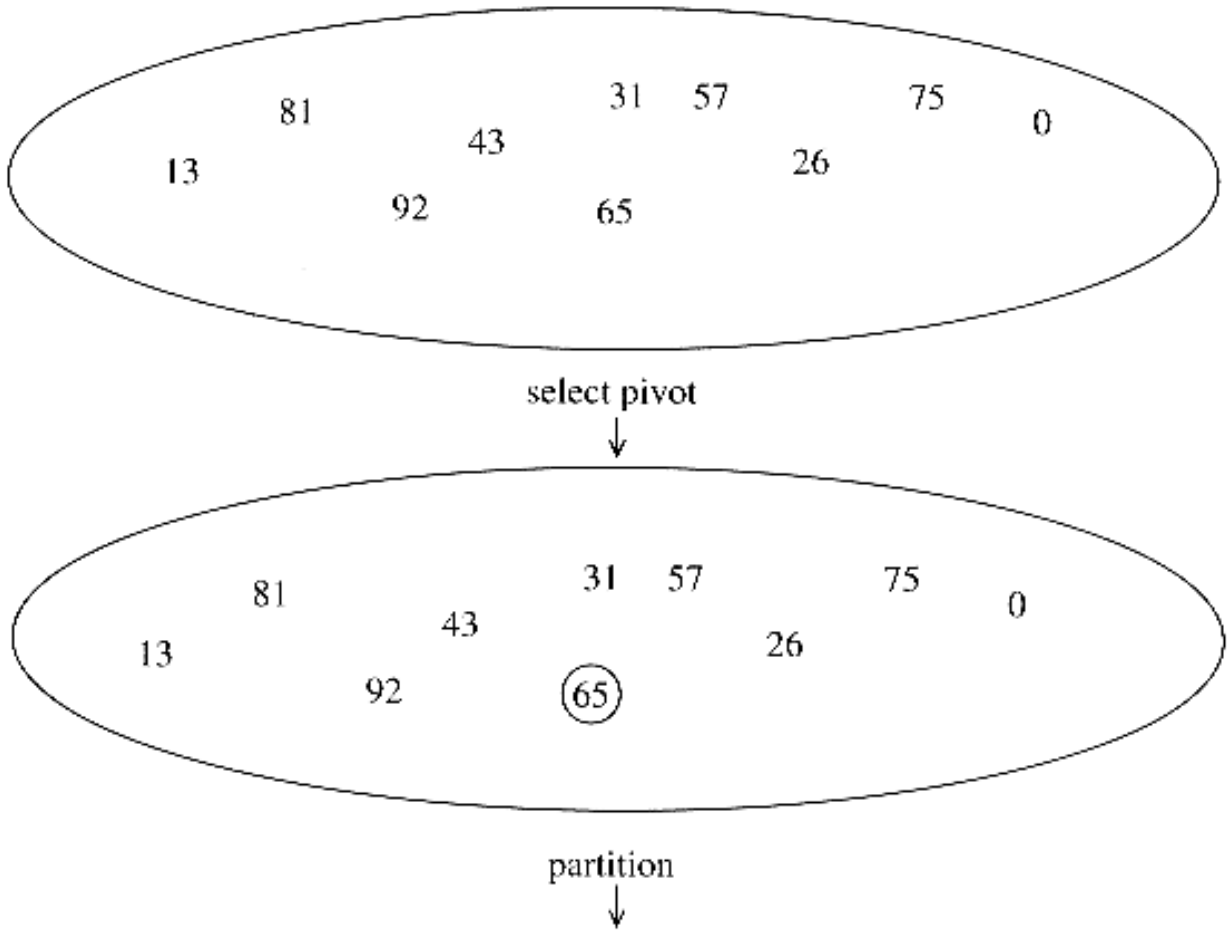
Quick sort



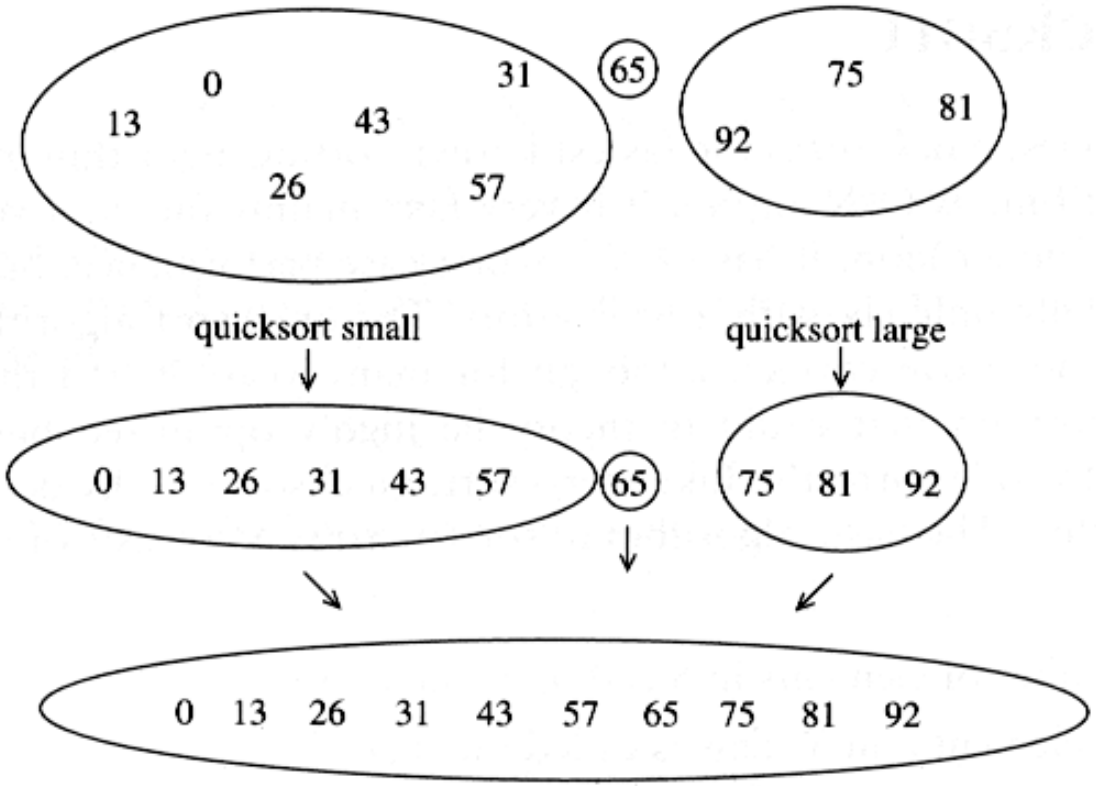
- Sort an array $A[p…r]$
- **Divide**
  - Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
  - Need to find index q to partition the array



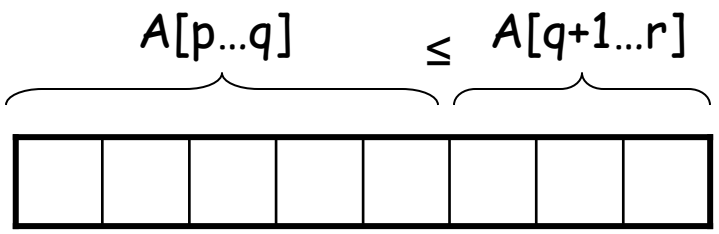
Quick Sort: Example



Example of Quick Sort...



