Merge Sort and Insertion Sort

George Bebis (UNR)

Farrukh Salim Shaikh

Merge Sort

Divide-and-Conquer

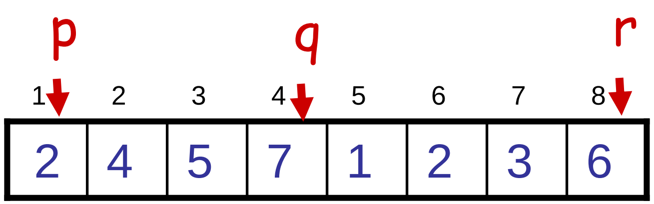
- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

Merge Sort Approach

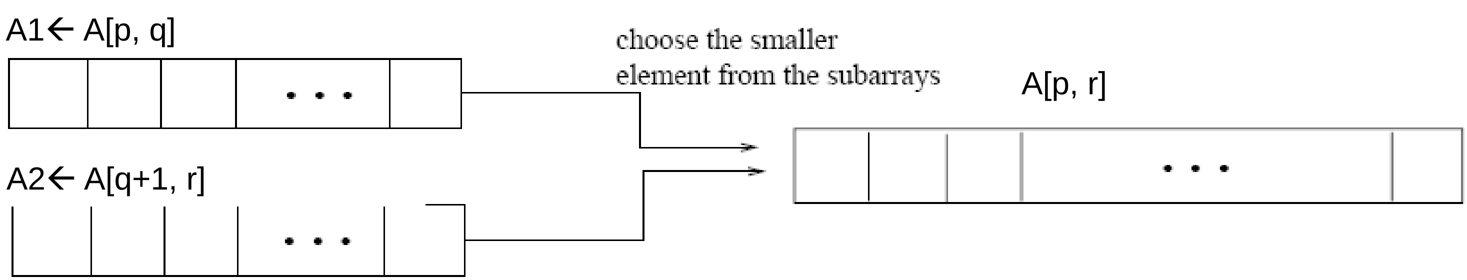
- To sort an array $A[p \,..\, r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge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

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

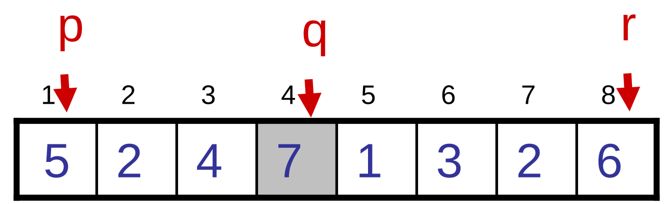


Merging

p			q				r
1	2	3	4	5	6	7	8
2	4	5	7	1	2	3	6

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

Merge Sort



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

if $p < r$ ▷ Check for base case

then $q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

MERGE-SORT(A, p, q) ▷ Conquer

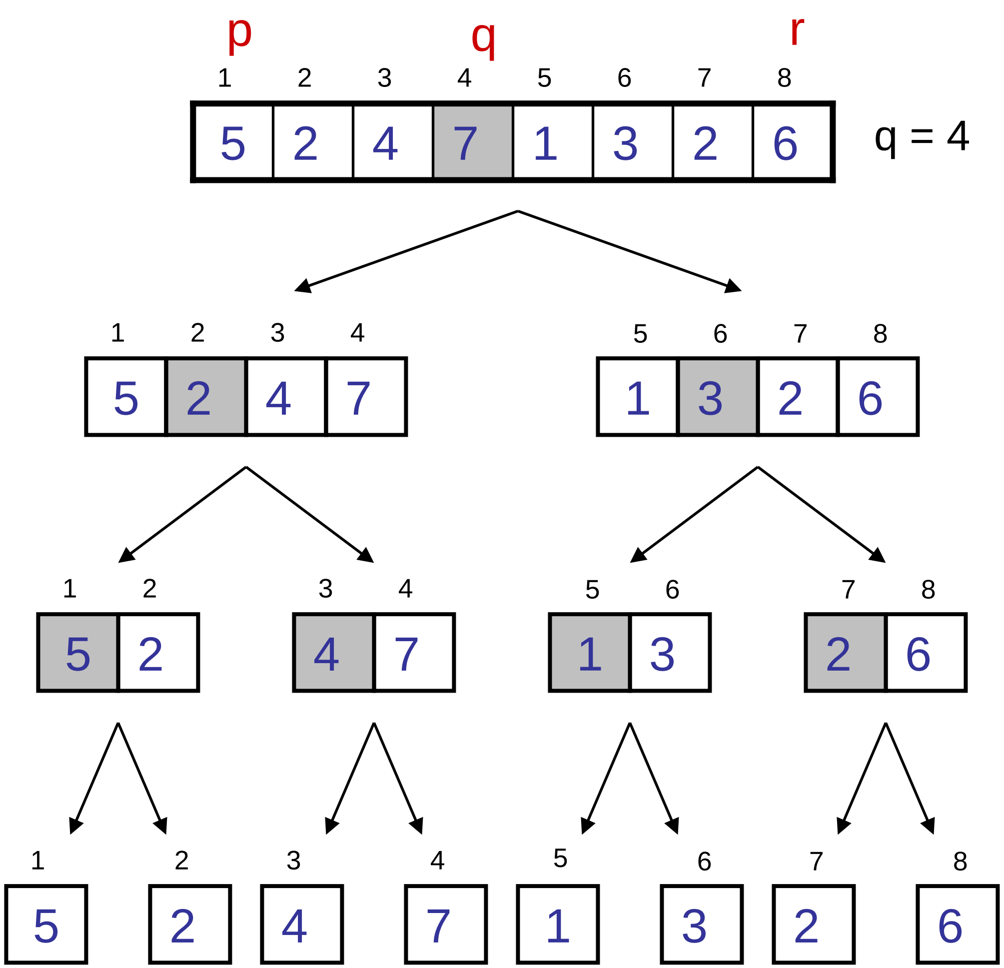
MERGE-SORT(A, $q + 1$, r) ▷ Conquer

MERGE(A, p, q, r) ▷ Combine

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

Example – n Power of 2

Divide



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

if p < r

then $q \leftarrow \lfloor (p + r)/2 \rfloor$

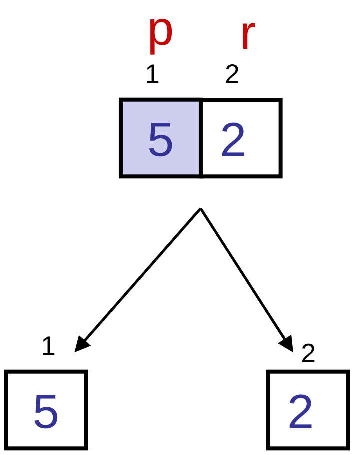
MERGE-SORT(A, p, q)

MERGE-SORT(A, q + 1, r)

MERGE(A, p, q, r)

Example – n Power of 2

Divide



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

if p < r

then $q \leftarrow \lfloor (p + r)/2 \rfloor$

MERGE-SORT(A, p, q)

MERGE-SORT(A, q + 1, r)

MERGE(A, p, q, r)

Alg.: MERGE-SORT(A, 1, 2)

if 1 < 2

then $q \leftarrow \lfloor (1 + 2)/2 \rfloor$

MERGE-SORT(A, 1, 1)

MERGE-SORT(A, 2, 2)

MERGE(A, 1, 1, 2)

