Lecture 3:
Sorting Algorithms

Bo Tang @ SUSTech, Fall 2022

Sorting Problem

- Sorting problem
  - Input: an array $A[1..n]$ with n integers
  - Output: a sorted array A (in ascending order)

- Problem is: sort $A[1..n]$

- Input: | 8 | 6 | 1 | 3 | 7 | 2 | 5 | 4 |

- Output: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Our Roadmap

- Comparison-based Sorting

  - Quadratic Cost

    - Selection Sort, Insertion Sort, Bubble Sort

  - O(n log n) Cost

    - Merge Sort, Heap Sort (we will skip here)

  - Quick Sort

- Other sorting algorithms

  - Counting sort, radix sort, bucket sort

Selection Sort

Selection Sort

◈ Idea of a selection sort method

  ◈ Start with empty hand, all cards on table

  ◈ Pick the smallest card from table

  ◈ Insert the card into the hand

8
5
2
6
9
3
1
4
0
7

Selection Sort Algorithm

◈ SelectionSort

$$8 \mid 6 \mid 1 \mid 3 \mid 7 \mid 2 \mid 5 \mid 4$$

◈ Input: an **array** A of n numbers

◈ Output : an **array** A of n numbers in the <u>ascending order</u>

◈ Selection-Sort ($A[1..n]$)

1. for integer $i \leftarrow$ 1 to n–1

2. $k \leftarrow i$

3.   for integer $j \leftarrow i$+1 to n

4.   if $A[k] > A[j]$ then

5. $k \leftarrow j$

6.   swap $A[i]$ and $A[k]$

$$1 \mid 6 \mid 8 \mid 3 \mid 7 \mid 2 \mid 5 \mid 4$$

sorted   *unsorted*

$$1 \mid 2 \mid 8 \mid 3 \mid 7 \mid 6 \mid 5 \mid 4$$

sorted   *unsorted*

Selection Sort Time Complexity

◈ Selection Sort

  ◈ Input: an **array** A of n numbers

  ◈ Output : an **array** A of n numbers in the <u>ascending order</u>

  ◈ Selection-Sort ($A[1..n]$)

    1. for integer $i \leftarrow 1$ to $n-1$ Cost: n-1=O(n)

    2. $k \leftarrow i$ Cost: n-1=O(n)

    3.   for integer $j \leftarrow i+1$ to n Cost: n-1+n-2+...+1=O(n²)

    4.   if $A[k] > A[j]$ then   Cost: O(n²)

    5. $k \leftarrow j$ Cost: O(n²)

    6.   swap $A[i]$ and $A[k]$ Cost: O(n)
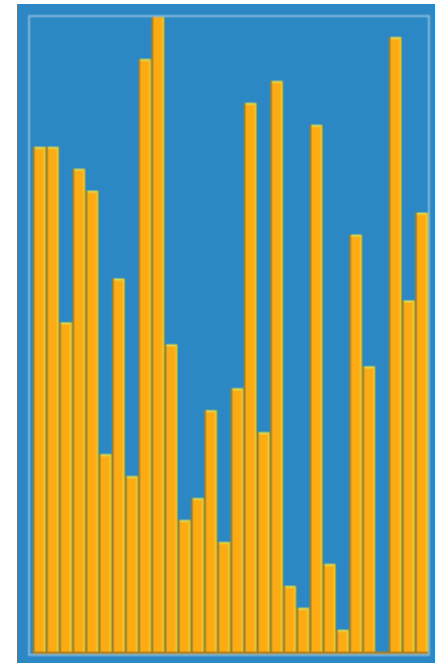
◈ Selection sort total cost:

◈ O(n)+O(n)+O(n²) +O(n²) +O(n²) +O(n)=O(n²)