Quicksort



- **Conquer**
  - Recursively sort $A[p..q-1]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

QUICKSORT

Initially: p=0, r=n-1

Alg.: QUICKSORT(*A*, p, r)

if p < r

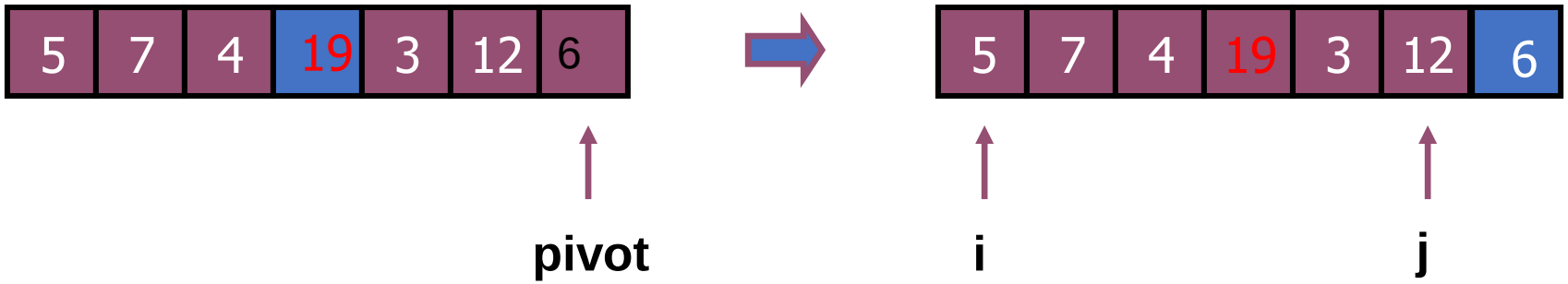
then *q* ← PARTITION(*A*, p, r)

QUICKSORT (*A*, p, *q*-1)

QUICKSORT (*A*, *q*+1, r)

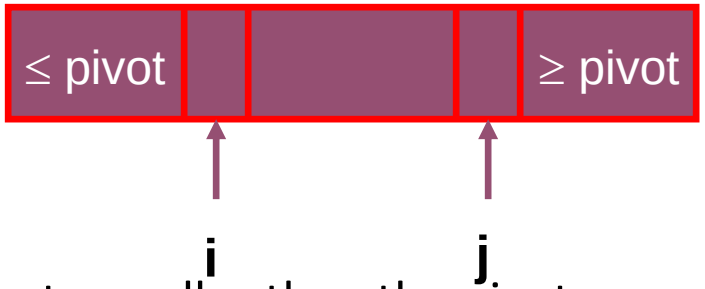
Partitioning Strategy

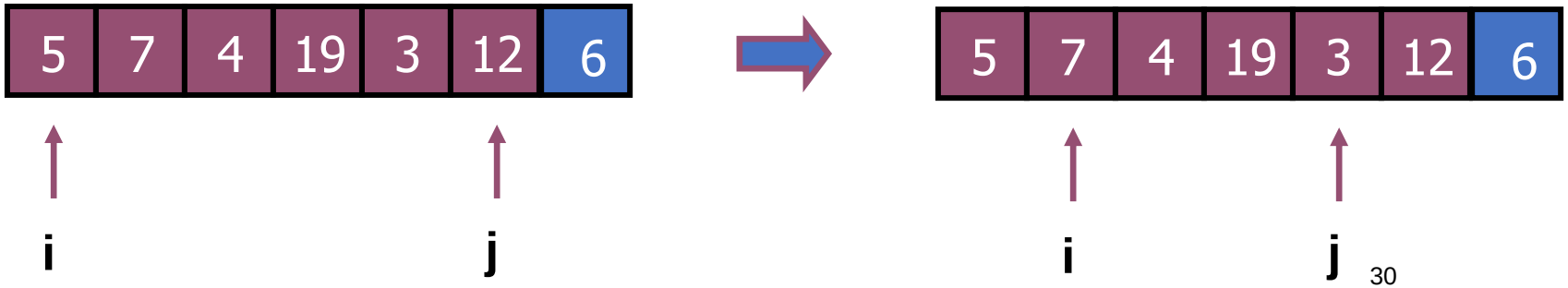
- For now, assume that pivot = A[right].
- We want to partition array A[left .. right].
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)