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

	p			q				r
	1	2	3	4	5	6	7	8
A	2	4	5	7	1	2	3	6

$\underbrace{2\;4\;5\;7}_{n_1}\;\underbrace{1\;2\;3\;6}_{n_2}$

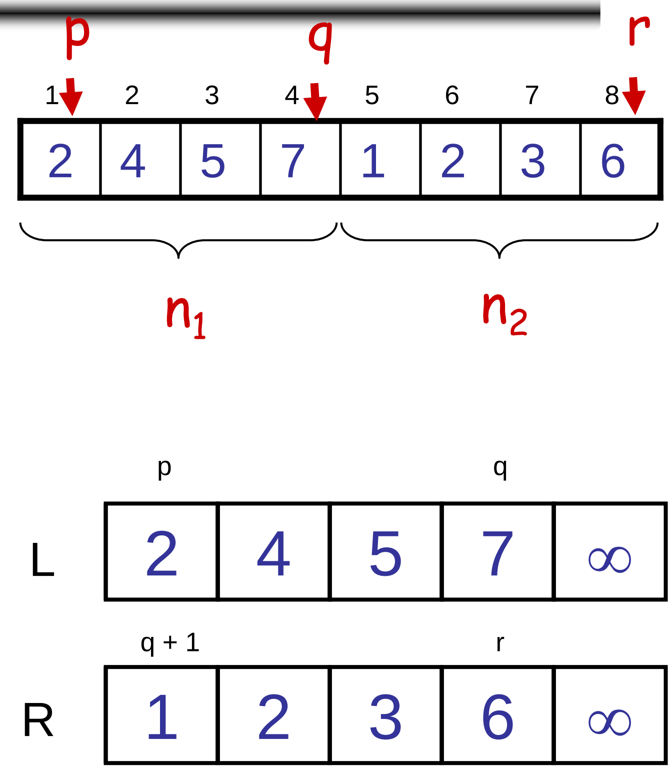
	p			q	
L	2	4	5	7	∞

	q + 1			r	
R	1	2	3	6	∞

Merge - Pseudocode

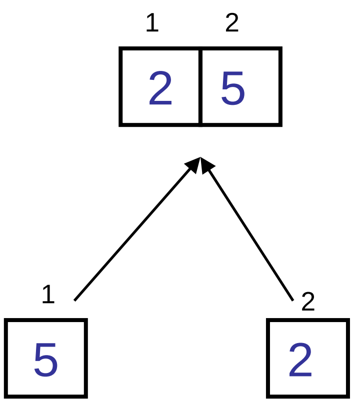
Alg.: MERGE(A, p, q, r)

```
1.  n1 = q - p + 1 and n2 = r - q
2.  create arrays L[1 . . n1 + 1]; R[1 . . n2 + 1]
      For i ← 1 to n1      do L[i] ← A[p+i-1]
      For j ← 1 to n2      do L[j] ← A[q+1]
3.  L[n1 + 1] ← ∞;      R[n2 + 1] ← ∞
4.   i ← 1;    j ← 1
5.   for k ← p to r
6.         do if L[ i ] ≤ R[ j ]
7.              then A[k] ← L[ i ]
8.                     i ← i + 1
9.              else A[k] ← R[ j ]
10.                    j ← j + 1
```



Example – n Power of 2

Merge



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

1. $n_1 = q - p + 1$ and $n_2 = r - q$
2. create arrays $L[1 \,..\, n_1 + 1]$; $R[1 \,..\, n_2 + 1]$

   For $i \leftarrow 1$ to n_1 do $L[i] \leftarrow A[p+i-1]$

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

Alg.: MERGE(A, 1, 1, 2)

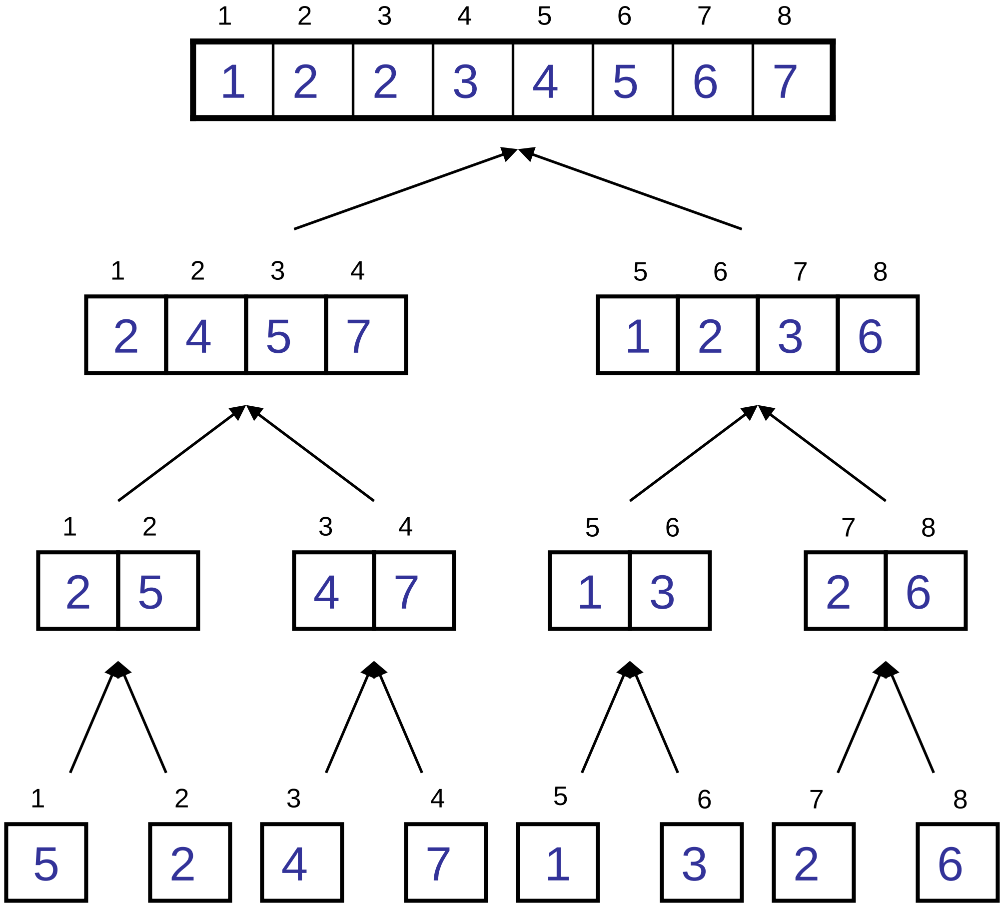
1. $n_1 = 1$ and $n_2 = 1$
2. create arrays $L[1 \,..\, 2]$; $R[1 \,..\, 2]$

   For $i \leftarrow 1$ to 1 do $L[1] \leftarrow A[1]$

   For $j \leftarrow 1$ to 1 do $L[j] \leftarrow A[2]$
3. $L[2] \leftarrow \infty$; $R[2] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow 1$ **to** 2
6. **do if** $L[\,1\,] \leq R[\,1\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[1] \leftarrow R[\,1\,]$
10. $j \leftarrow 2$
11. **do if** $L[\,1\,] \leq R[\,2\,]$
12. **then** $A[2] \leftarrow L[\,1\,]$
13. $i \leftarrow 2$
14. **else** $A[k] \leftarrow R[\,j\,]$
15. $j \leftarrow j + 1$

Example – n Power of 2

Conquer and Merge



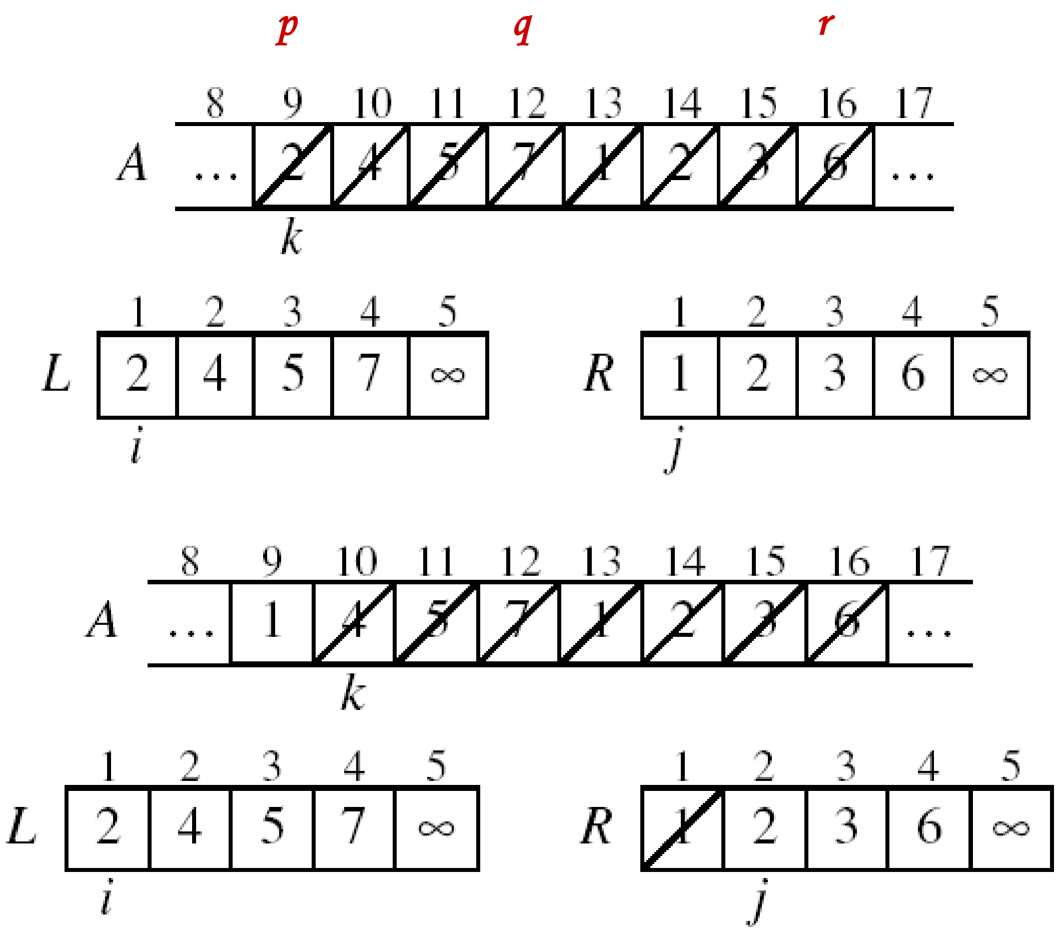
Alg.: MERGE(A, p, q, r)

