EL9343

Data Structure and Algorithm

Lecture 3: Divide-and-Conquer algorithms, Introduction to Sorting

Instructor: Yong Liu

Some slides from David Luebke & George Bebis

Last Lecture: Solving Recurrence

- Recursion tree
  - Convert recurrence into a tree
  - Each node represents the cost incurred at various levels of recursion
  - Sum up the costs of all levels

- Substitution method
  - Guess a solution
  - Use induction to prove that the solution works

- Master method

Master's Method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, a ≥ 1, b > 1, and f(n) > 0

- **Case 1:** if $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
- **Case 2:** if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$;
- **Case 3:** if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Today

- Divide-and-conquer algorithms
  - maximum subarray problem
- Introduction to sorting
  - Insertion sort
  - Bubble sort
  - Mergesort

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 ⇒ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem
- Examples: Fibonacci number, binary search

More Divide-and-Conquer Algorithms

- Maximum Subarray: For a given array A[1..n], find the contiguous subarray A[l..r], such that the summation of A[l]+A[l+1]…+A[r] is the maximum among all contiguous subarrays
  - example:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

- Brute-force solution: check all pairs {l,r}, $O(n^2)$

- Divide-and-Conquer:
  - Divide A[1..n] in the middle: A[1,mid], A[mid+1,n]
  - Any subarray A[i,..j] is
    - (1) Entirely in A[1,mid]
    - (2) Entirely in A[mid+1,n]
    - (3) In both
  - (1) and (2) can be found recursively, (3) need to find maximum subarray crossing midpoint: A[i..mid], A[mid+1..j], 1 <= i, j <= n
  - Take subarray with largest sum of (1), (2), (3)

