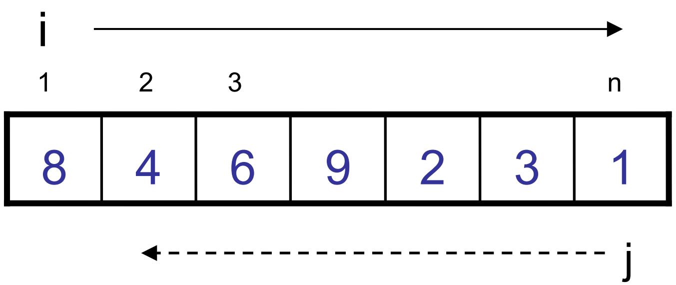
Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 7

Bubble Sort

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

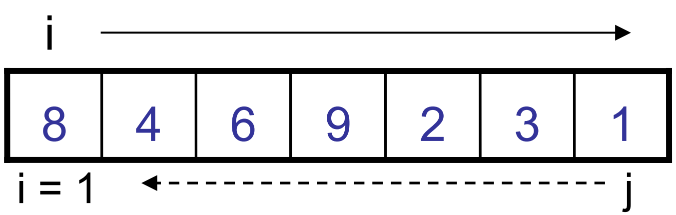


- Easier to implement, but slower than Insertion sort

Bubble Sort

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[*A*]
    do for j ← length[*A*] **downto** i + 1
        do if *A*[j] < *A*[j -1]
            then exchange *A*[j] ⟺ *A*[j-1]



Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

do for j ← length[A] **downto** i + 1

do if A[j] < A[j -1]

then exchange A[j] ⟺ A[j-1]

Comparisons: $\approx n^2/2$

Exchanges: $\approx n^2/2$

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n} (n-i+1) + ¿¿ \; c_3 \sum_{i=1}^{n} (n-i) + ¿¿ \; c_4 \sum_{i=1}^{n} (n-i)$$

$$= \Theta(n) + (c_2 + c_3 + c_4) \sum_{i=1}^{n} (n-i)$$

$$\approx \sum_{i=1}^{n} (n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

$$T(n) = \Theta(n^2)$$

Selection Sort

8	4	6	9	2	3	1

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Invariant:
  - All elements to the left of the current index are in sorted order and never changed again
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Example

8	4	6	9	2	3	1

1	4	6	9	2	3	8

1	2	6	9	4	3	8

1	2	3	9	4	6	8

1	2	3	4	9	6	8

1	2	3	4	6	9	8

1	2	3	4	6	8	9

1	2	3	4	6	8	9

Selection Sort

Alg.: SELECTION-SORT(*A*)

8	4	6	9	2	3	1

```
n ← length[A]
for j ← 1 to n - 1
        do smallest ← j
            for i ← j + 1 to n
                do if A[i] < A[smallest]
                    then smallest ← i
            exchange A[j] ⟺ A[smallest]
```

Analysis of Selection Sort

Alg.: SELECTION-SORT(*A*) cost times

n ← length[*A*]

for j ← 1 **to** n - 1 c_1 1

do smallest ← j c_2 n

≈$n^2/2$ comparisons

for i ← j + 1 **to** n c_3 $\sum_{j=1}^{n-1}(n-j+1)$ n

do if *A*[i] < *A*[smallest] 1 $\sum_{j=1}^{n-1}(n-j)$

then smallest ← i c_4 $\sum_{j=1}^{n-1}(n-j)$

≈n exchanges

exchange *A*[j] ⟺ *A*[smallest] c_5

$T(n) = \Theta(n^2)$

c

Divide-and-Conquer

- **Divide** the problem into a number of subproblems
  - 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 to the sub-problems
  - Obtain the solution for the original problem

Analyzing Divide and Conquer Algorithms

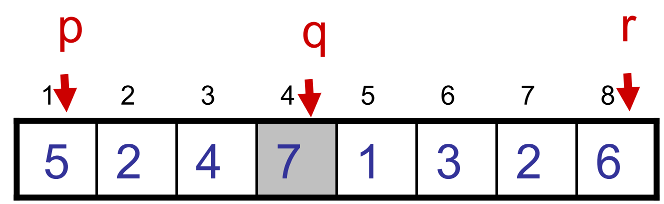
- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size n
  - **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
  - **Conquer** (solve) the subproblems: takes $aT(n/b)$
  - **Combine** the solutions: takes $C(n)$