1. $n_1 = q - p + 1$ and $n_2 = r - q$
2. create arrays $L[1 \,.\,.\, n_1 + 1]$; $R[1 \,.\,.\, n_2 + 1]$

   For $i \leftarrow 1$ to n_1 do $L[i] \leftarrow A[p+i-1]$

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

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



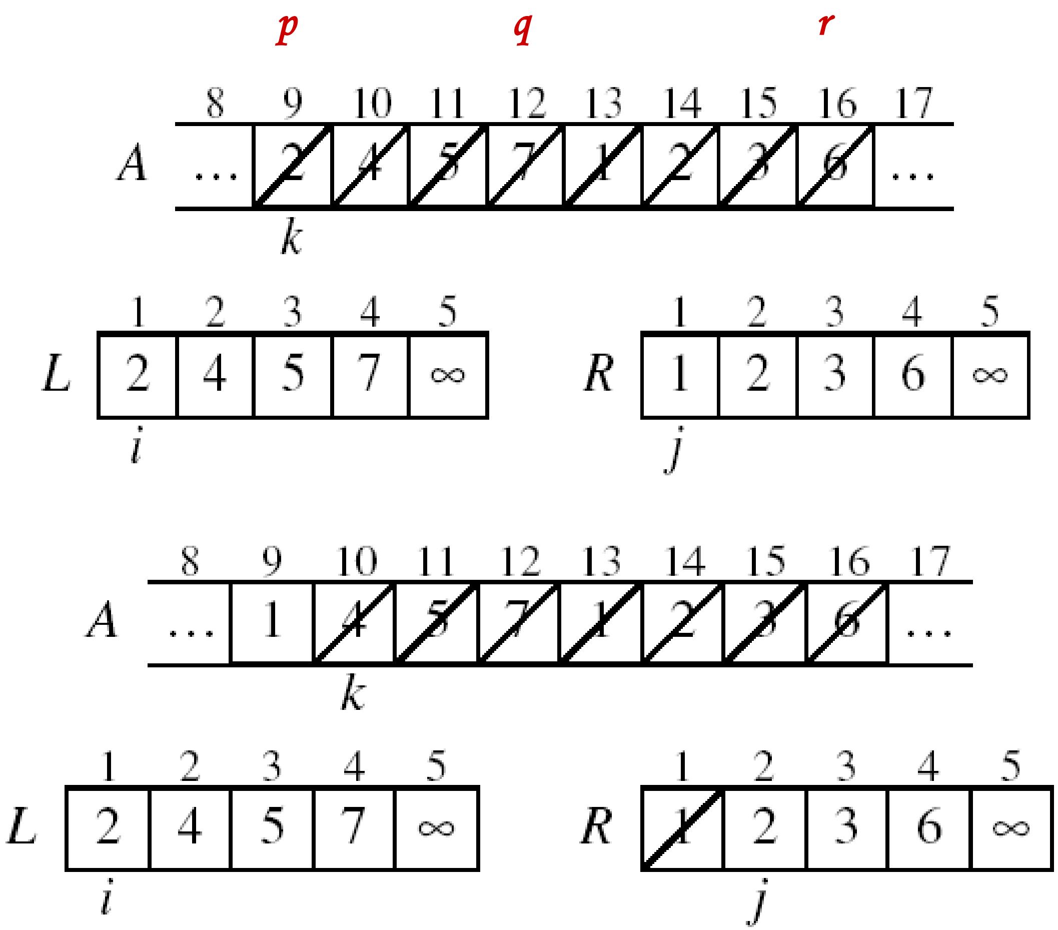
Alg.: MERGE(A, p, q, r)

1. $n_1 = q - p + 1$ and $n_2 = r - q$
2. create arrays $L[1 \,..\, n_1 + 1]$; $R[1 \,..\, n_2 + 1]$

   For $i \leftarrow 1$ to n_1 do $L[i] \leftarrow A[p+i-1]$

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

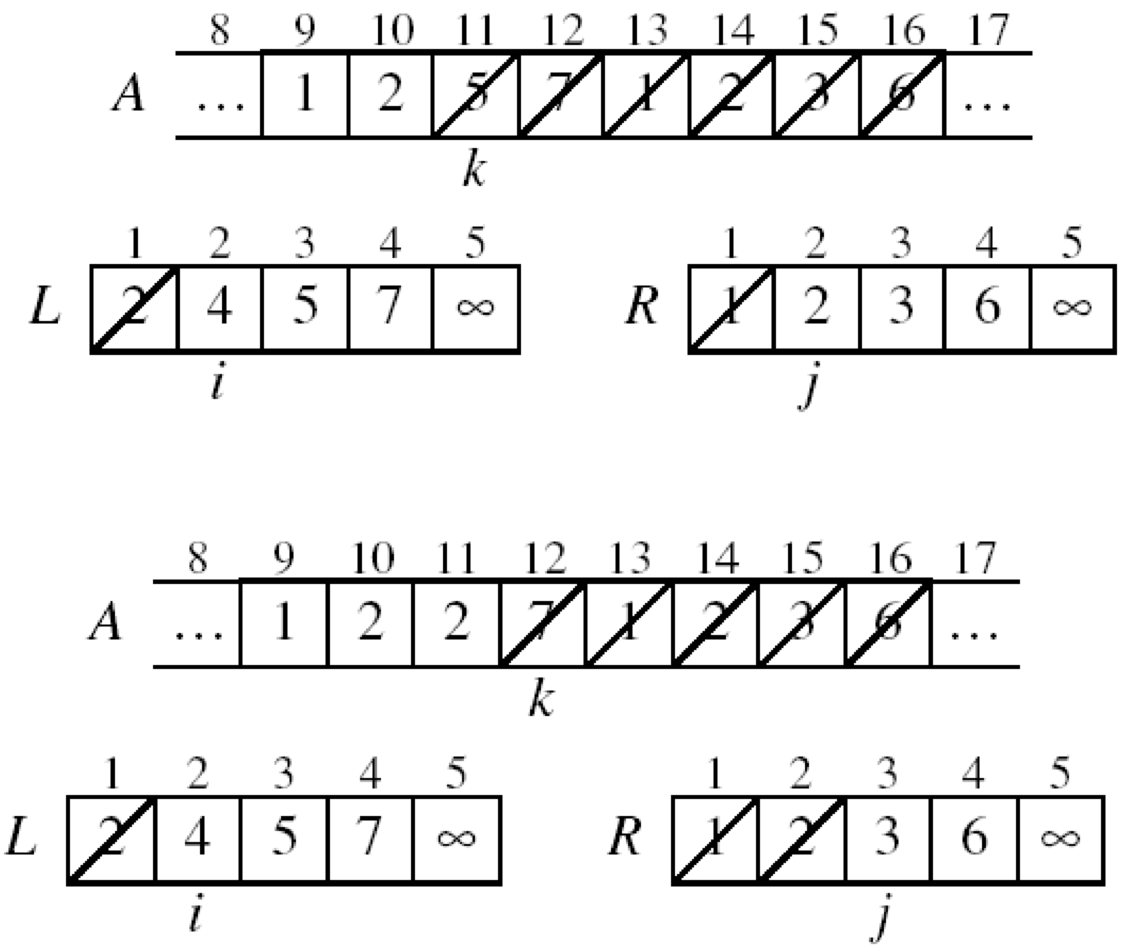
Example: MERGE(A, 9, 12, 16)