Partitioning Strategy

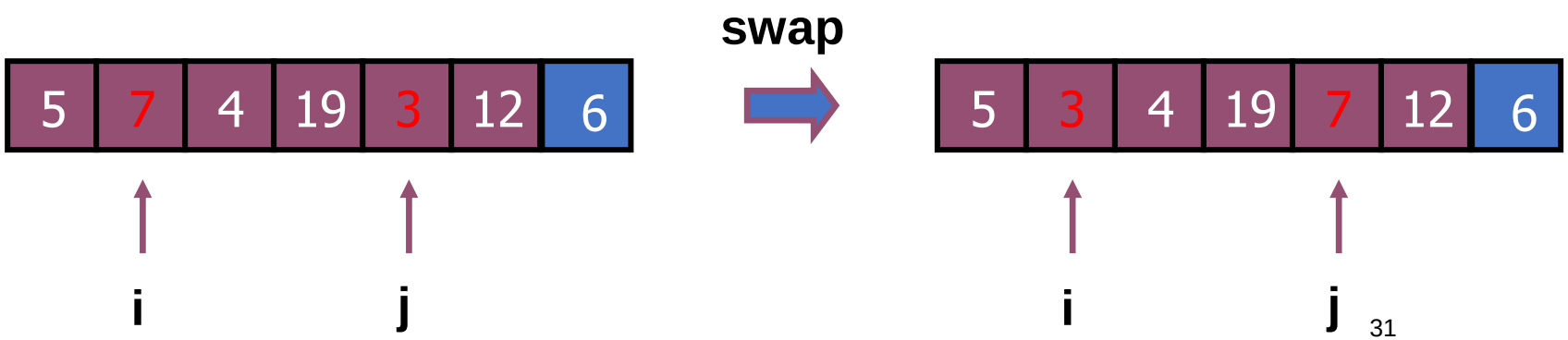
- Want to have
  - A[k] ≤ pivot, for k < i
  - A[k] ≥ pivot, for k > j
- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] and A[j] should now be swapped





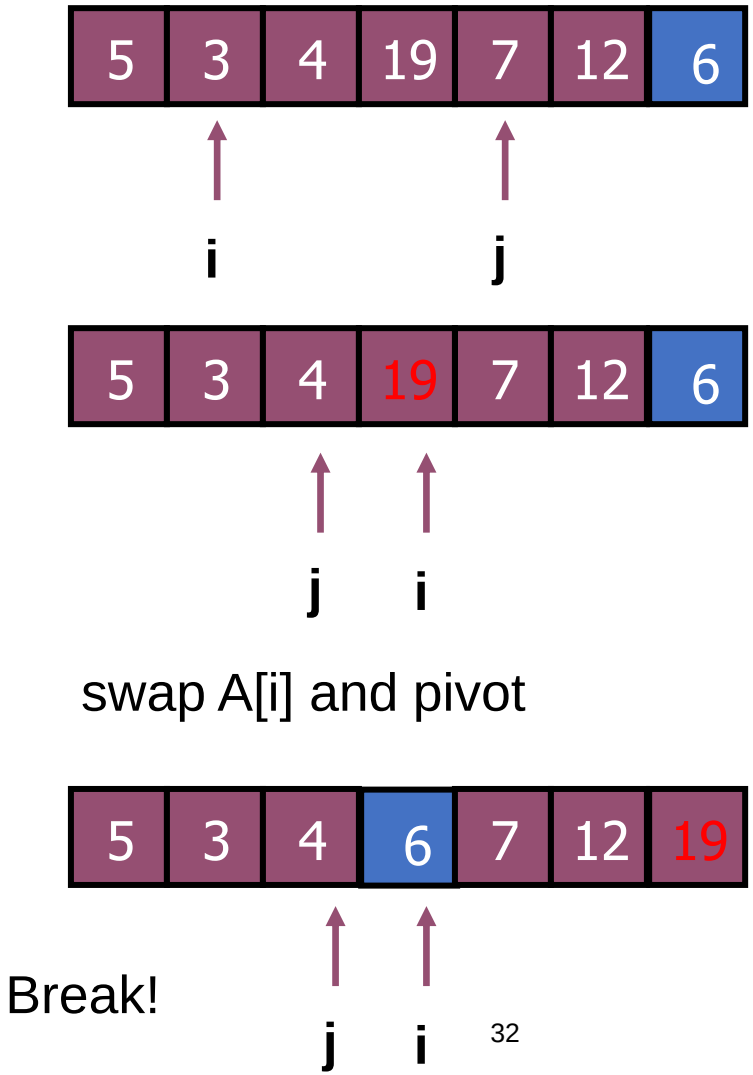
Partitioning Strategy (2)