maximum subarray

```
Find-Max-Cross-Subarray(A,low,mid,high)
    left-sum = -∞
    sum = 0
    for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum then
            left-sum = sum
            max-left = i
    right-sum = -∞
    sum = 0
    for j = mid+1 to high
      sum = sum + A[j]
      if sum > right-sum then
            right-sum = sum
            max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Total time: $T(n)=2T(n/2)+\Theta(n)$, $T(n)=\Theta(n\log n)$

The Sorting Problem

- **Input:**
  - A sequence of n numbers $a_1, a_2, . . . , a_n$
- **Output:**
  - A permutation (reordering) $a_1', a_2', . . . , a_n'$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$

Structure of Data

- The numbers to be sorted are part of collection of data called a record
- Each record contains a key, which is the value to be sorted

Example of a record

Key	**Other data**

- Noted that when the key must be arranged, the data associated with the key must also be rearranged (time consuming!)
- Pointer can be used instead (space consuming!)

Why Study Sorting Algorithms?

- Most fundamental problem in algorithm
- Widely encountered in practice
- Rich set of classical sorting algorithms using different techniques
- A variety of situations that we can encounter
  - Do we have randomly ordered keys?
  - Are all keys distinct?
  - How large is the set of keys to be ordered?
  - Need guaranteed performance?
- Certain algorithms are better suited to certain situations

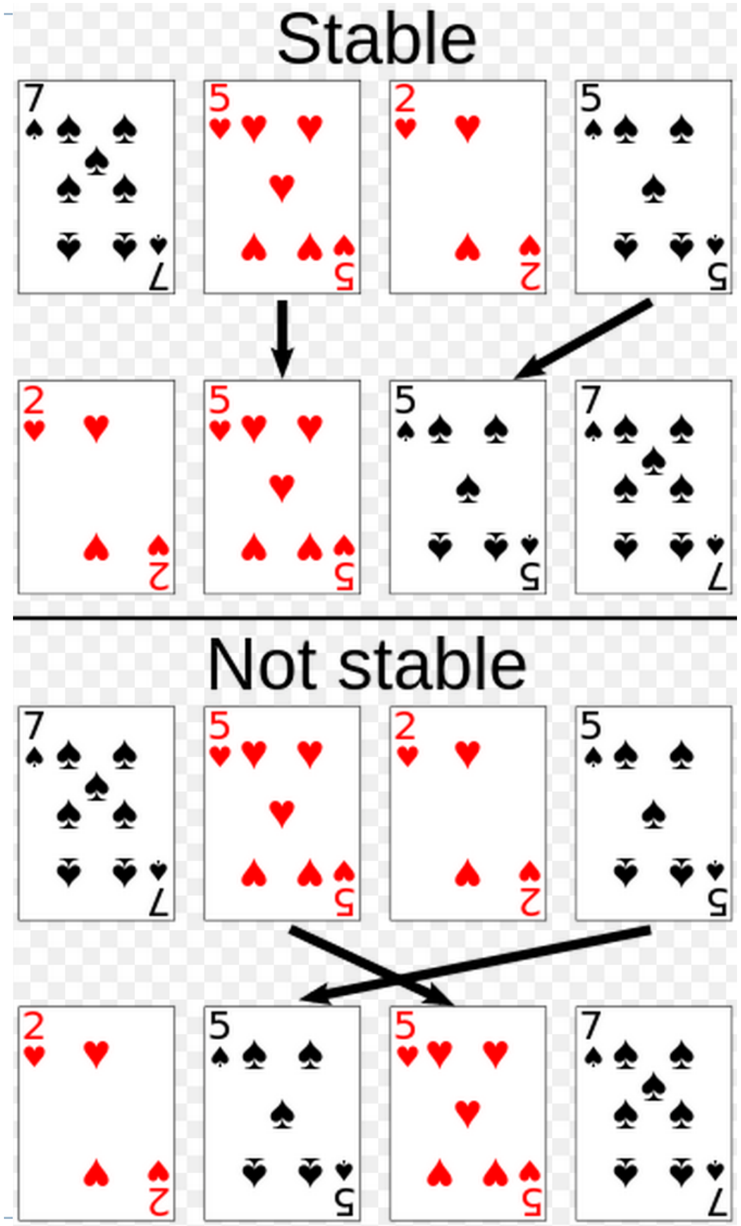
Some Definitions about Sorting

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Stability

- A STABLE sort preserves relative order of records with equal keys

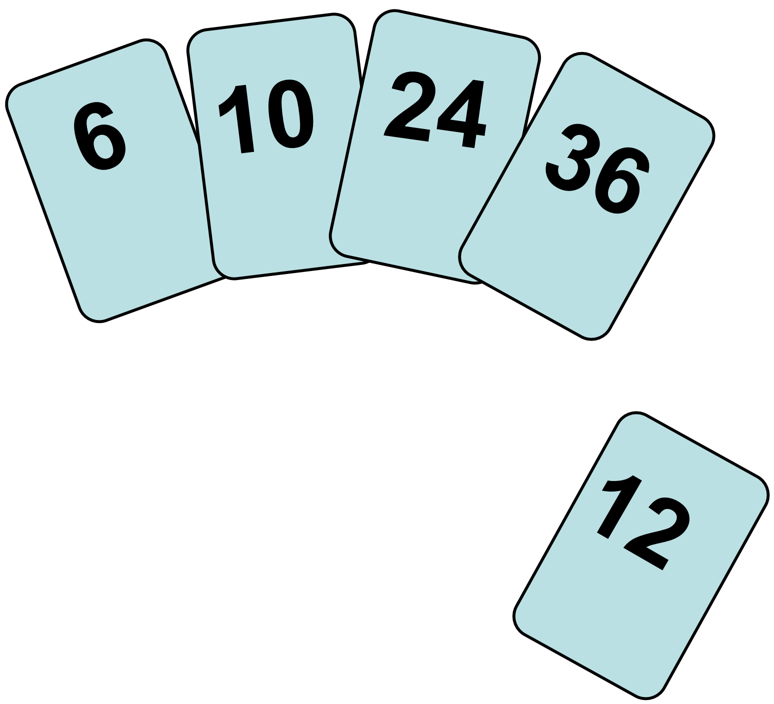
- A playing cards example
  - When the cards are sorted by rank with a stable sort, the two 5s must remain in the same order in the sorted output that they were originally in.
  - When they are sorted with a non-stable sort, the 5s may end up in the opposite order in the sorted output.



Insertion Sort

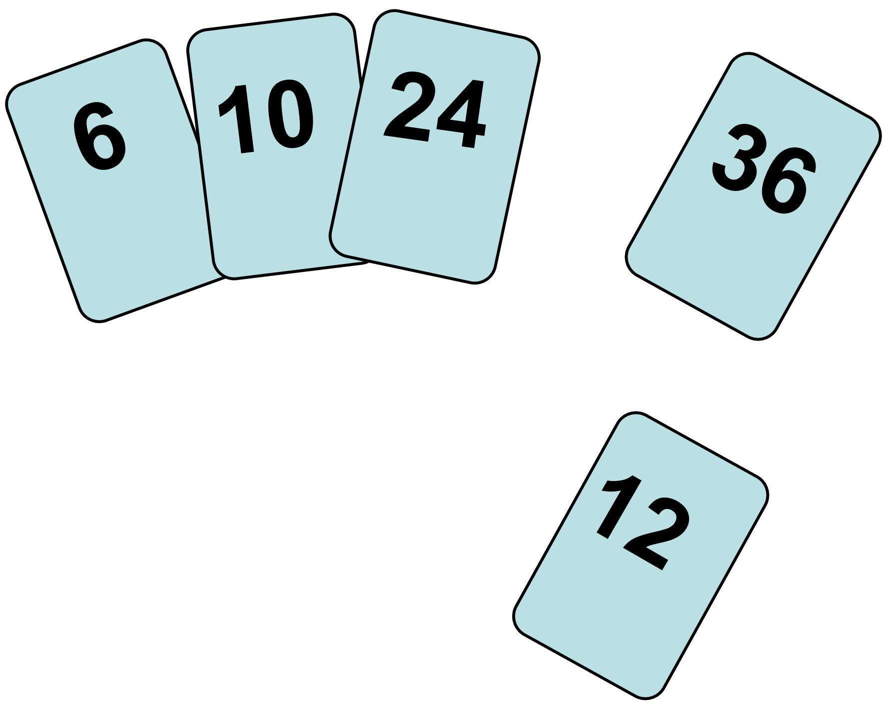
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - Compare it with each of the cards already in the left hand, from right to 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