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

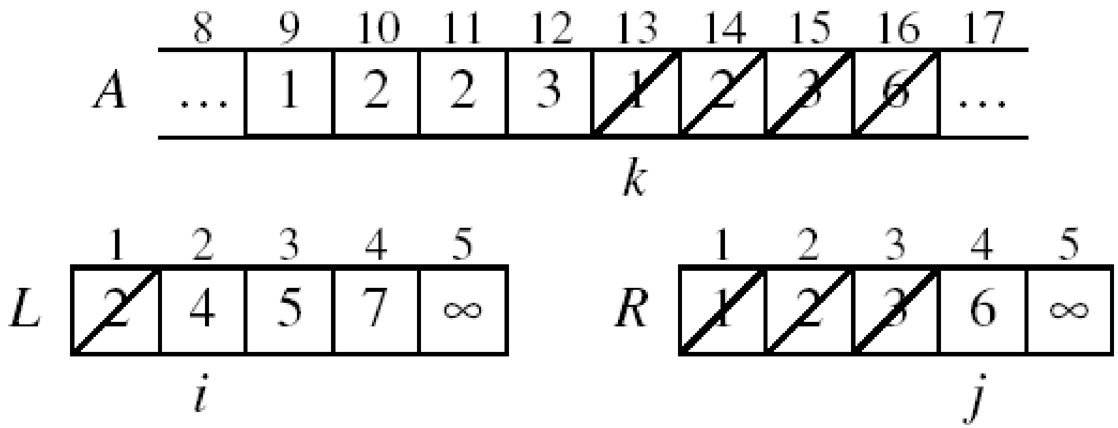
Example: MERGE(A, 9, 12, 16)

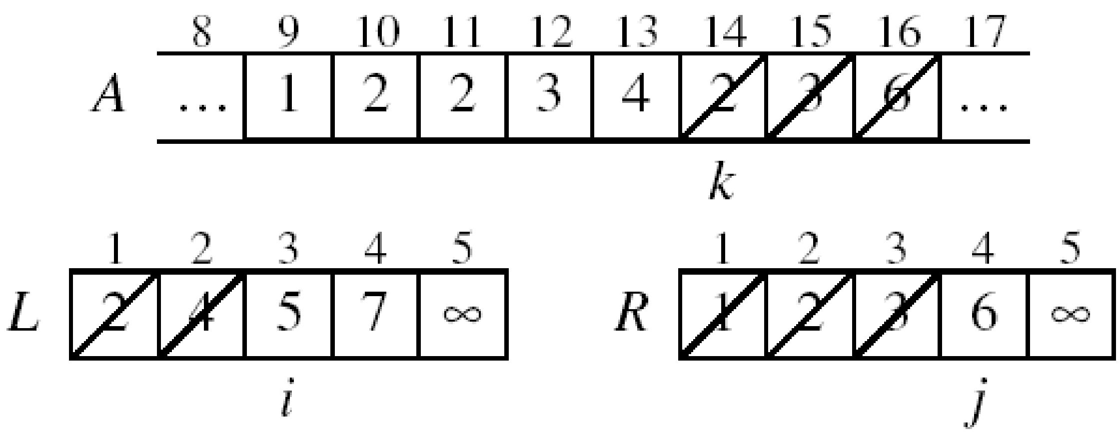


Alg.: MERGE(A, p, q, r)

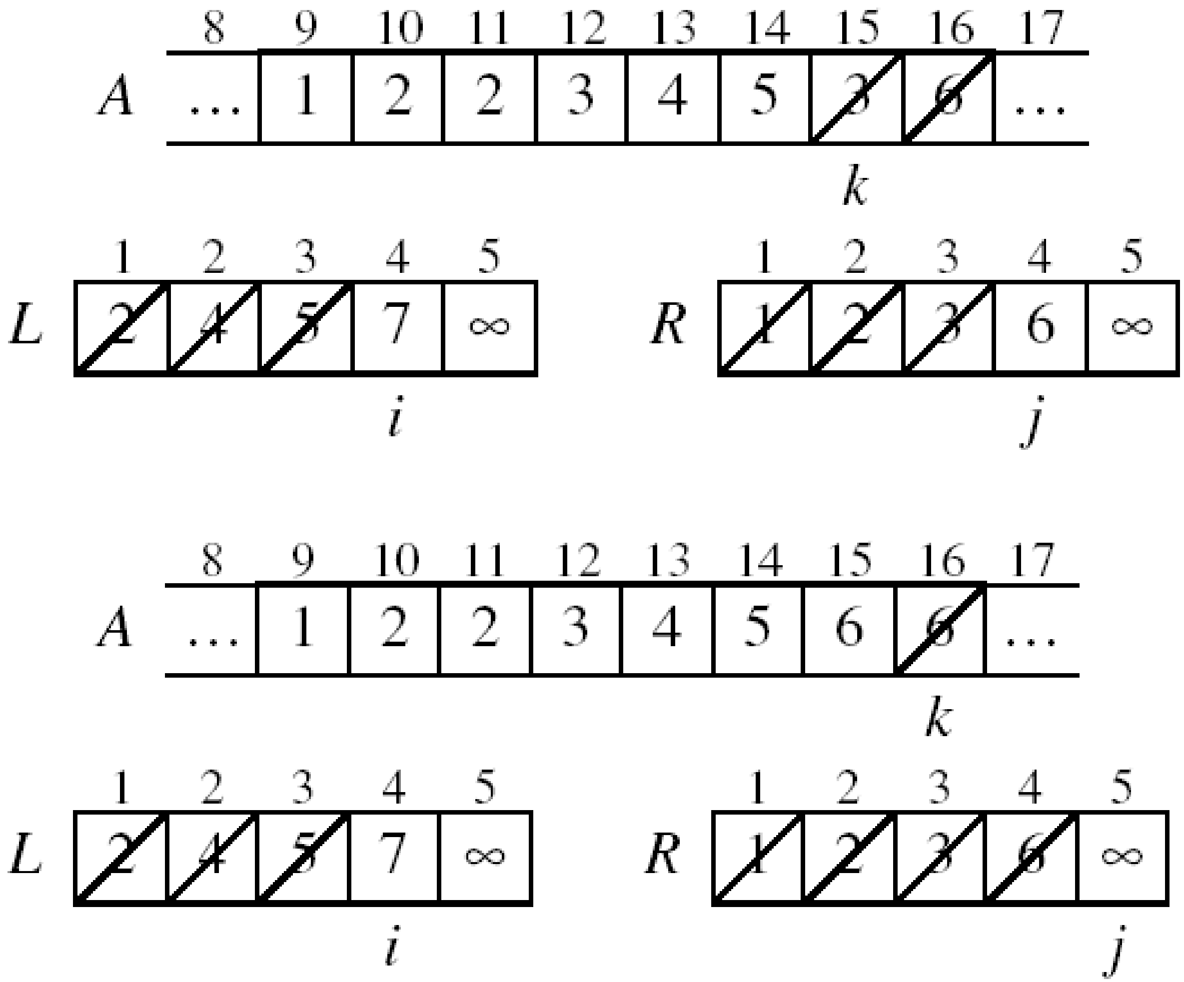
1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \,..\, n_1 + 1]$ and the next n_2 elements into $R[1 \,..\, n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\,i\,] \le R[\,j\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\,j\,]$
10. $j \leftarrow j + 1$

Example (cont.)

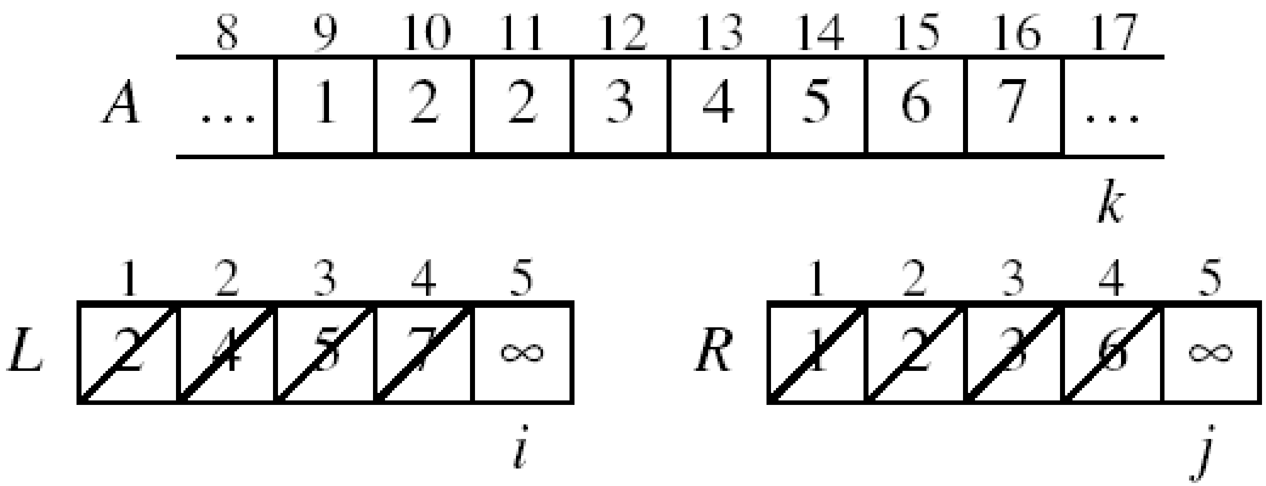




Example (cont.)