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

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

    MERGE-SORT(A, p, q) ▷ Conquer

    MERGE-SORT(A, q + 1, r) ▷ Conquer

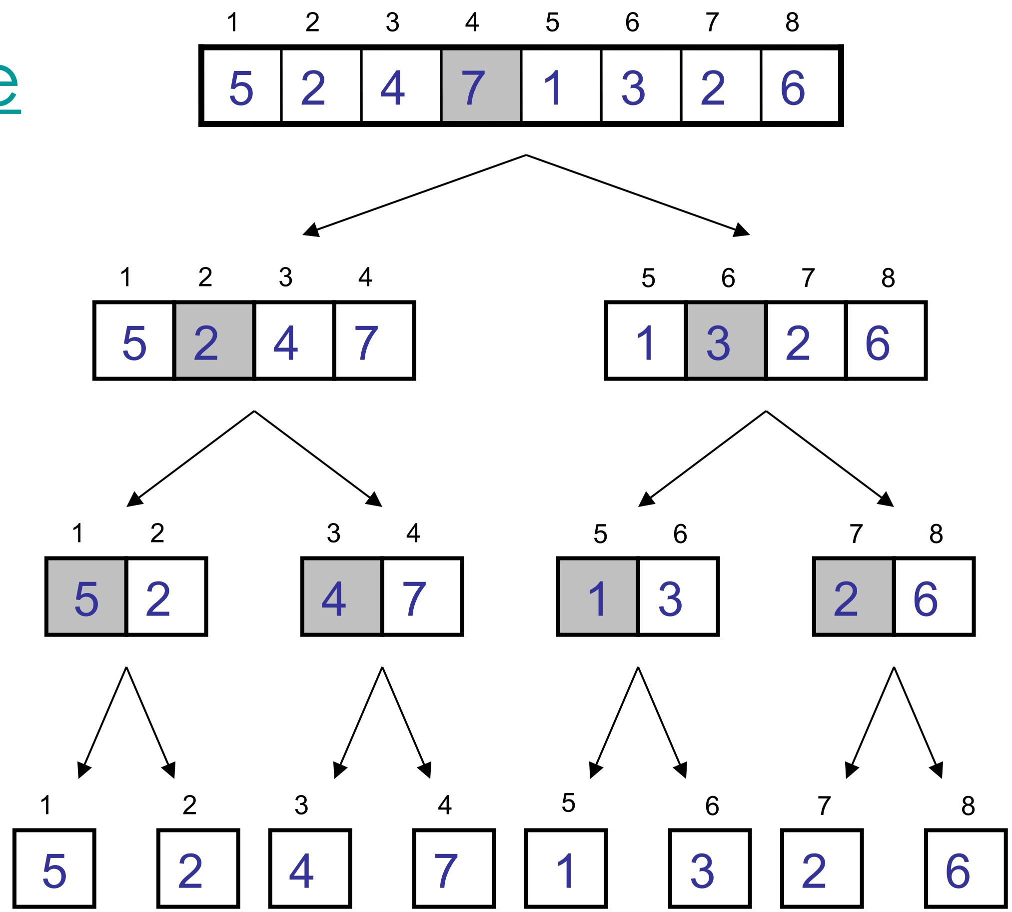
    MERGE(A, p, q, r) ▷ Combine

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

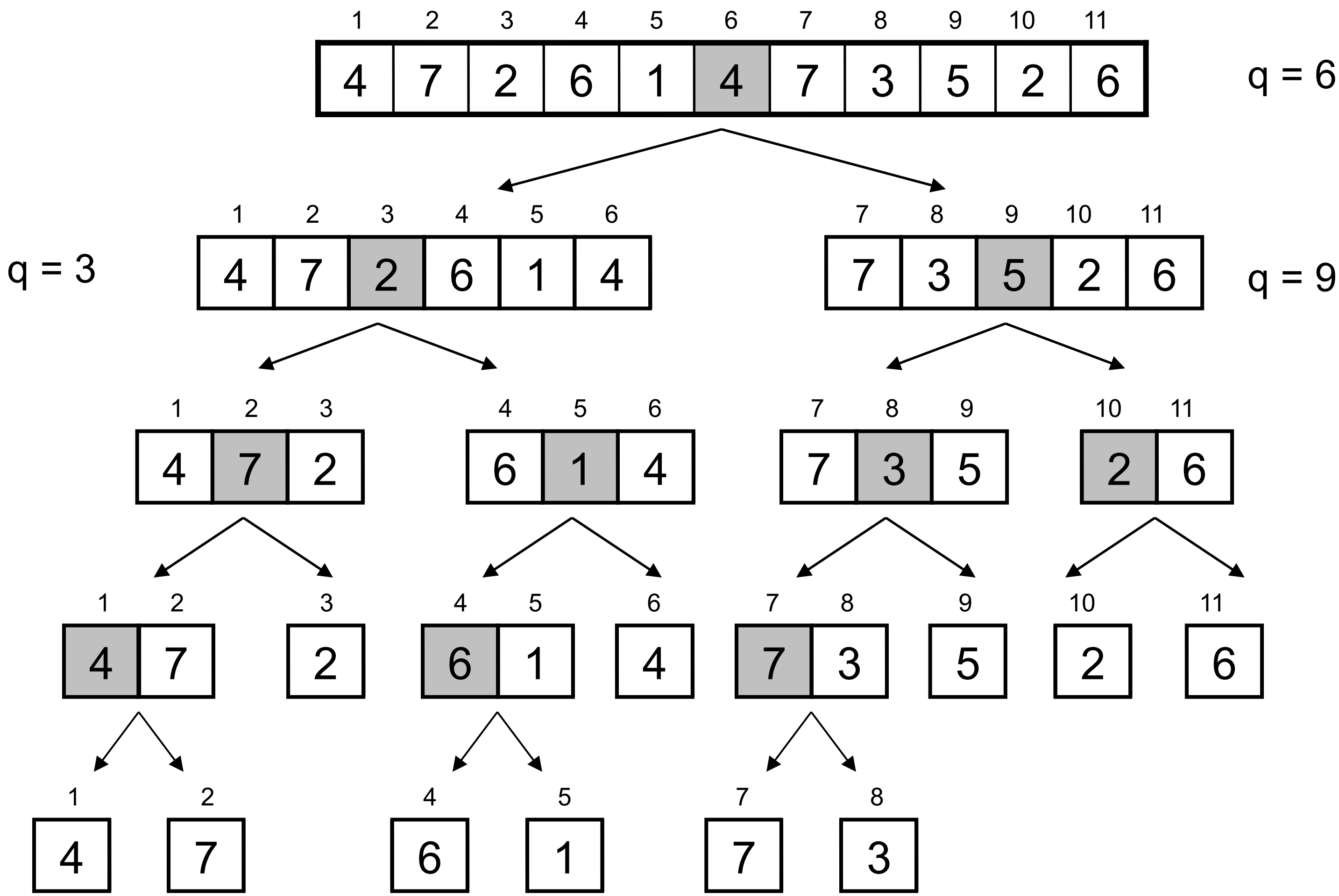
Example – *n* Power of 2

Example

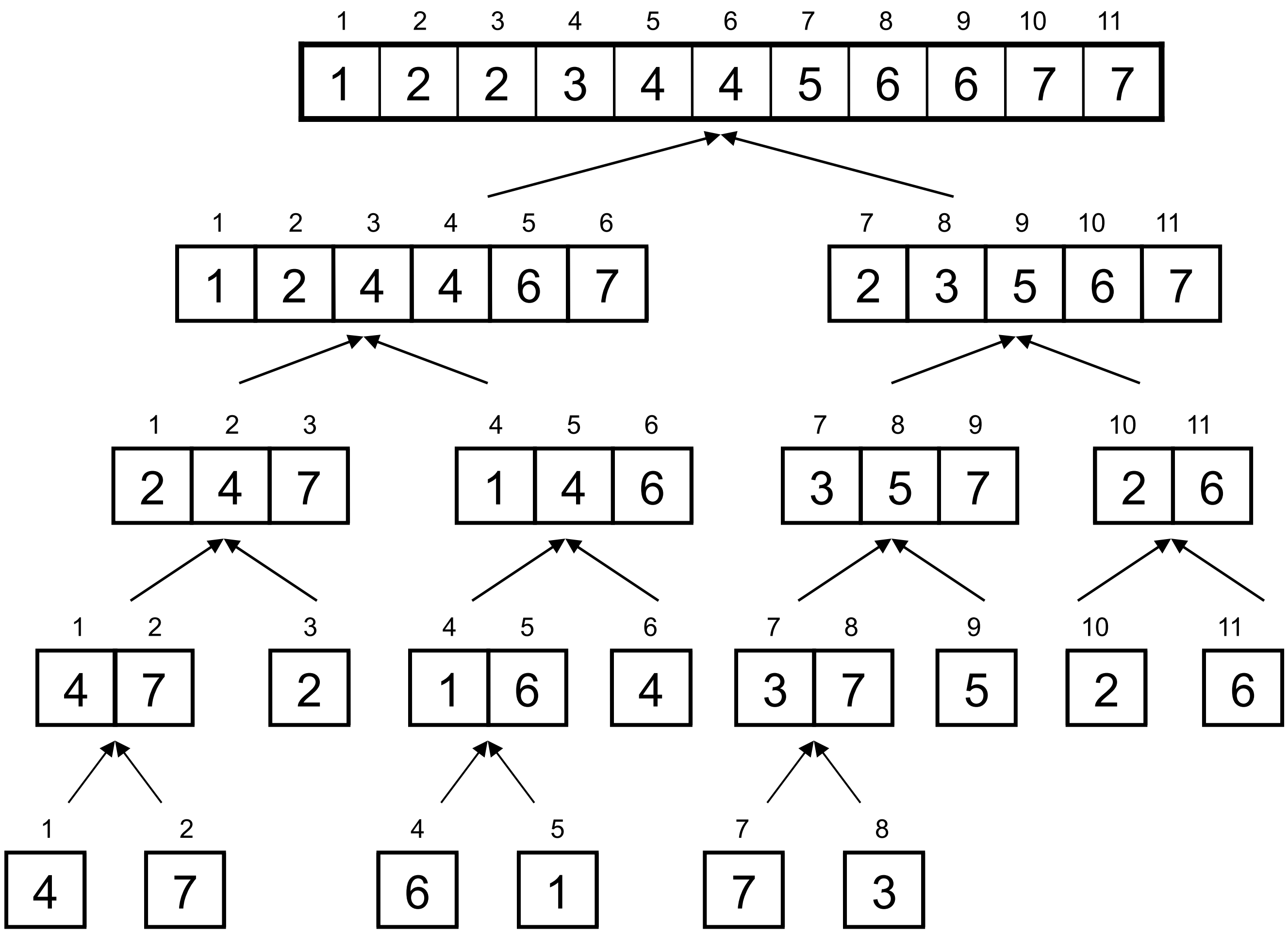
q = 4



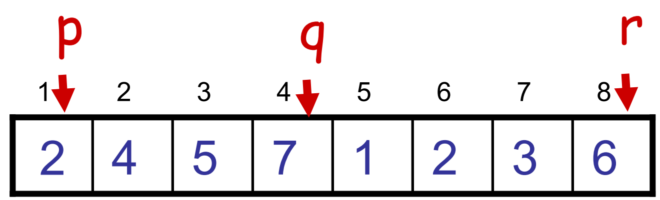