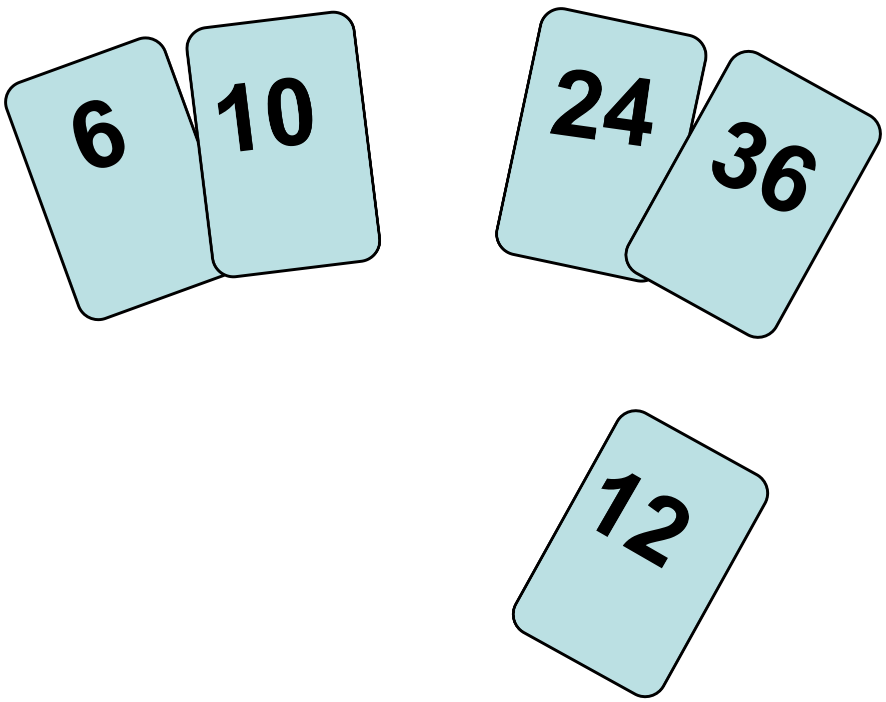


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

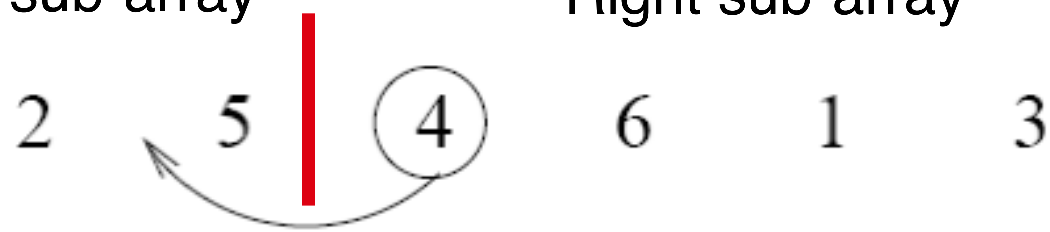
Input array

5 2 4 6 1 3

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

Left sub-array

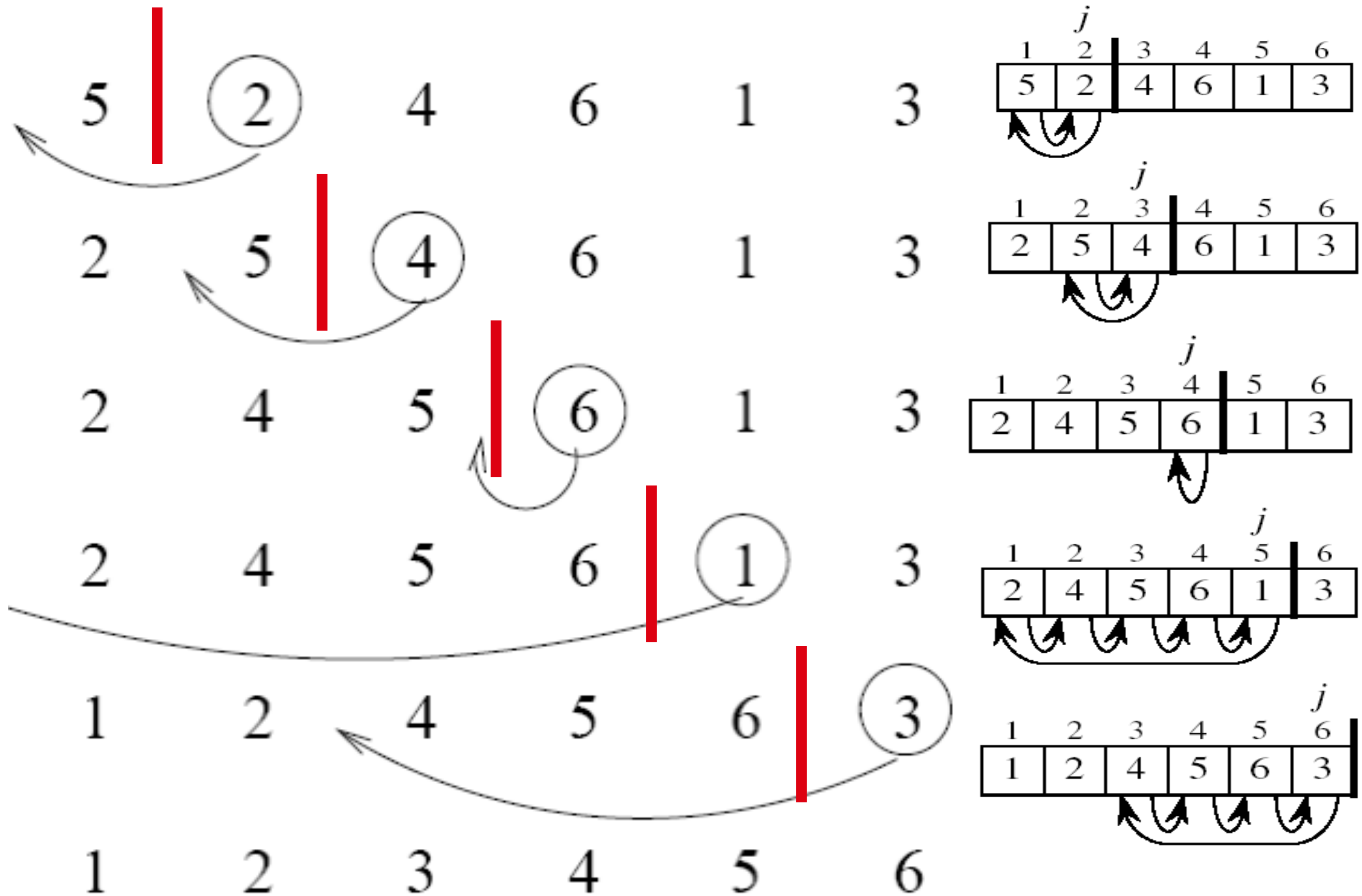
Right sub-array



Sorted

Unsorted

Insertion Sort



Pseudo-code: insertion sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

- In-place?
- Stable?

Proving Loop Invariants

Proving loop invariants works like induction

- 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

- **Loop Invariant:**

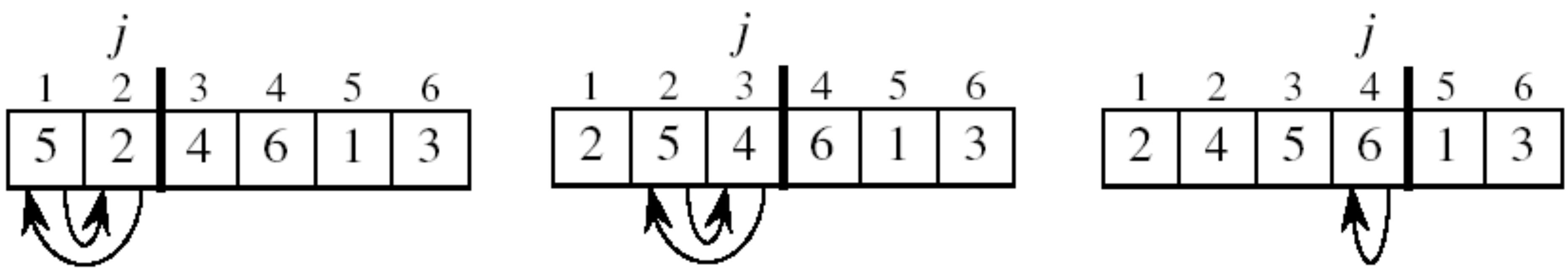
  at the start of each iteration of the for loop, the subarray A[1..j-1] consists of elements originally in A[1..j-1], but in sorted order

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

  the subarray A[1 . . j-1] = A[1], (the element originally in A[1]) – is sorted

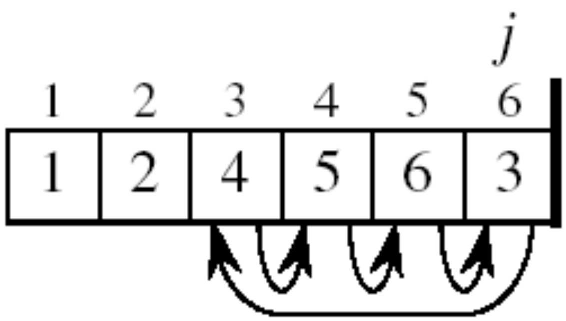
Loop Invariant for Insertion Sort

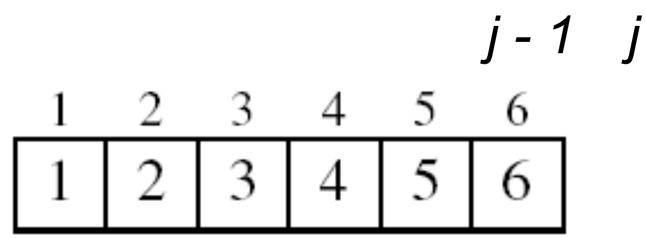
- **Maintenance:**
  - **while** inner loop moves A[j -1], A[j -2], A[j -3], and so on, by one position to the right until the proper position for key (which has the value that started out in A[j]) is found
