CSCI 3100

DIVIDE AND CONQUER: 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 $A[p \ldots r]$:

Divide

- Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each

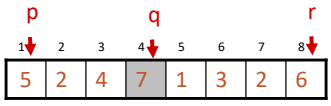
Conquer

- 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



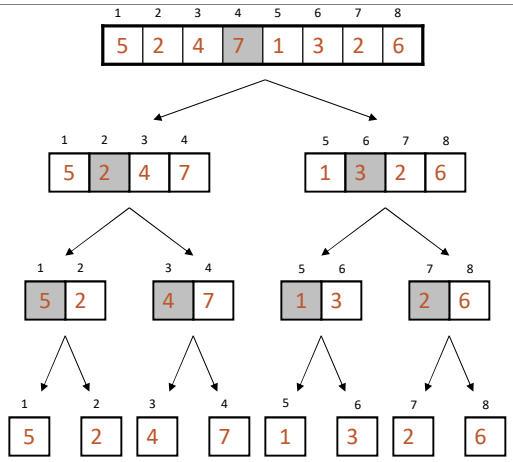
Alg.: **MERGE-SORT(A, p, r)**

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

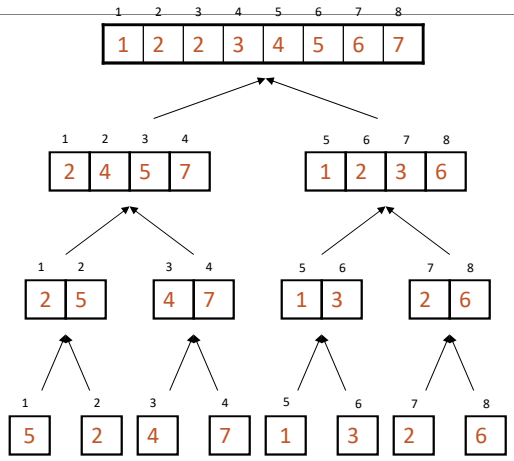
Divide



5

Example – **n** Power of 2

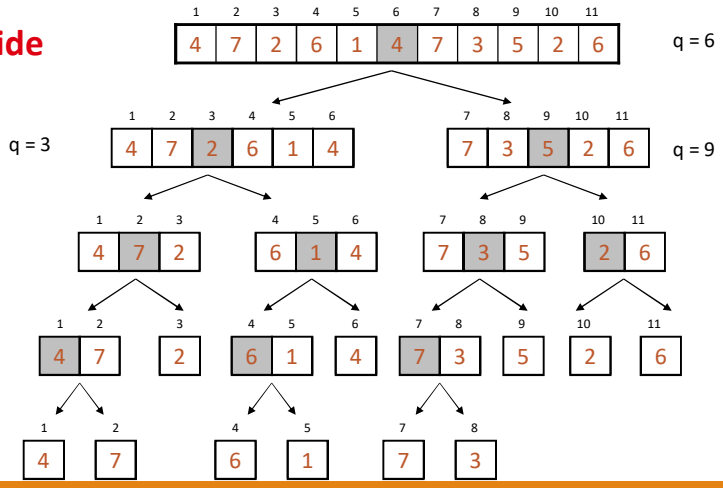
Conquer and Merge



6

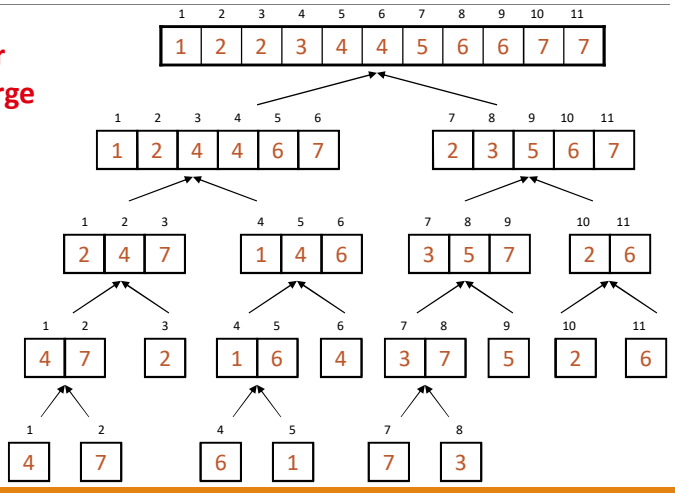
Example – **n** not a Power of 2

Divide

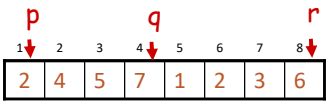


Example – **n** Not a Power of 2

Conquer and Merge



Merging



Input:

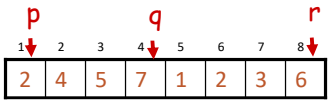
Array A and indices p, q, r such that $p \le q < r$

- Subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ are sorted

Output: One single sorted subarray $A[p \,..\, r]$

9

Merging



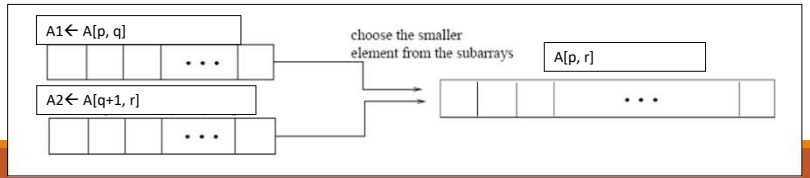
Strategy:

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

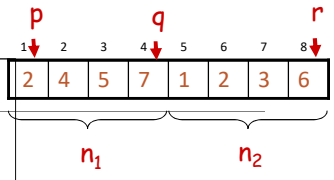


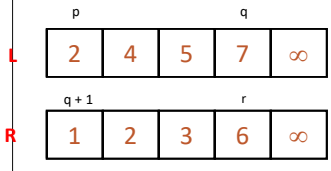
Try it yourself

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** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$





Running Time of Merge

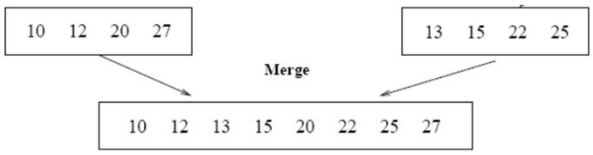
Initialization (copying into temporary arrays):

$\Theta(n_1 + n_2) = \Theta(n)$

Adding the elements to the final array:

- n iterations, each taking constant time $\Rightarrow \Theta(n)$

Total time for Merge: $\Theta(n)$



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) = Θ(1)

Conquer:
- recursively solve 2 subproblems, each of size n/2 ⇒ 2T (n/2)

Combine:
- MERGE on an n-element subarray takes Θ(n) time ➔ C(n) = Θ(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 $\boldsymbol{T(n)} = \begin{cases} \boldsymbol{c \; if \; n = 1} \\ \boldsymbol{2T\left(\frac{n}{2}\right) + cn \; if \; n > 1} \end{cases}$

Use Master Theorem: with a = 2, b = 2, f(n)=cn

$a = 2 \quad b = 2 \quad f(n) = cn$

$n^{\log_b a} = n$

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

If f(n) is	Then T(n) is
$O(n^{\log_b a - \varepsilon})$	$\Theta(n^{\log_b a})$
$\Theta(n^{\log_b a})$	$\Theta(n^{\log_b a} * \lg(n))$
$\Omega(n^{\log_b a + \varepsilon})$	$\Theta(f(n))$

Notes on Merge Sort

Running time insensitive of the input

Advantage:

- Guaranteed to run in $\Theta(n \lg n)$

Disadvantage:

- Requires extra space ≈N

17

Sorting Algorithms' Complexities

Insertion sort	
Solution type	Incremental
Sorts in place	Yes
Best case	$\Theta(n)$
Worst case	$\Theta(n^2)$

Bubble sort	
Solution type	Incremental
Sorts in place	Yes
Best case	$\Theta(n)$
Worst case	$\Theta(n^2)$

Selection sort	
Solution type	Incremental
Sorts in place	Yes
Best case	$\Theta(n)$
Worst case	$\Theta(n^2)$

Merge sort	
Solution type	Divide and Conquer
Sorts in place	No
Best case	$\Theta(n \lg n)$
Worst case	$\Theta(n \lg n)$

Sorting Challenge 1

Problem:

Sort a **huge** randomly-ordered file of **small** records

Example: transaction record for a phone company

Which method to use?

A. bubble sort
B. selection sort
C. merge sort, guaranteed to run in time ~nlg(n)
D. insertion sort

Sorting **Huge**, Randomly - Ordered Files

Selection sort?

- NO, always takes quadratic time

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 II

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?

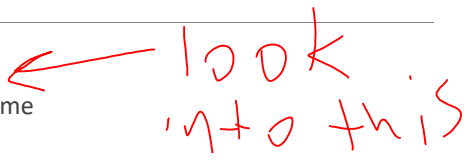
A. Mergesort, guaranteed to run in time ~nlg(n)
B. Selection sort
C. Bubble sort
D. A custom algorithm for almost in-order files
E. Insertion sort

21

Sorting files that are almost in order

Selection sort?

- NO, always takes quadratic time



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