Example (cont.)



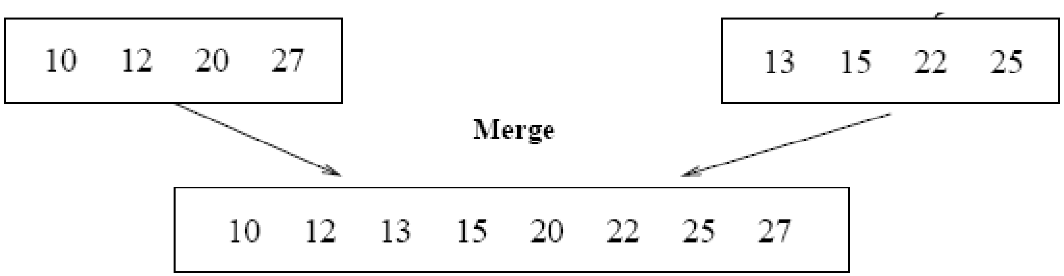
Done!

Running Time of Merge (assume last **for** loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$

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



Analyzing Divide-and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size n
  - **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
  - **Conquer** (solve) the subproblems $aT(n/b)$
  - **Combine** the solutions $C(n)$

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

MERGE-SORT Running Time

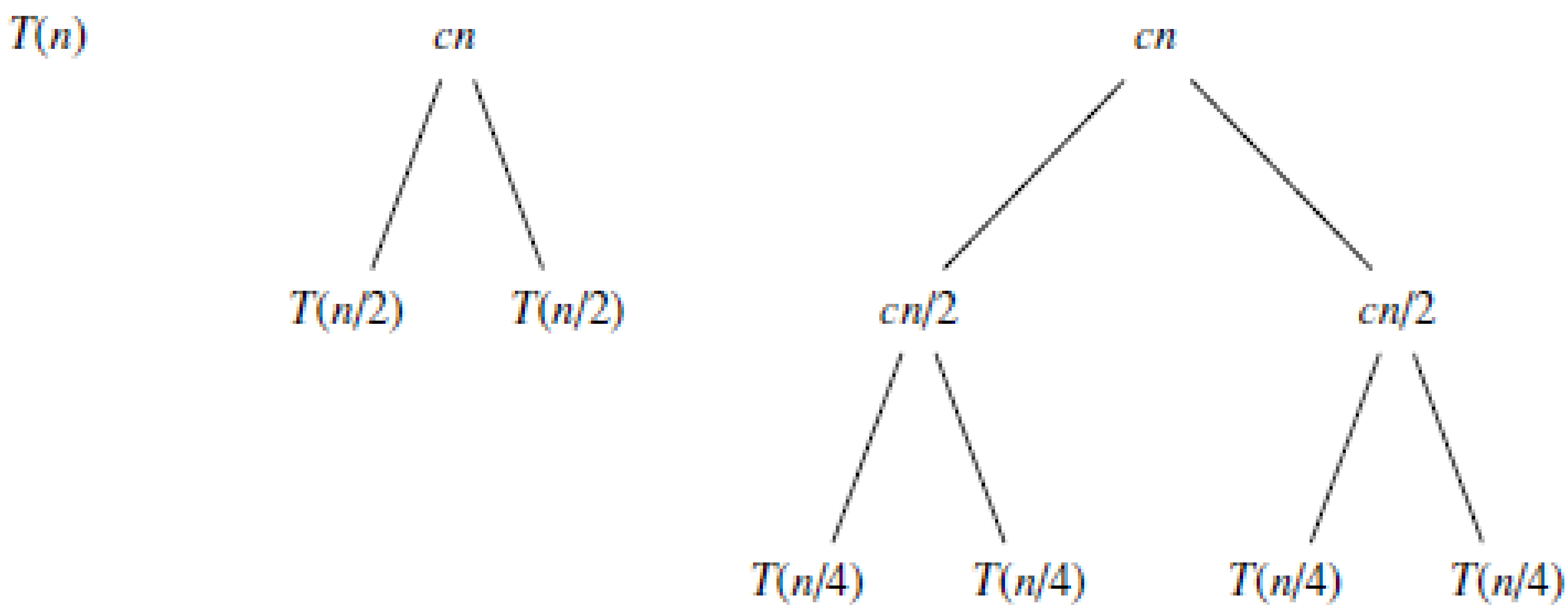
- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2$
    $\Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$