Insertion Sort

Insertion Sort

◈ Idea of a insertion sort method

  ◈ One input each iteration, growing a sorted output list

  ◈ Remove one element from input data

  ◈ Find the location it belongs within the sorted list
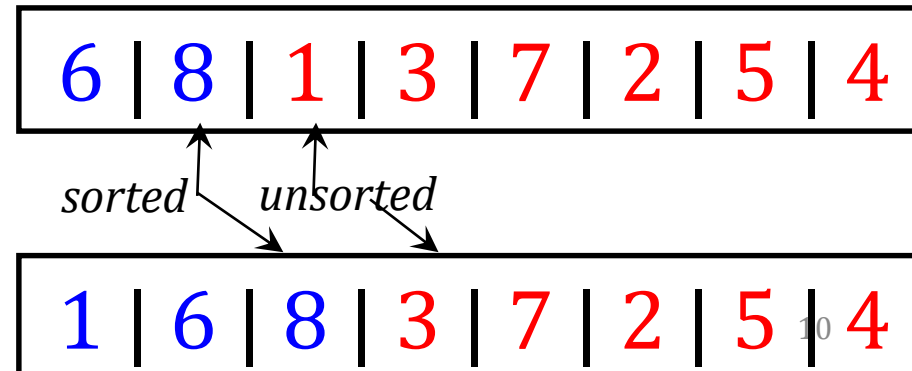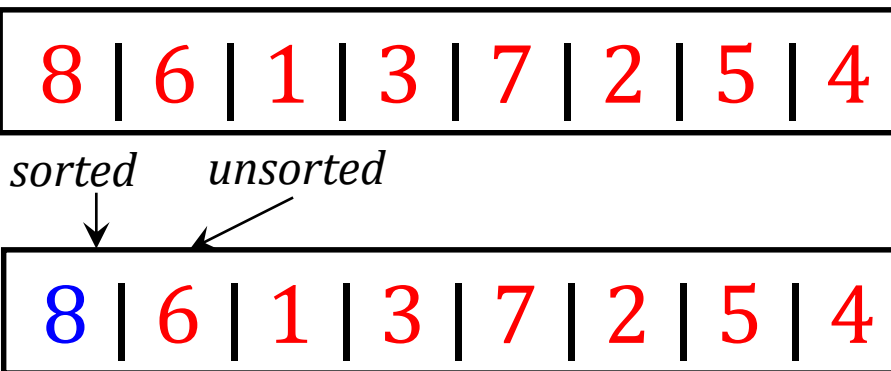
  ◈ Repeat until no input elements remain

6 5 3 1 8 7 2 4

Insertion Sort Algorithm

◈ InsertionSort

  ◈ Input: an **array** *A* of *n* numbers

  ◈ Output : an **array** *A* of *n* numbers in the <u>ascending order</u>

  ◈ Insertion-Sort (*A*[1..*n*])

      1. for integer *i* ← 1 to *n*

      2.   for integer *j* ← *i to 1 with j > 1*

      3.   if   *A*[*j-1*] > *A*[*j*] then

      4.   swap *A*[*j-1*] and *A*[*j*]

      5.   else   break

| 8 | 6 | 1 | 3 | 7 | 2 | 5 | 4 |

sorted   *unsorted*

| 8 | 6 | 1 | 3 | 7 | 2 | 5 | 4 |

| 6 | 8 | 1 | 3 | 7 | 2 | 5 | 4 |

sorted   *unsorted*

| 1 | 6 | 8 | 3 | 7 | 2 | 5 | 4 |

Insertion Sort Time Complexity

◈ Insertion Sort

   ◈ Input: an **array** A of n numbers

   ◈ Output : an **array** A of n numbers in the <u>ascending order</u>

   ◈ Insertion-Sort ($A[1..n]$)

1. for integer $i \leftarrow 1$ to n	Cost: n-1=O(n)
2. for integer $j \leftarrow i$ to 1 with $j > 1$	Cost: $1+\ldots+n-2=O(n^2)$
3. if $A[j-1] > A[j]$ then	Cost: $O(n^2)$
4. swap $A[j-1]$ and $A[j]$	Cost: $O(n^2)$
5. else break	

◈ Insertion sort total cost:

   ◈ $O(n)+O(n^2) +O(n^2) +O(n^2) =O(n^2)$

Bubble Sort

Bubble Sort

- Idea of a bubble sort method
  - For each pass
    - Compare the pair of adjacent item
    - Swap them if they are in the wrong order
  - Repeat the pass through until no swaps are needed

6 5 3 1 8 7 2 4

Bubble Sort Algorithm

◈ BubbleSort (optimized version)

　◈ Input: an **array** A of n numbers

　◈ Output : an **array** A of n numbers in the <u>ascending order</u>

　◈ Bubble-Sort ($A[1..n]$)