Example – *n* Not a Power of 2



Example – *n* Not a Power of 2



Merging



- **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 $A[p \,..\, r]$

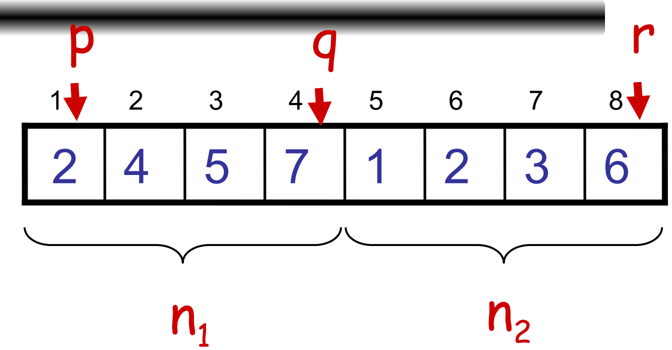
Merging

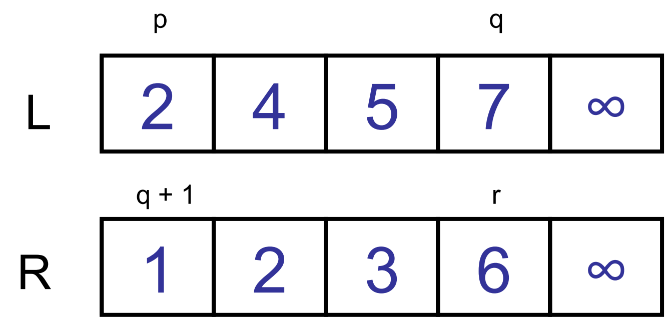
- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

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. $j \leftarrow j + 1$





Running Time of Merge

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