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

Solve the Recurrence

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

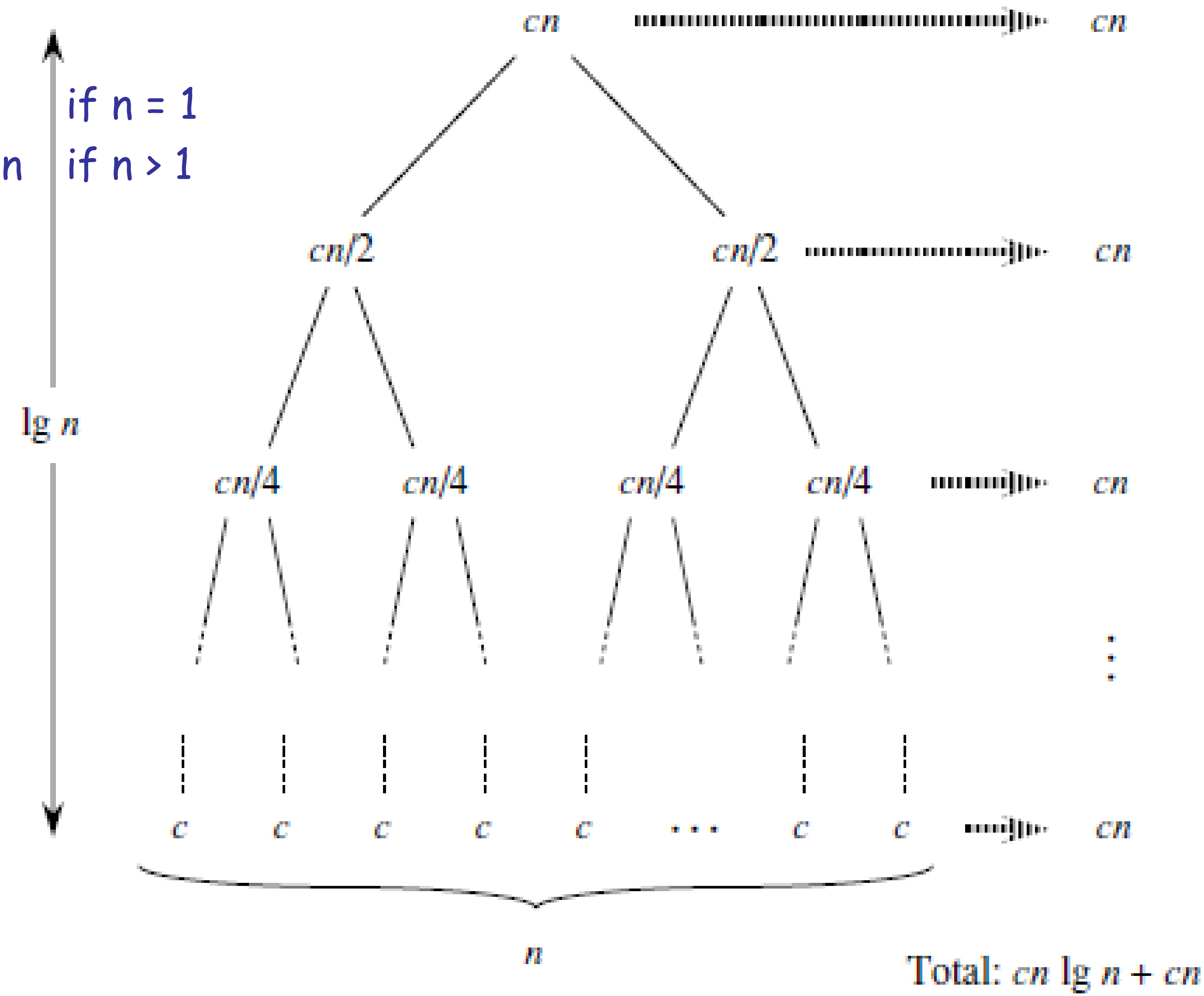
Recurrence Tree method



Solve the Recurrence

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

Recurrence Tree method



Solve the Recurrence

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

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

Substitution method

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

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

$T(n/2) = n/2 + 2T(n/4)$

$T(n/4) = n/4 + 2T(n/4/2)$

$T(n/4) = n/4 + 2T(n/8)$

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

$T(n) = n + 2(n/2+2T(n/4))$

$T(n) = n + 2n/2 + 4T(n/4)$

$T(n) = n + 2n/2 + 4(n/4 + 2T(n/8))$

$T(n) = n + n + n + 8T(n/8)$

$T(n) = 3n + 8(n/8 + 2T(n/16))$

$T(n) = 4n + 16T(n/2\text{^}4)$

.....

$T(n) = kn + 2\text{^}k.T(n/2\text{^}k)$

For $k = \log n \Rightarrow n = 2\text{^}k$

$T(n) = n.\log n + T(1)$

$T(n) = O(n\log n)$

Merge Sort - Discussion

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

Loop invariant for Merge Sort

- For a while, skip the loop invariant property of Merge sort in the next slides.
- After going through the loop invariant property of Insertion sort, it will then be easier to understand this.

Correctness of MergeArray

- Loop-invariant
  - At the start of each iteration of the **for** loop, the subarray *A*[*1:k-1*] contains the *k-1* smallest elements of *L*[*1:n1*] and *R*[*1:n2*] in sorted order. Moreover, *L*[*i*] and *R*[*j*] are the smallest elements of their arrays that have not been copied back to *A*

Inductive Proof of Correctness

- **Initialization:** (the invariant is true at beginning)

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

Inductive Proof of Correctness

- **Maintenance:** (the invariant is true after each iteration)

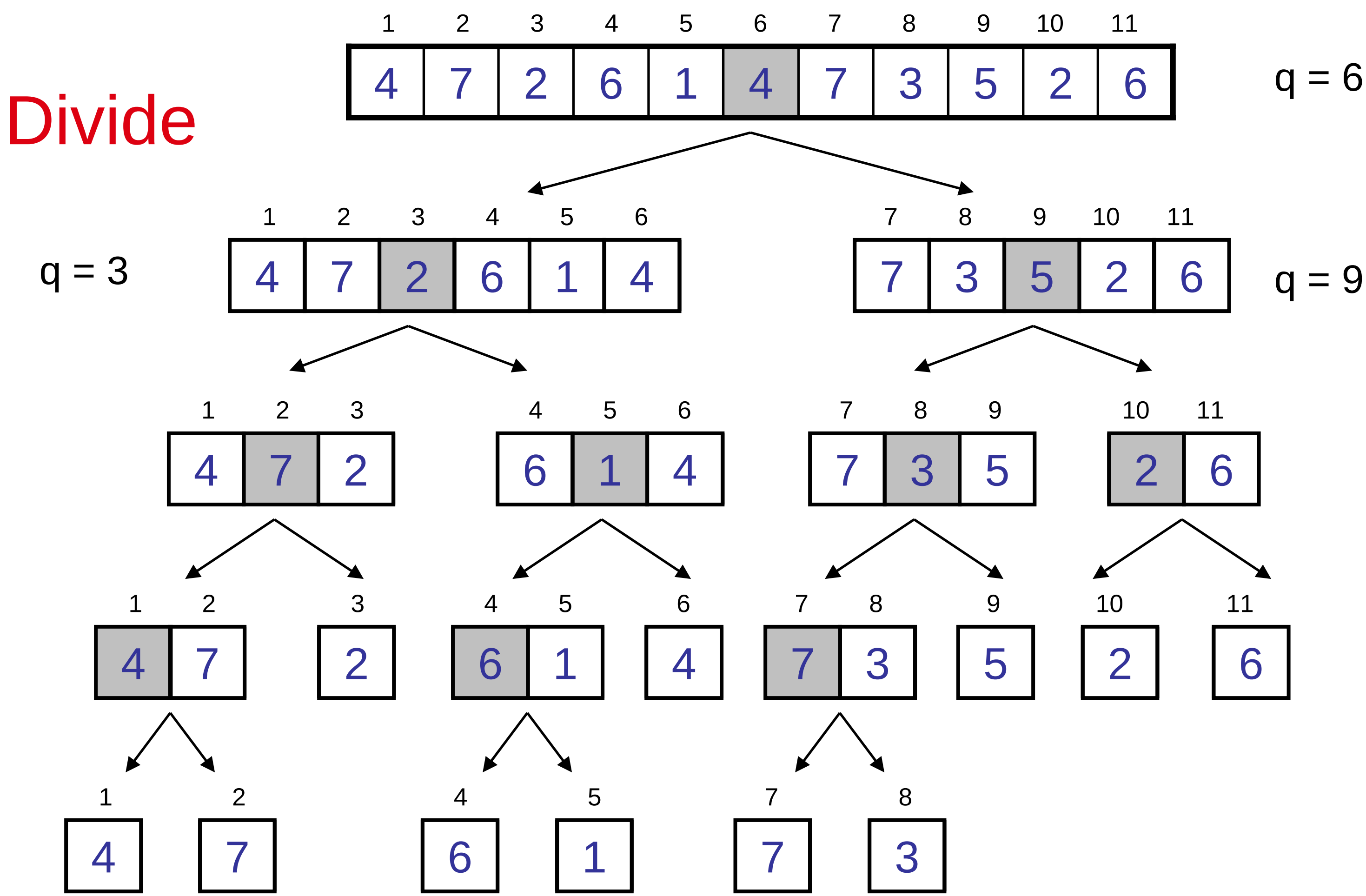
  assume $L[i] <= R[j]$, the $L[i]$ is the smallest element not yet copied back to A. Hence after copy $L[i]$ to $A[k]$, the subarray $A[1..k-1]$ contains the k smallest elements. Increasing k and i by 1 reestablishes the loop invariant for the next iteration.

Inductive Proof of Correctness

- **Termination:** (loop invariant implies correctness)