　　1. for integer $i \leftarrow 1$ to $n-1$

　　2. 　　for integer $j \leftarrow 2$ to n

　　3. 　　　　if $A[j\text{-}1] > A[j]$ then

　　4. 　　　　　swap $A[j\text{-}1]$ and $A[j]$

| 8 | 6 | 1 | 3 | 7 | 2 | 5 | 4 |

| 1 | 3 | 6 | 2 | 5 | 4 | 7 | 8 |

unsorted sorted

| 6 | 1 | 3 | 7 | 2 | 5 | 4 | 8 |

unsorted sorted

| 1 | 3 | 2 | 5 | 4 | 6 | 7 | 8 |

Bubble Sort Time Complexity

◈ Bubble Sort

  ◈ Input: an **array** A of n numbers

  ◈ Output : an **array** A of n numbers in the <u>ascending order</u>

  ◈ Bubble-Sort ($A[1..n]$)

  1. for integer $i \leftarrow 1$ to $n-1$ Cost: n-1=O(n)

  2.   for integer $j \leftarrow 2$ to n Cost: n-1+...+n-1=O(n^2)

  3.   if $A[j\text{-}1] > A[j]$ then   Cost: O(n^2)

  4.   swap $A[j\text{-}1]$ and $A[j]$ Cost: O(n^2)

◈ Bubble sort total cost:

  ◈ $O(n)+O(n^2) +O(n^2) +O(n^2) =O(n^2)$

Pop Quiz

◈ We say a sorting algorithm is "stable" if it does not change the relative order of elements with equal keys, which of the following is/are stable ()

  A: Selection sort B: Insertion Sort C: Bubble Sort

◈ Watch a video:

  ◈ 1) Which sorting algorithm is used in that video?

  ◈ 2) TB is No. x, so x = ?

Merge Sort
(Divide-and-Conquer)

Divide and Conquer

- Divide and Conquer: an algorithmic technique
  - **Divide:** divide the problem into smaller subproblems
  - **Conquer:** solve each subproblem recursively
  - **Combine:** combine the solution of subproblems into the solution of the original problem

1. divide

3. combine

n

n/b n/b ... n/b

2. conquer
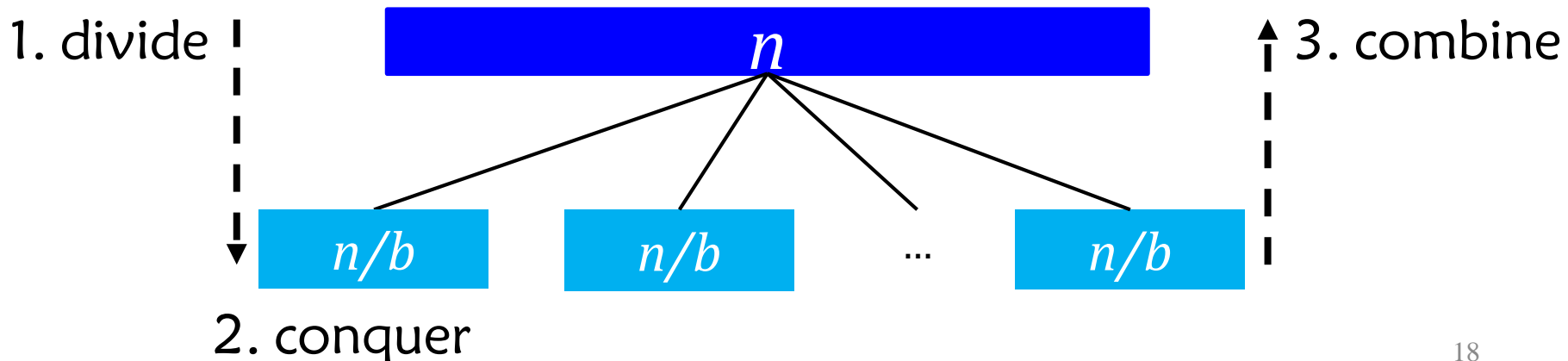
Example: Merge Sort

- Sorting problem
  - Input: an array $A[1..n]$ with n integers
  - Output: a sorted array A (in ascending order)

- Original problem is: sort $A[1..n]$

  | 8 | 6 | 1 | 3 | 7 | 2 | 5 | 4 |

- What is a subproblem?
  - Sort a subarray $A[l..r]$ | 7 | 2 | 5 | 4 |

Merge Sort

◈ Merge Sort

  ◈ **Divide**: divide the array into two subarrays of n/2 numbers each

  ◈ **Conquer**: sort two subarrays recursively

  ◈ **Combine**: merge two sorted subarrays into a sorted array

Merge-Sort(A, n)