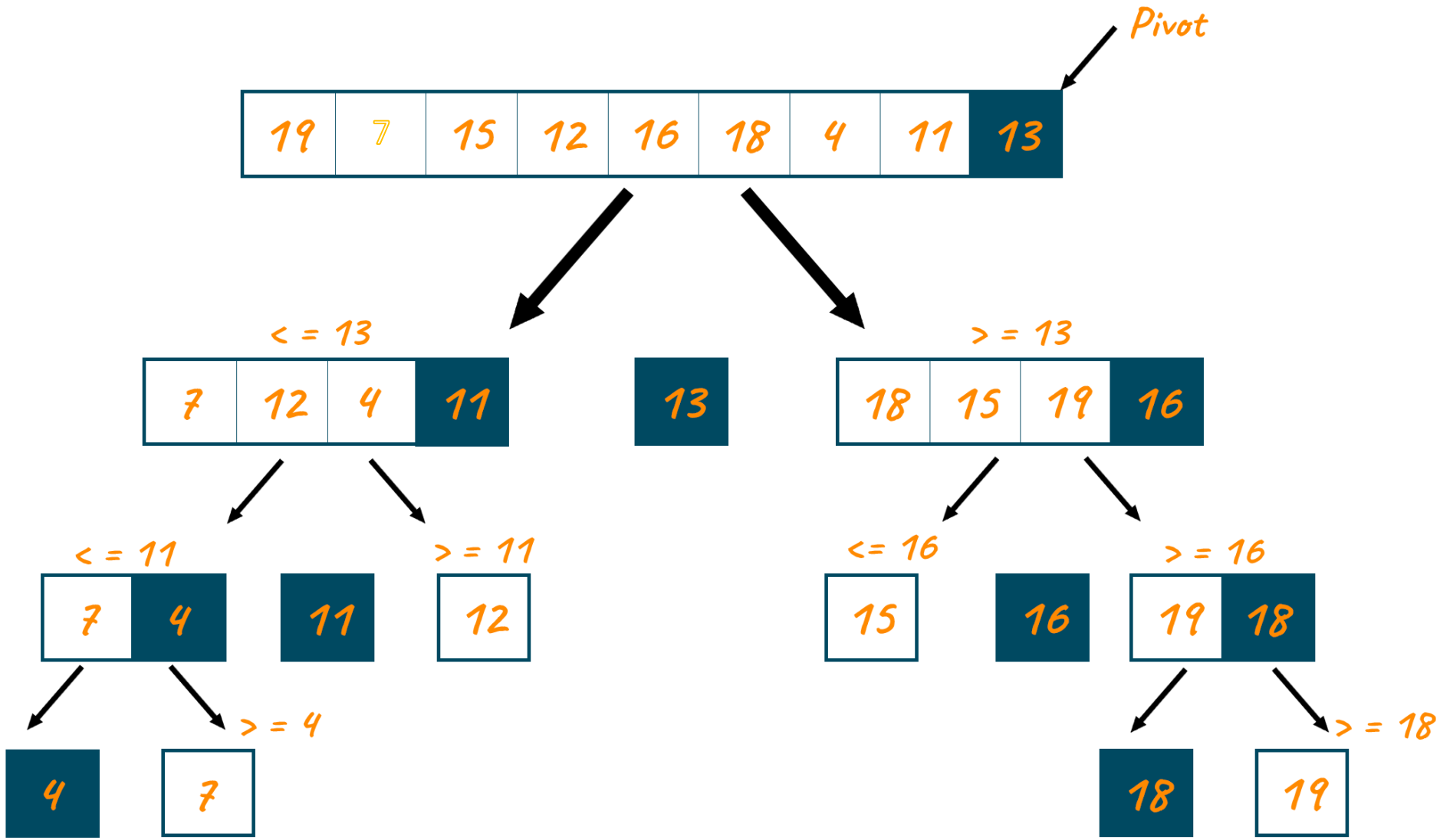
- When i and j have stopped and i is to the left of j (thus legal)
  - Swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - A[i] ≤ pivot
    - A[j] ≥ pivot
  - Repeat the process until i and j cross