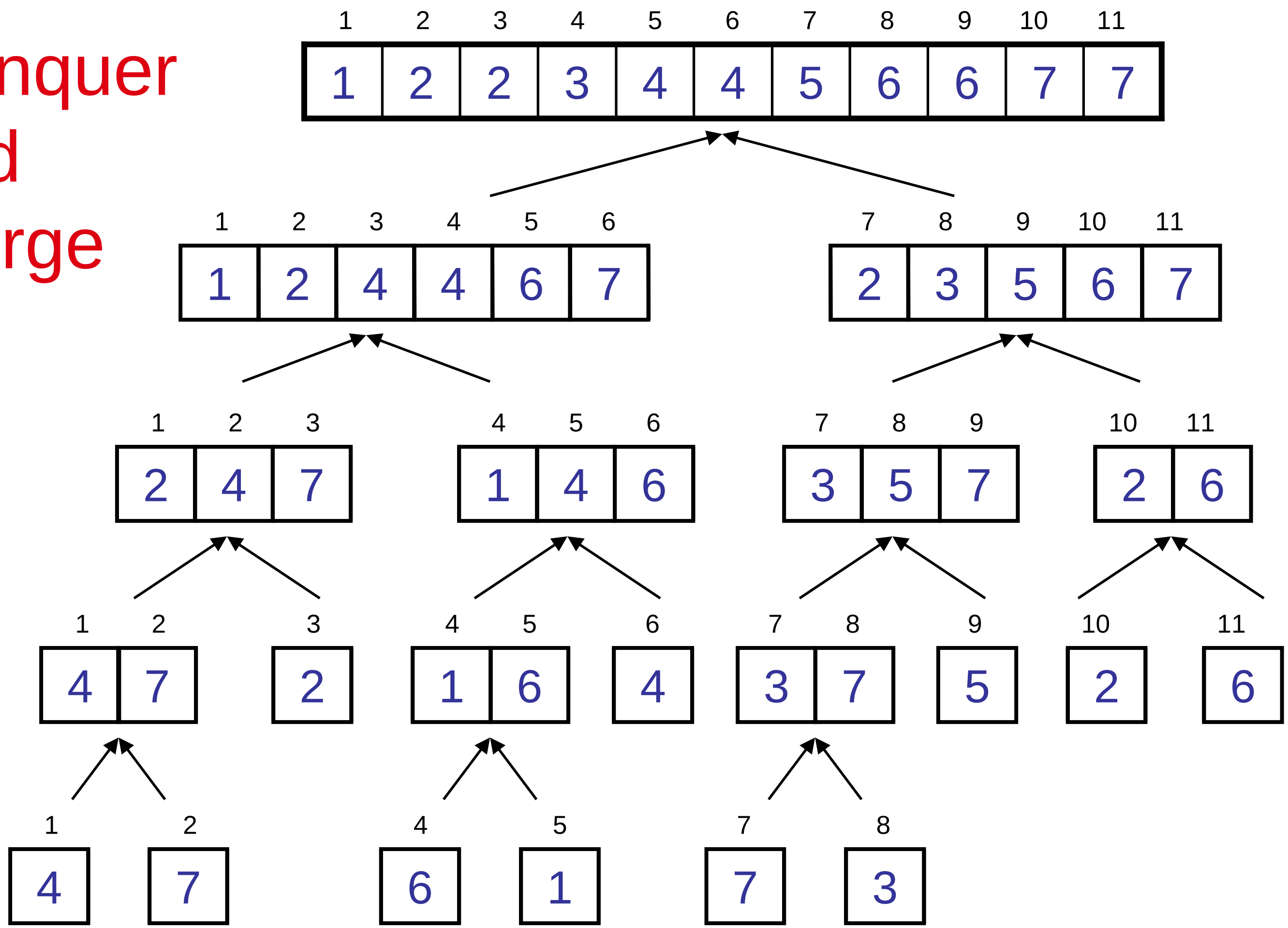
  At termination we have k - 1 = n1 + n2, by the loop invariant, we have A contains the k -1 (n1 + n2) smallest elements of L and R in sorted order.

Example – n Not a Power of 2



Example – n Not a Power of 2

Conquer and Merge



Solve the Recurrence (Practice example)

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

Substitution method

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

$T(n) = n + (n/2+T(n/4))$

$T(n) = n + n/2 + T(n/4)$

$T(n) = n + n/2 + n/4 + T(n/8))$

$T(n) = n + n/2 + n/4 + T(n/8)$.

$T(n) =n (1 + 1/2 + 1/4) + T(n/8)$

.....

$T(n) = n(1 + ½ + ¼ \ldots\ldots.. + 1/k) + T(n/2k)$

For $k = n$

$T(n) = n(1+ ½ + ¼ + \ldots.. + 1/n) + T(1/2)$

// Assume $T(1/2) = 1$

$T(n) = n.(1+1) + 1$

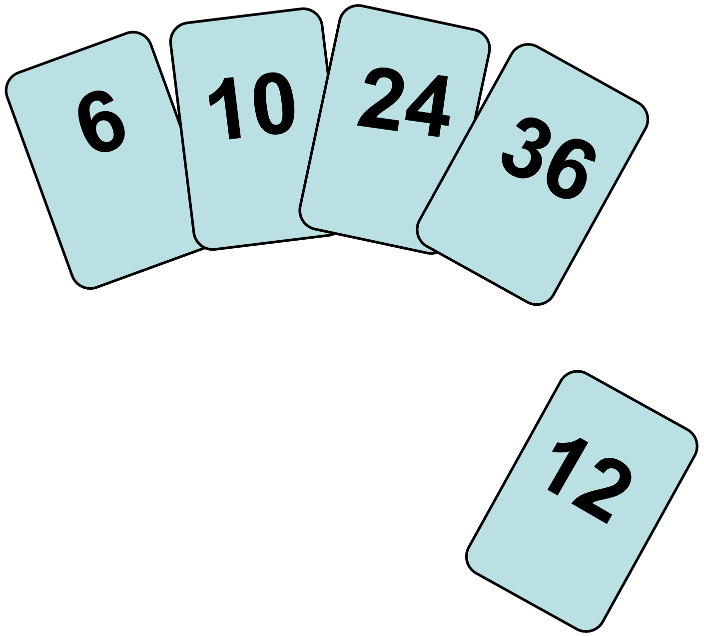
$T(n) = \text{Theta}(n)$

Insertion Sort

Insertion Sort

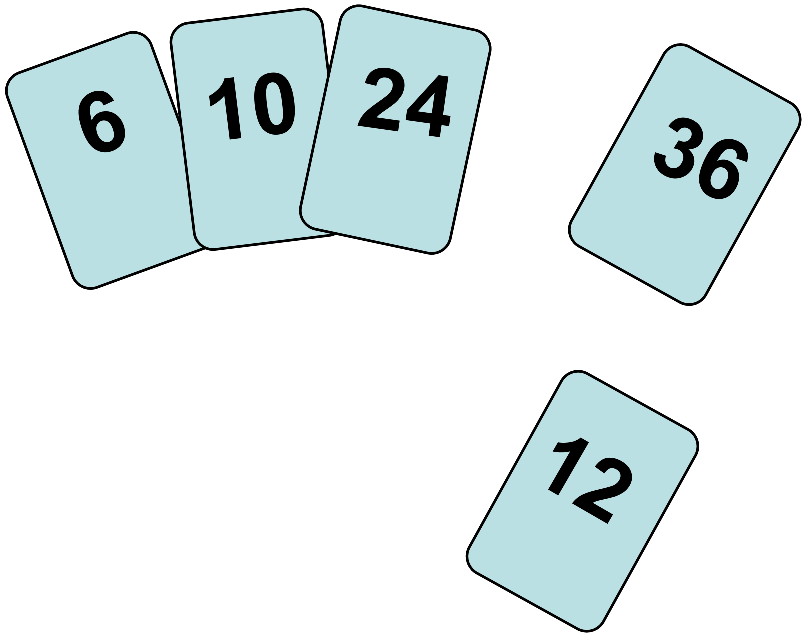
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