1. if $n > 1$
2. $p \leftarrow \lfloor n/2 \rfloor$
3. $B[1..p] \leftarrow A[1..p]$
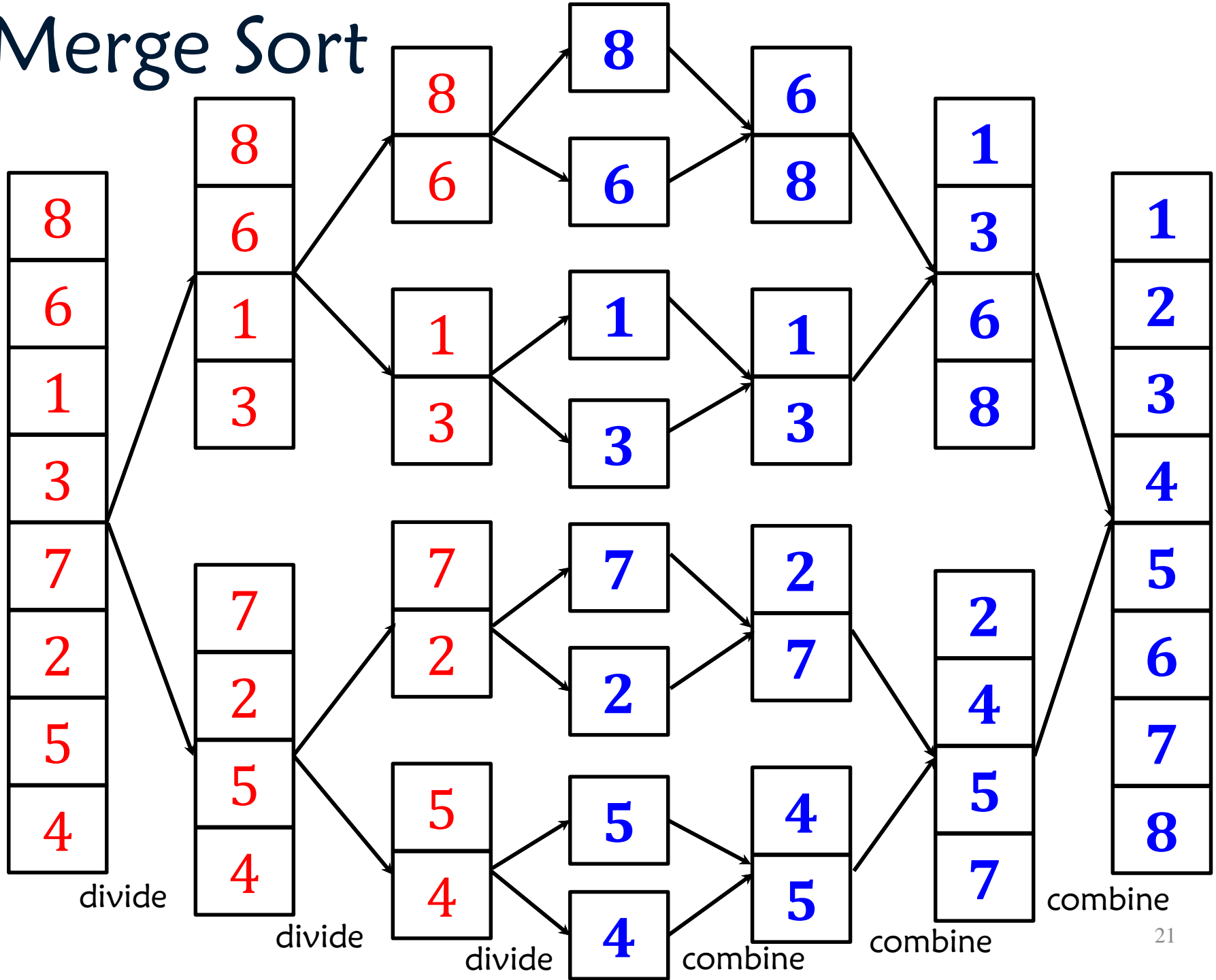4. $C[1..n\text{-}p] \leftarrow A[p+1..n]$
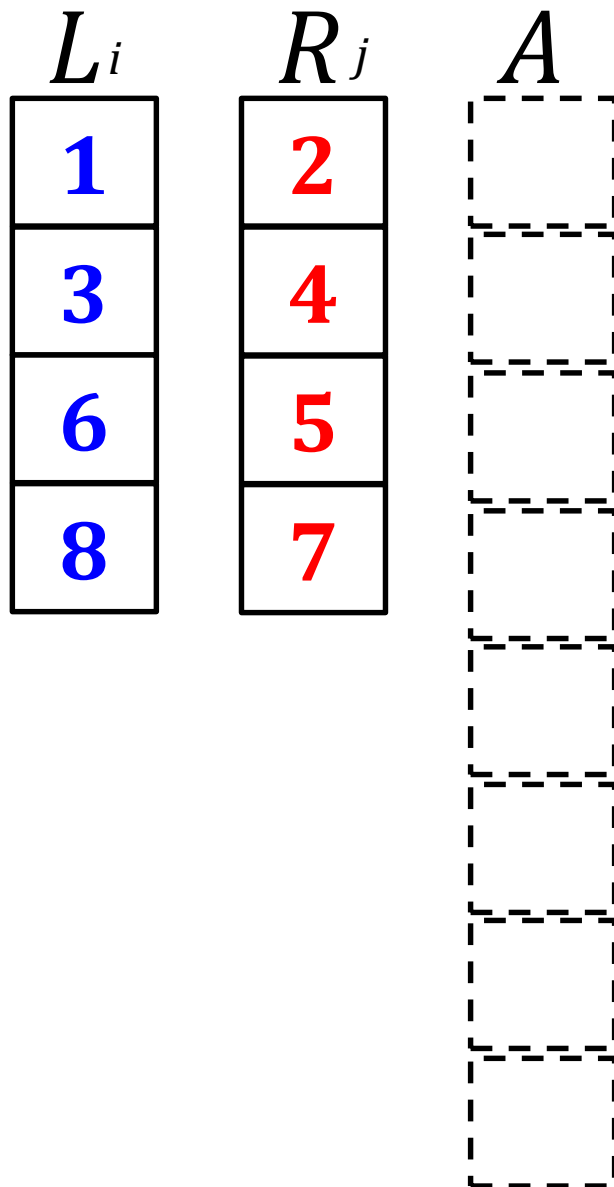5.   Merge-Sort(B, p)
6.   Merge-Sort(C, $n\text{-}p$)
7. $A[1..n] \leftarrow$ Merge(B, p, C, $n\text{-}p$)