Partitioning Strategy (3)

- When i and j have crossed
  - swap A[i] and pivot



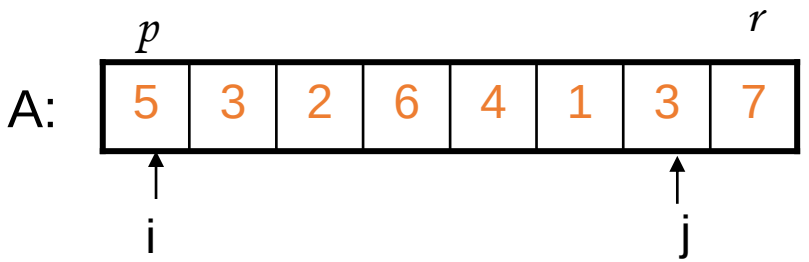

Pivot
19 7 15 12 16 18 4 11 13
< = 13
7 12 4 11
13
> = 13
18 15 19 16
< = 11
7 4
11
> = 11
12
<= 16
15
16
> = 16
19 18
> = 4
4
7
> = 18
18
19

Partitioning the Array

Alg. PARTITION (A, p, r)

1. pivot← $A[r]$
2. i ← p
3. j ← r -1
4. **while** i<j
5. **while j>p and** $A[j]$ > pivot
   j ← j - 1
   While i<r and A[i] <= pivot
   i ← i + 1
   if i < j
6. **then** exchange $A[i] \leftrightarrow A[j]$
7. if A[i]>pivot
   A[i] ↔ $A[r]$

Return i



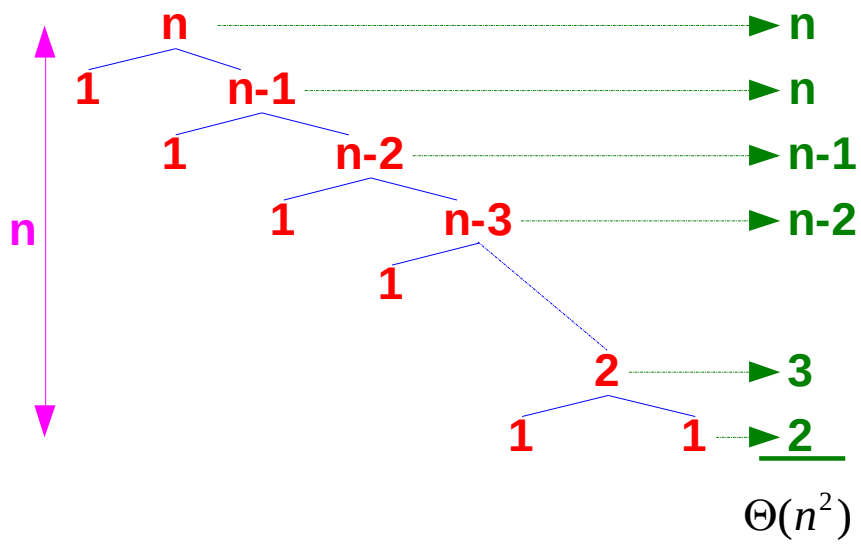
Worst-Case Partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one region with ***n-1*** elements and one with only ***1*** element. Let us assume that this unbalanced partitioning arises at every step of the algorithm. Since partitioning costs $\Theta(n)$ time and $T(1) = \Theta(1)$, the recurrence for the running time is :