  - At that point, the value of key is placed into this position.



Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when j = n + 1 ⇒ 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!

Running time analysis

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	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	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	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	$n-1$

Running time analysis

- running time

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2} t_j + c_6 \sum_{j=2} (t_j - 1) \\
&\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \, .
\end{aligned}
$$

- best case:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

- worst case:

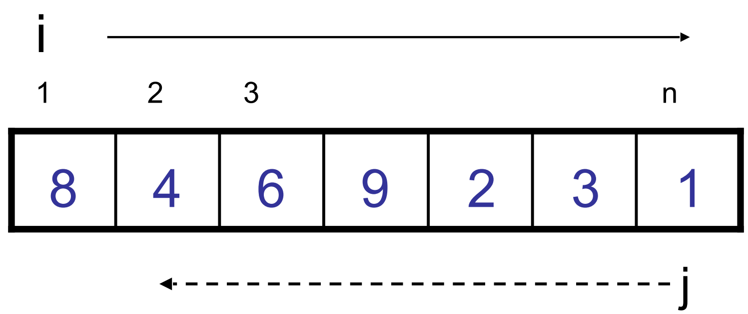
$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\
&\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

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

Bubble Sort

- Idea
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slower than Insertion sort

Bubble Sort: Example

8	4	6	9	2	3	1

i = 1 j

8	4	6	9	2	1	3

i = 1 j

8	4	6	9	1	2	3

i = 1 j

8	4	6	1	9	2	3

i = 1 j

8	4	1	6	9	2	3

i = 1 j

8	1	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 2 j

1	2	8	4	6	9	3

i = 3 j

1	2	3	8	4	6	9

i = 4 j

1	2	3	4	8	6	9

i = 5 j

1	2	3	4	6	8	9

i = 6 j

1	2	3	4	6	8	9

i = 7

j

Sorting

- Insertion sort
  - Design approach: Incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

- Bubble Sort
  - Design approach: Incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

Sorting

- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: Let's see!!

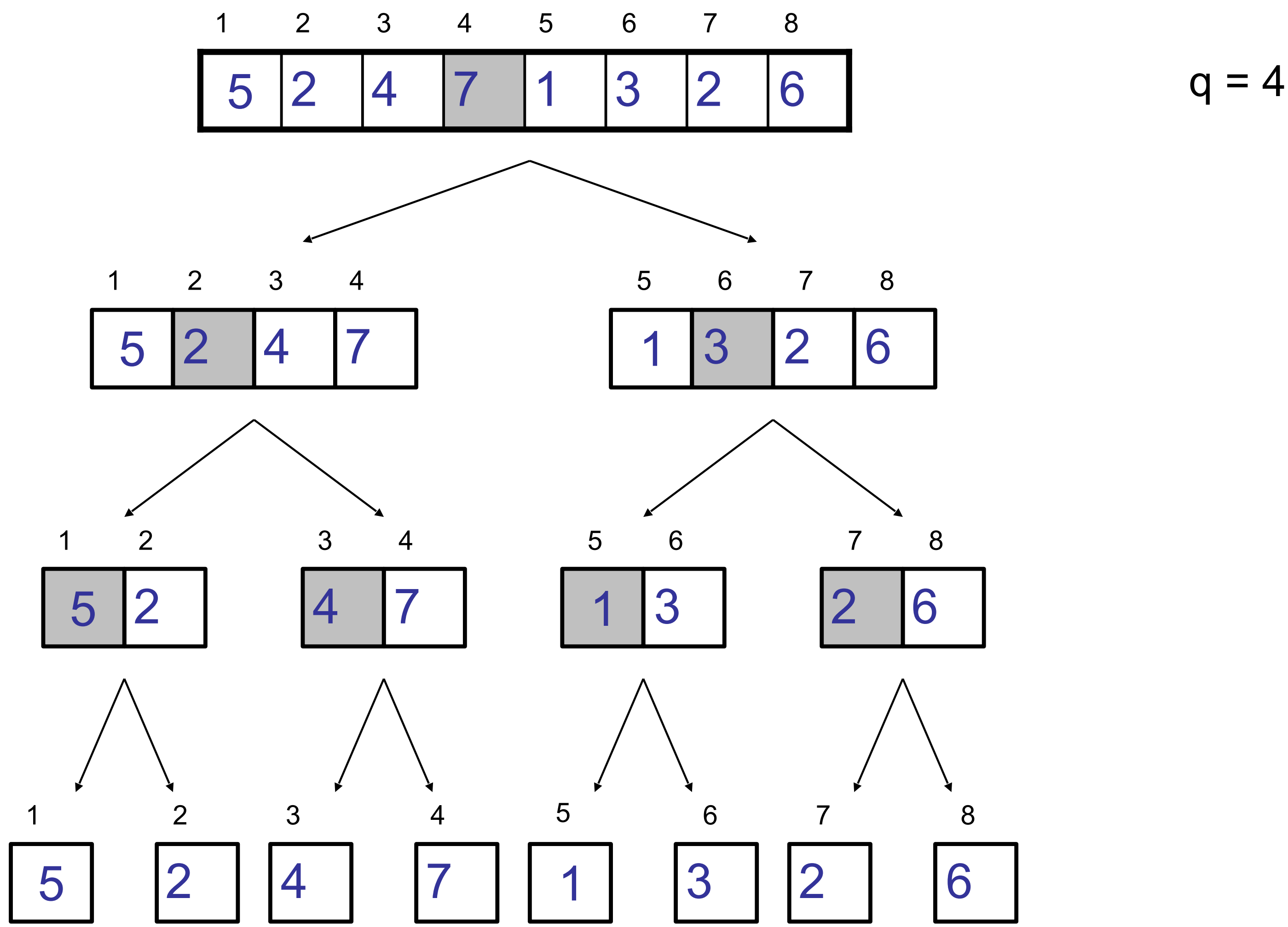
Merge Sort Approach

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

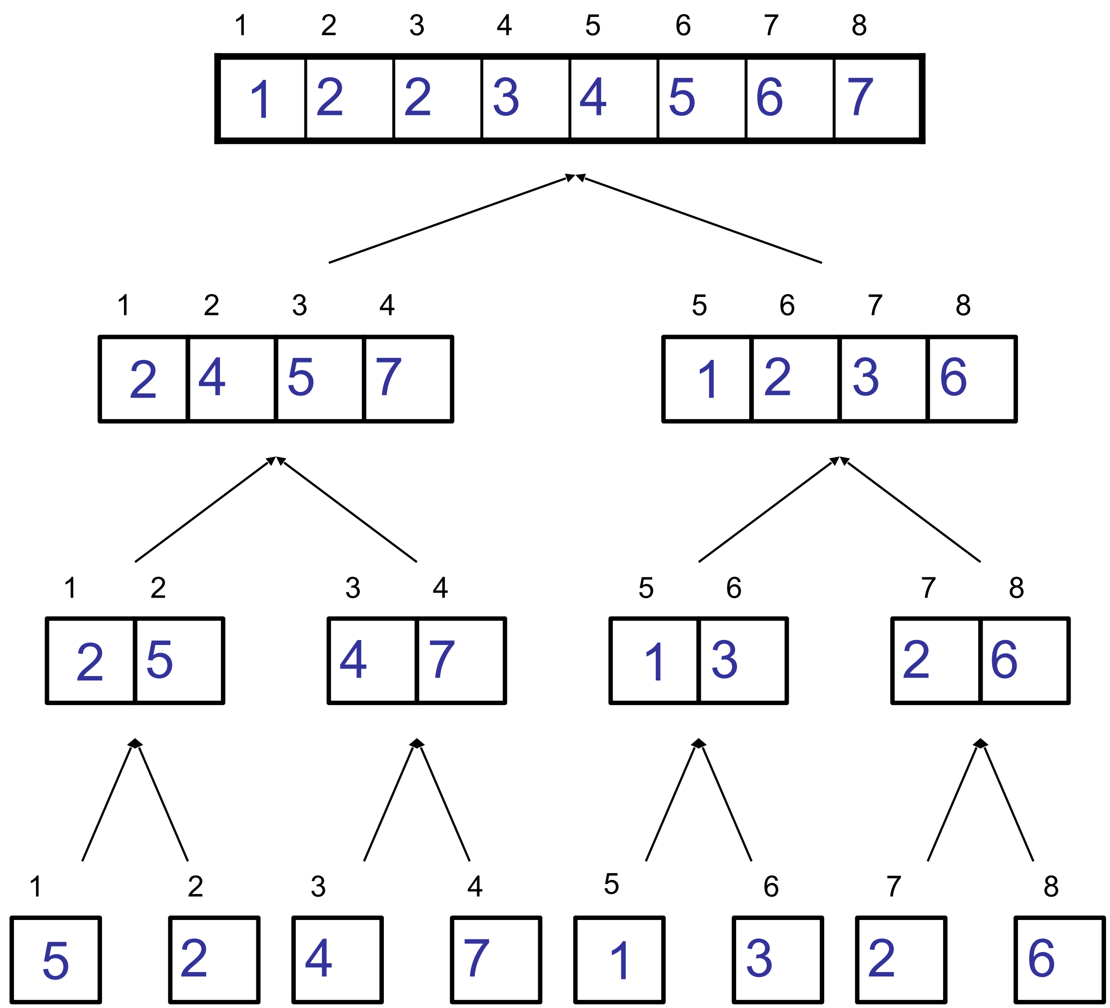
Merge Sort: Example 1

Divide

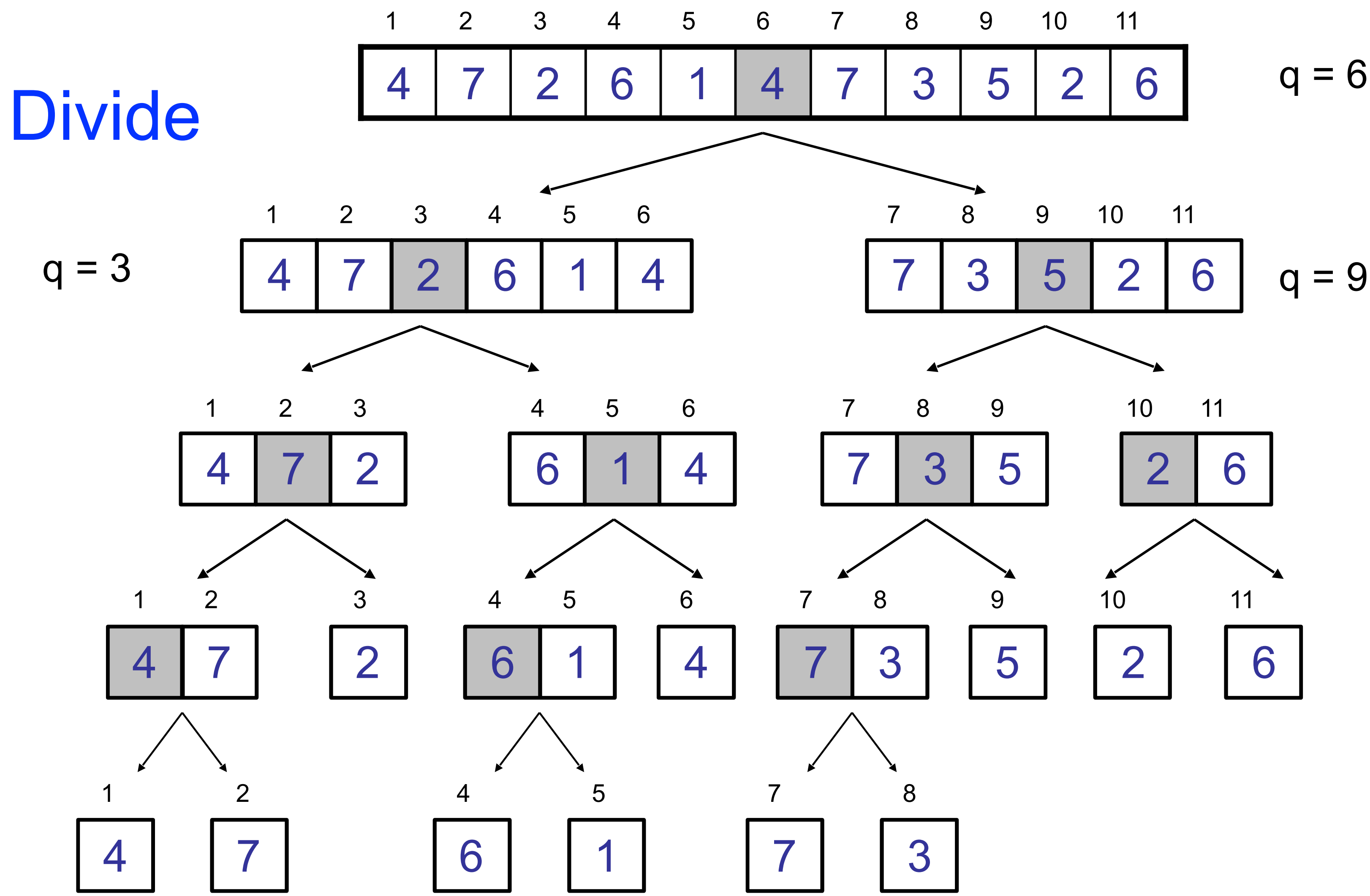


Merge Sort: Example 1

Conquer and Merge

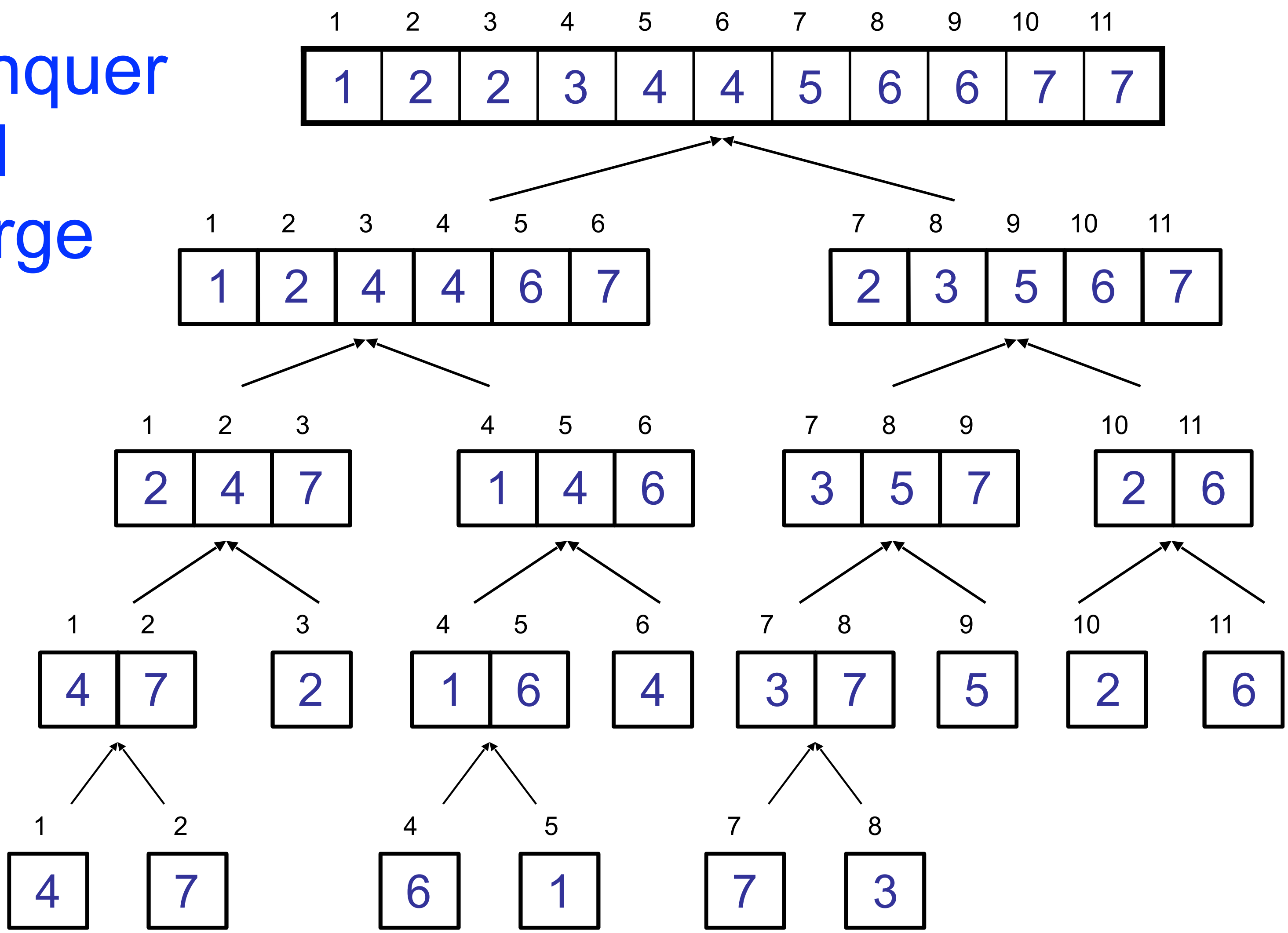


Merge Sort: Example 2

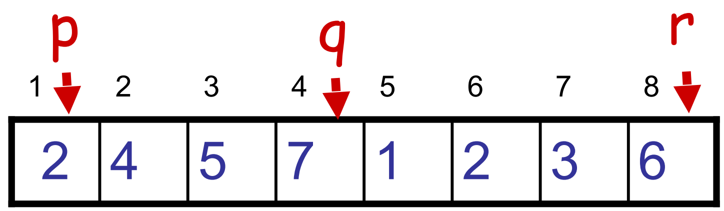


Merge Sort: Example 2

Conquer
and
Merge



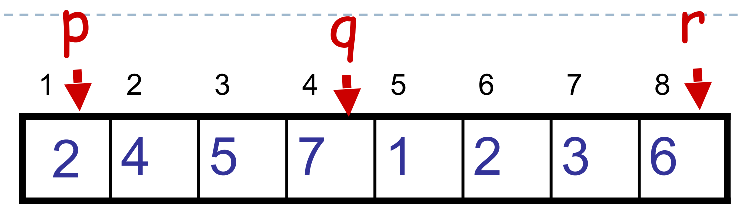
Merging



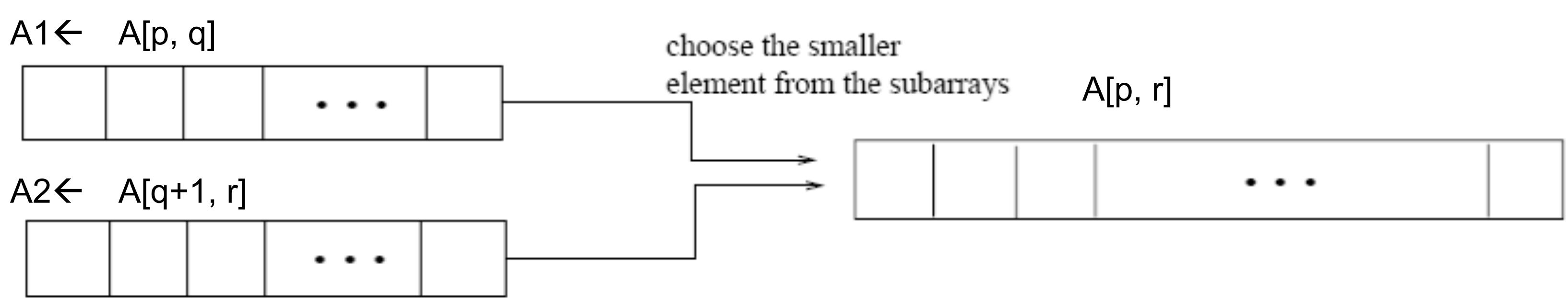
- **Input:** Array A and indices p, q, r such that p ≤ q < r