We'll discuss the Combine phase ("Merge" function) later

Merge Sort

divide divide divide combine combine combine

21

Merge Sort: Combine Phase

L_i R_j A

L_i
1
3
6
8

R_j
2
4
5
7

Sorted arrays

Merge(L, n_L, R, n_R)

1. n $\leftarrow n_L + n_R$

2. let $A[1..n]$ be a new array

3. $i \leftarrow 1; j \leftarrow 1$

4. for $k \leftarrow 1$ to n

5.   if $i \leq n_L$ and ($j > n_R$ or $L[i] \leq R[j]$)

6. $A[k] \leftarrow L[i]$; $i \leftarrow i + 1$

7.   else

8. $A[k] \leftarrow R[j]$; $j \leftarrow j + 1$

9. return A

Running time of Merge

Sorted arrays

Merge(L, n_L, R, n_R)

1. n ← n_L + n_R
2. let A[1..n] be a new array
3. i ← 1; j ← 1
4. for k ← 1 to n
5.   if $i ≤ n_L$ and ($j > n_R$ or $L[i] ≤ R[j]$)
6.   A[k] ← $L[i]$; i ← i + 1
7.   else
8.   A[k] ← $R[j]$; j ← j + 1
9. return A

◈ Let $n = n_L + n_R$ be the total number of items

◈ Time of merge: $O(n)$ time
  ◈ Line 1: $O(1)$
  ◈ Line 2: $O(n)$
  ◈ Line 3: $O(1)$
  ◈ Lines 4-8: $O(n)$

Running time of Merge Sort

Merge-Sort(*A, n*)

1. if *n* > 1
2.   *p* ← ⌊*n/2*⌋
3.   *B*[*1..p*] ← *A*[*1..p*]
4.   *C*[*1..n-p*] ← *A*[*p+1..n*]
5.   Merge-Sort(*B, p*)
6.   Merge-Sort(*C, n-p*)
7.   *A*[*1..n*] ← Merge(*B, p, C, n-p*)