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

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

Example: Worst-Case Partitioning



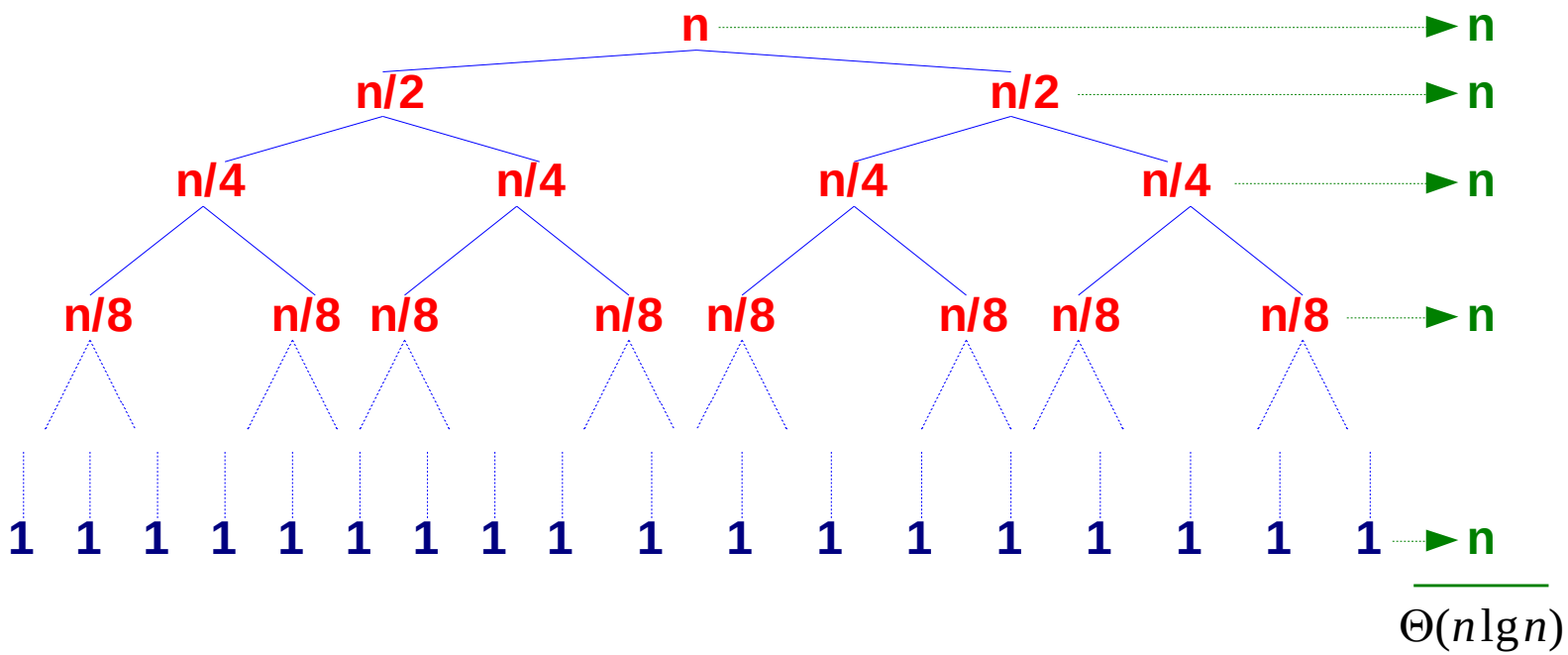
$\Theta(n^2)$

Best-case Partitioning

If the partitioning procedure produces two regions of size ***n/2***, quicksort ***runs much faster***. The recurrence is then:

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

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



Thank You