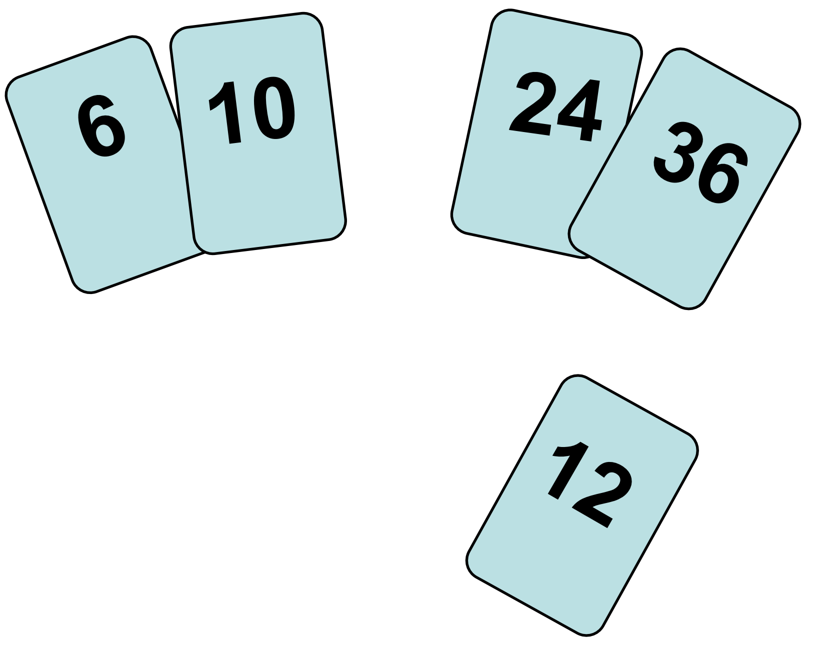


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array

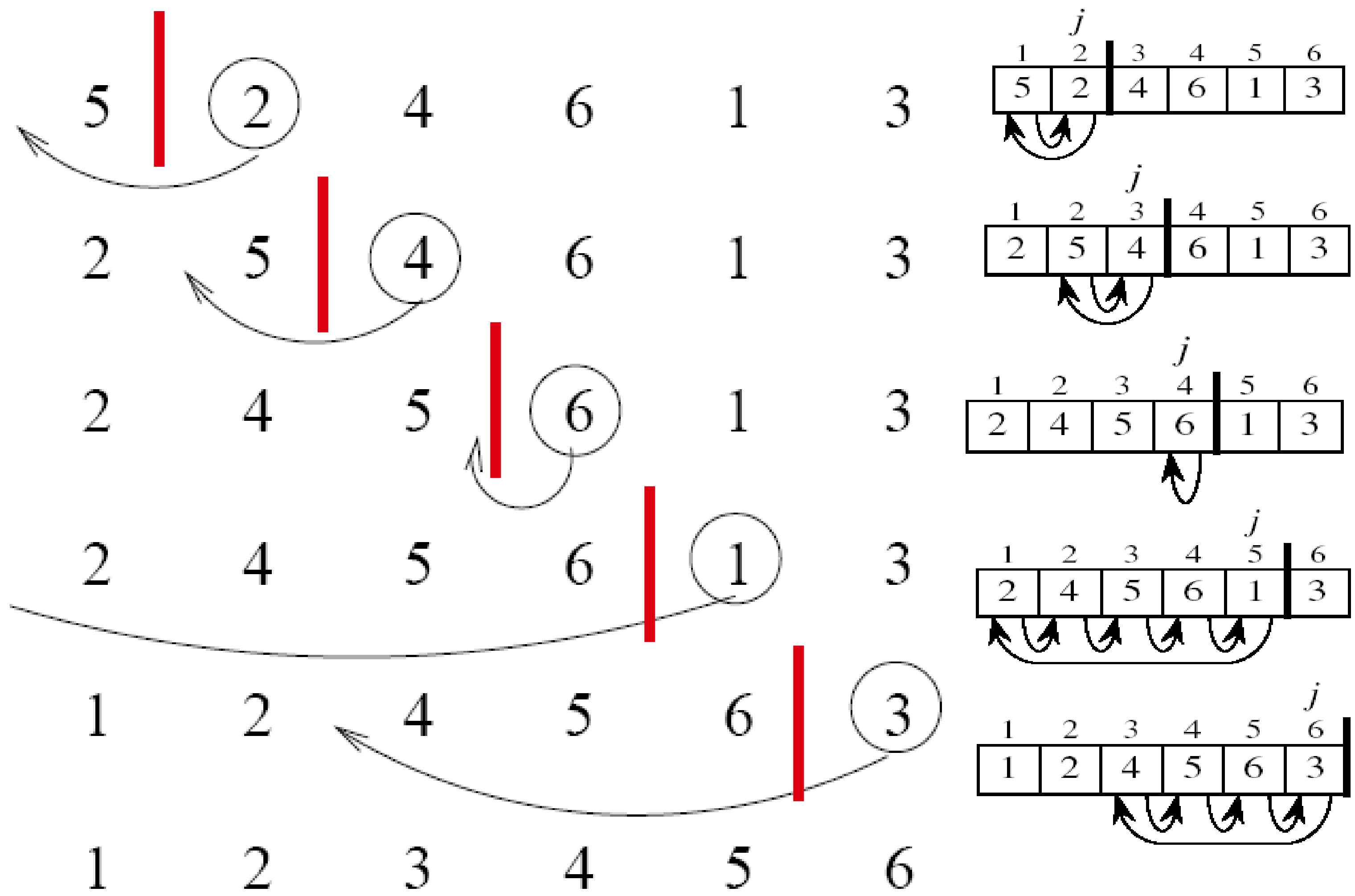
right sub-array

2 5 | 4 6 1 3

sorted

unsorted

Insertion Sort



INSERTION-SORT

Alg.: INSERTION-SORT(*A*)

1	2	3	4	5	6	7	8
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

key

```
for j ← 2 to n
    do key ← A[ j ]
        ▷Insert A[ j ] into the sorted sequence A[1 . . j -1]
        i ← j - 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i - 1
        A[i + 1] ← key
```

- Insertion sort – sorts the elements in place

Proving Loop Invariants

- A loop invariant is a statement about program variables that is true before and after each iteration of a loop.
- **Initialization (base case):**
  - It is true prior to the first iteration of the loop
- **Maintenance (inductive step):**
  - If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:**
  - When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
  - Stop the induction when the loop terminates

Loop Invariant for Insertion Sort

Alg.: INSERTION-SORT(*A*)

for j ← 2 **to** n

    do key ← *A*[j]

        Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

        i ← j - 1

        while i > 0 and *A*[i] > key

            do *A*[i + 1] ← *A*[i]

                i ← i - 1

        A[i + 1] ← key

Invariant: at the start of the **for** loop the elements in *A*[1 . . j-1] are in sorted order

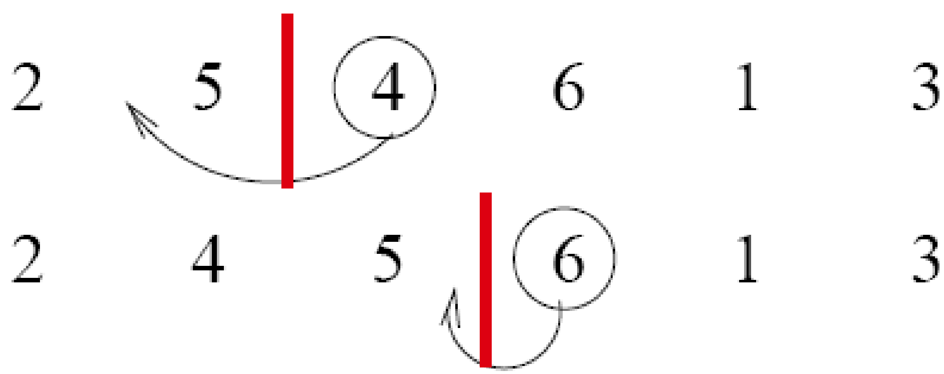
Loop Invariant for Insertion Sort

- **Initialization:**
  - Just before the first iteration, $j = 2$:

    the subarray $A[1 \, . \, . \, j\text{-}1] = A[1]$, (the element originally in $A[1]$) – is sorted

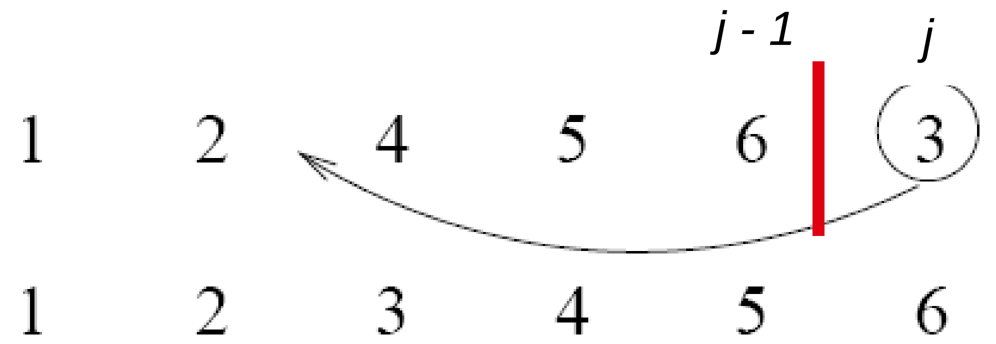
Loop Invariant for Insertion Sort

- **Maintenance:**
  - the **while** inner loop moves $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found
  - At that point, the value of *key* is placed into this position.



Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when $j = n + 1 \Rightarrow j-1 = n$
  - Replace n with $j-1$ in the loop invariant:
    - the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order



- The entire array is sorted!

Invariant: at the start of the **for** loop the elements in $A[1 \,..\, j-1]$ are in sorted order

Analysis of Insertion Sort

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} \left(t_j - 1\right) + c_7 \sum_{j=2}^{n} \left(t_j - 1\right) + c_8(n-1)$$

Best Case Analysis

- The array is already sorted **"while** i > 0 and A[i] > key"
  - $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n-1)$

  $= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

  $= an + b = \Theta(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Worst Case Analysis

- The array is in reverse sorted order **“while** i > 0 and A[i] > key”
  - Always A[i] > key in **while** loop test
  - Have to compare key with all elements to the left of the j-th position ⇒ compare with j-1 elements ⇒ $t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$ => $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ => $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1 ≈ $n^2/2$ comparisons	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1 ≈ $n^2/2$ exchanges	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

Insertion Sort - Summary

- Advantages
  - Good running time for “almost sorted” arrays $\Theta(n)$
- Disadvantages
  - $\Theta(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and exchanges