  - Subarrays A[p . . q] and A[q + 1 . . r] are sorted
- **Output:** One single sorted subarray A[p . . r]

Merging

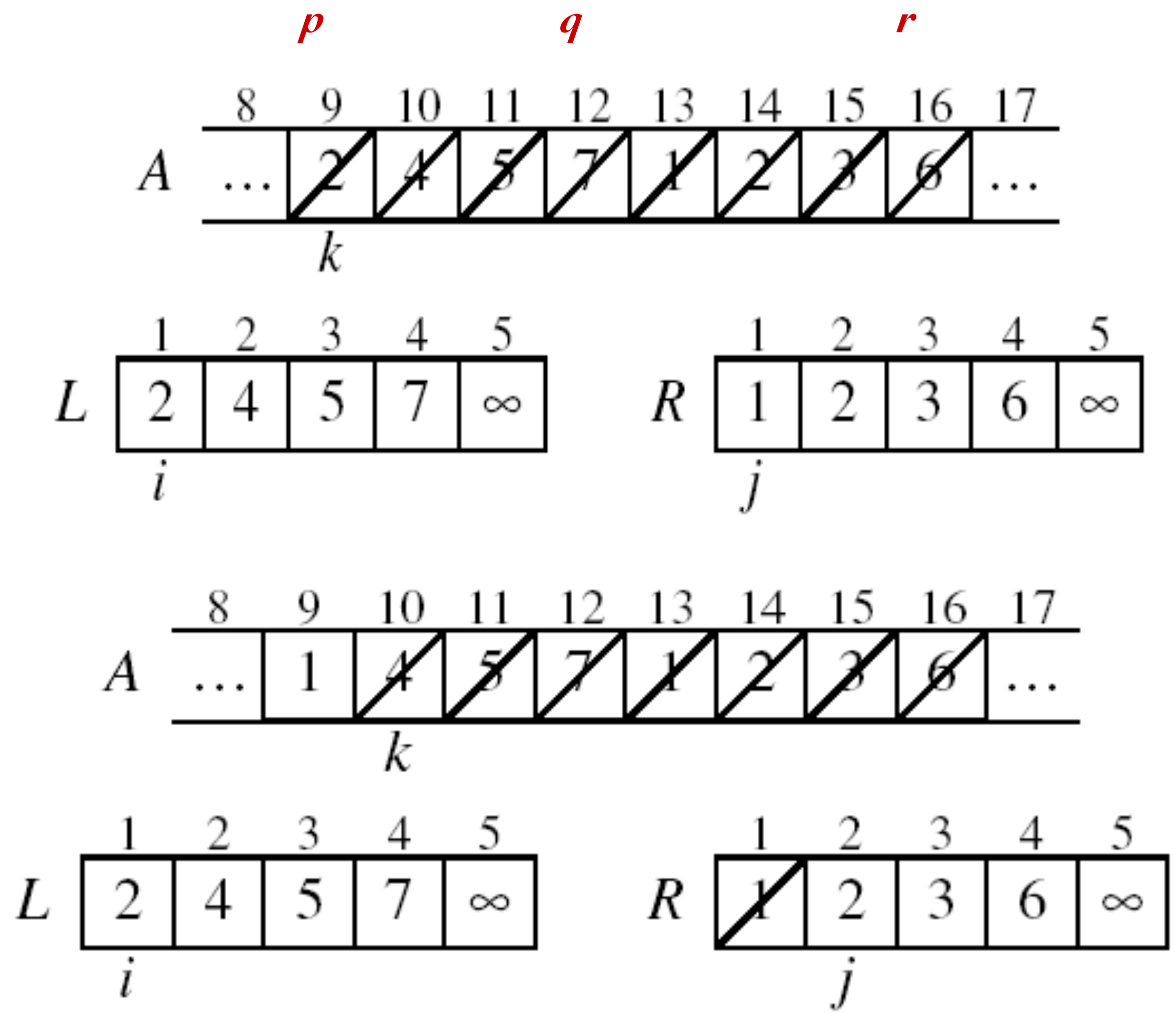
- Idea for merging



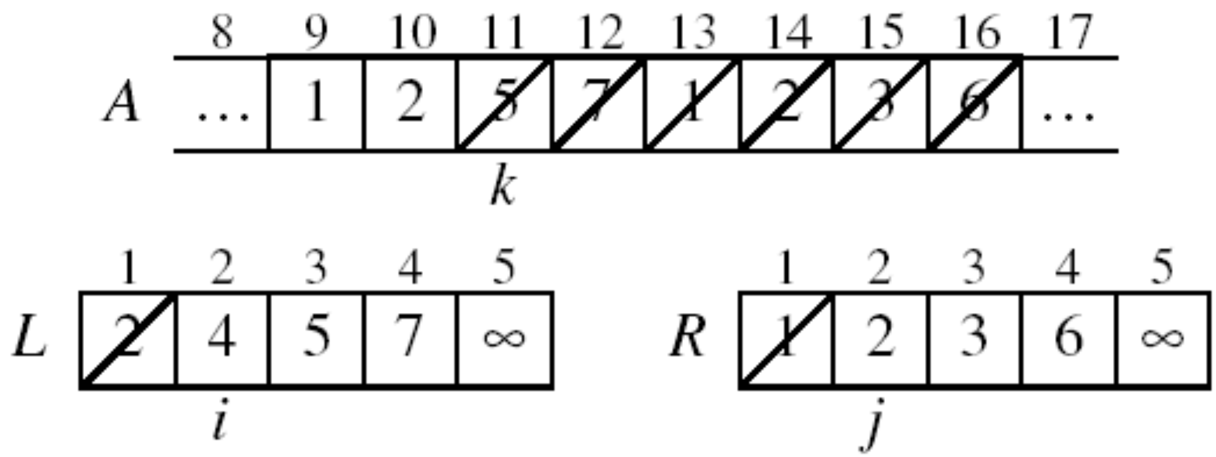
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

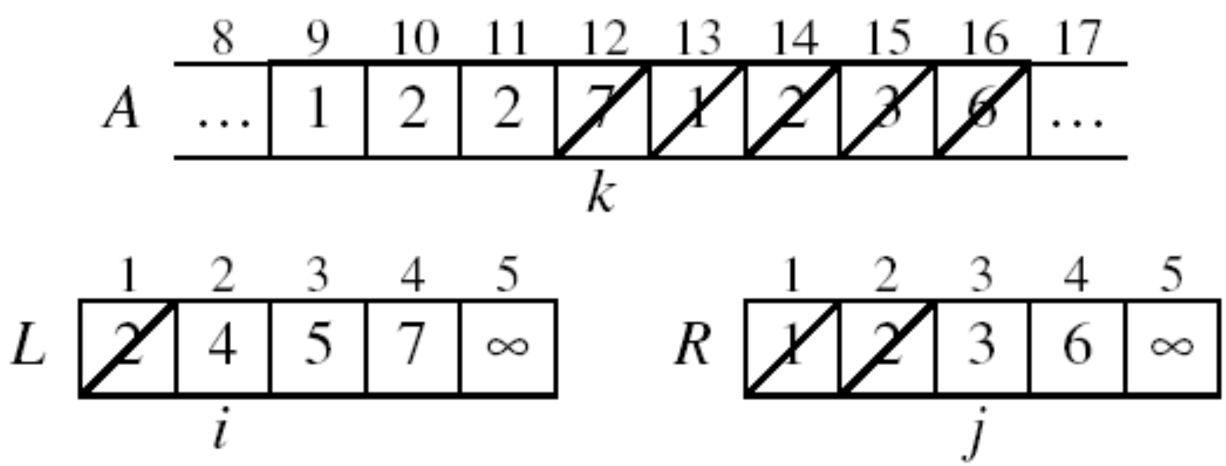


Example: MERGE(A, 9, 12, 16)



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





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

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	~~1~~	~~2~~	~~3~~	~~6~~	…
						k				

	1	2	3	4	5
L	~~2~~	4	5	7	∞
		i			

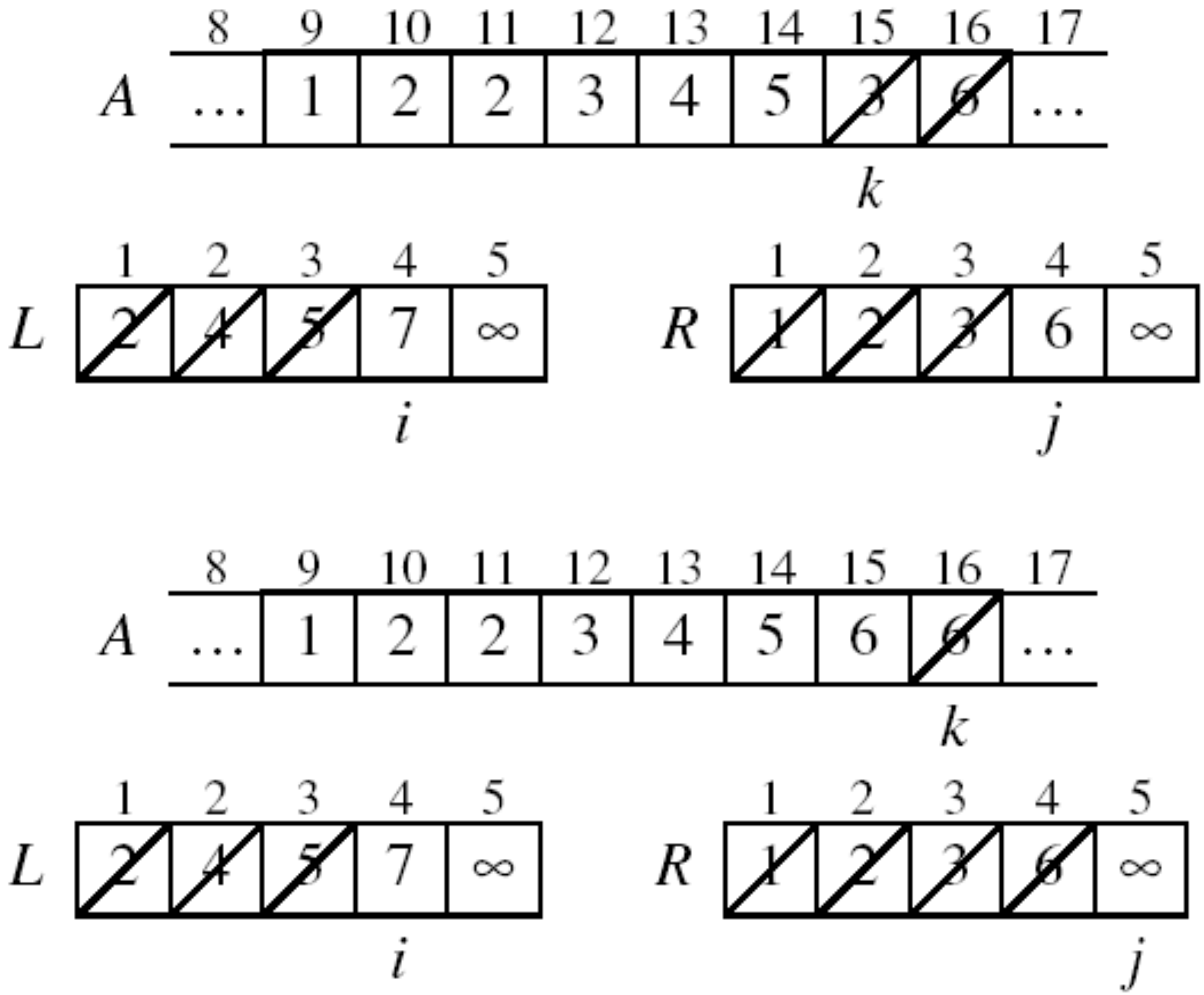
	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	~~2~~	~~3~~	~~6~~	…
							k			

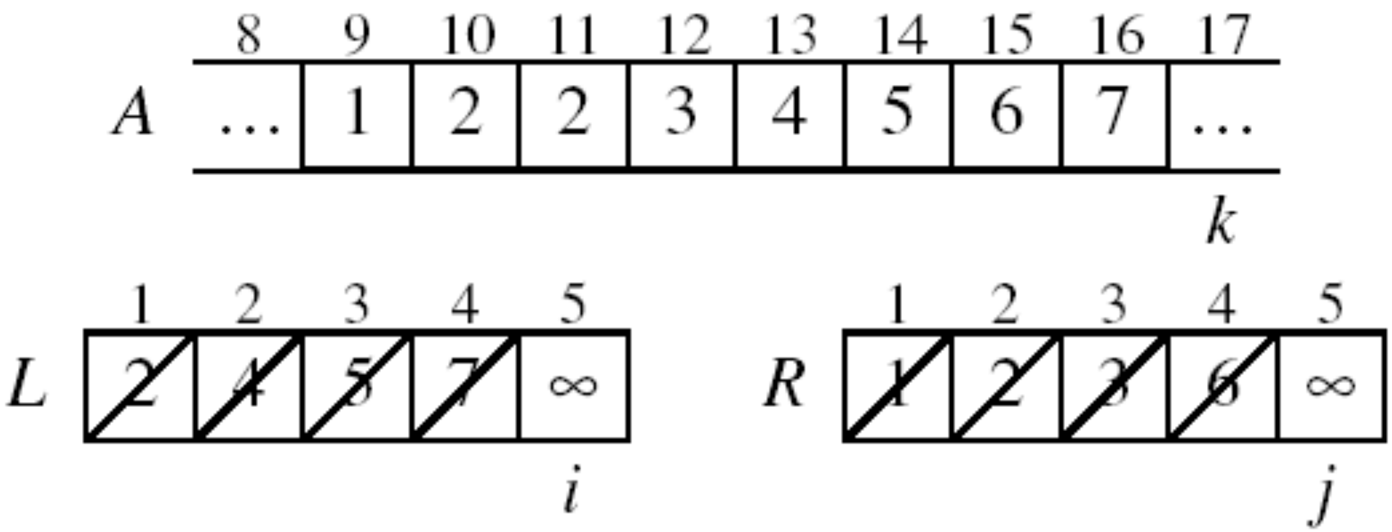
	1	2	3	4	5
L	~~2~~	~~4~~	5	7	∞
			i		

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

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



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



Done!

- **In Place?**

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

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

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

Merge Sort - Discussion

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

Sorting Challenge 1

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. Bubble sort
B. Mergesort guaranteed to run in time ~nlogn
C. Insertion sort

Sorting Huge, Randomly-Ordered Files

- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Mergesort?
  - YES, it is designed for this problem

Sorting Challenge 2

- Problem: sort a file that is already almost in order
- Applications:
  - Re-sort a huge database after a few changes
  - Double check that someone else sorted a file
- Which sorting method to use?
  - Mergesort, guaranteed to run in time ~nlgn
  - Bubble sort
  - Insertion sort

Sorting Files That are Almost in Order

- Bubble sort?
  - NO, bad for some definitions of “almost in order”
  - Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Insertion sort?
  - YES, takes linear time for most definitions of “almost in order”
- Mergesort or custom method?
  - Probably not: insertion sort simpler and faster

What's next...

- More sorting algorithms
  - Heapsort
  - Quicksort
  - ...