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: takes $aT(n/b)$
  - **Combine** the solutions: takes $C(n)$

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

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

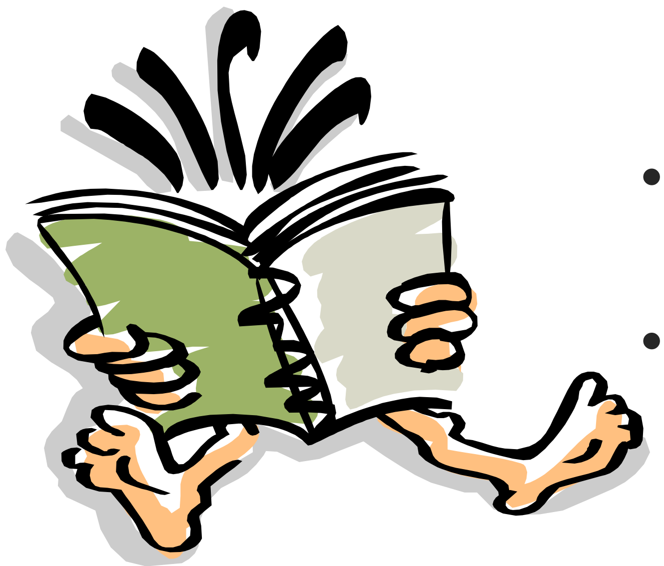
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

Compare n with $f(n) = cn$

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

Readings



- For this lecture
  - Section 2.2, 2.3, 7.1
- Coming next
  - Section 7.2-7.4