◈ Let $T(n)$ be the running time of Merge Sort
  - ◈ Lines 3, 4 take $O(n)$ time
  - ◈ Line 5 takes $T(n/2)$ time
  - ◈ Line 6 takes $T(n/2)$ time
  - ◈ Line 7 takes $O(n)$ time

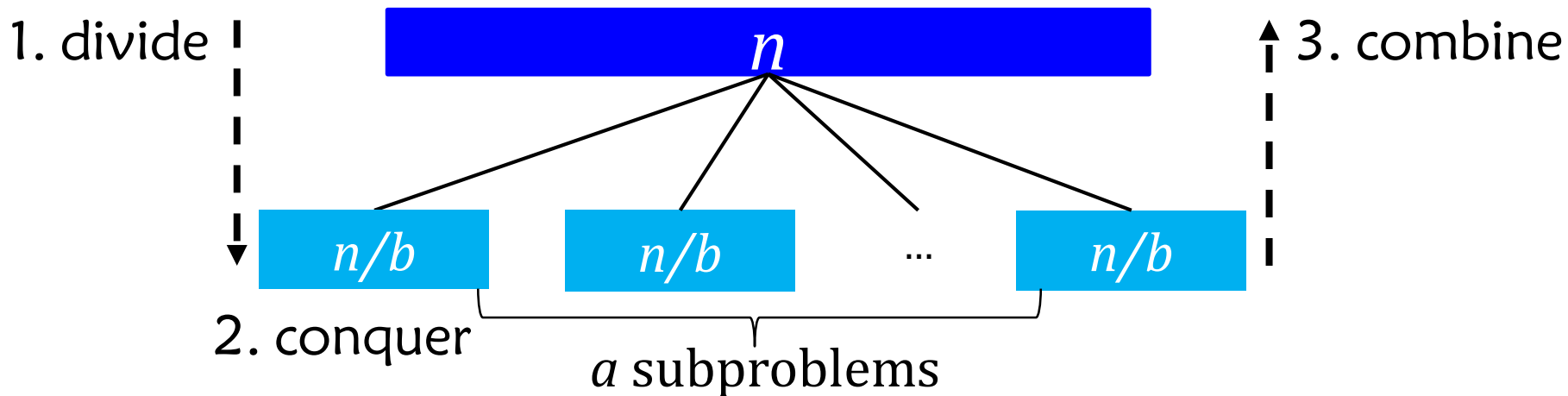◈ Thus, we obtain the recurrence

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

◈ Solving it, we get:

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

Time Complexity

◈ T(n): time complexity of algorithm at input size n

  ◈ Divide the problem into a subproblems

  ◈ Size of each subproblem is n/b

  ◈ Combine phase takes $f(n)$ time

Note: a and b can have different values

1. divide

n

3. combine

n/b n/b ... n/b

2. conquer

a subproblems

◈ Recurrence equation: $T(n) = a\ T(n/b) + f(n)$

◈ E.g., Merge Sort: $T(n) = 2\ T(n/2) + O(n)$

Methods for Solving Recurrences

- Recurrence equation: $T(n) = a\, T(n/b) + f(n)$
- Two methods for solving recurrences
    - Master theorem
    - Substitution method
- Master theorem
    - It could be proved by carefully applying the "expansion method", the details are tedious and omitted from this course
- Substitution method (we skip it here)
    - It is mathematical induction

Master Theorem

- Recurrence equation: T(n) = a T(n/b) + $f(n)$
- Let T(n) be a function that return a positive value for every integer n>0. We know that:

  - $T(1) = O(1)$

  - $T(n) = \alpha T\left(\left\lceil \frac{n}{\beta} \right\rceil\right) + O(n^{\gamma})$ for (n $\geq$ 2)

  where $\alpha \geq 1, \beta > 1,$ and $\gamma \geq 0$. Then:

  - If $\log_{\beta} \alpha < \gamma$, then $T(n) = O(n^{\gamma})$

  - If $\log_{\beta} \alpha = \gamma$, then $T(n) = O(n^{\gamma} \log n)$

  - If $\log_{\beta} \alpha > \gamma$, then $T(n) = O(n^{\log_{\beta} \alpha})$

Master Theorem

- Consider the recurrence of **binary search**:
  - $T(1) \leq c1$
  - $T(n) \leq T(n/2) + c2$ (for $n \geq 2$)
  - Hence, $\alpha = 1$, $\beta = 2$, and $\gamma = 0$. Since $\log_\beta \alpha = \log_2 1 = 0 = \gamma$, we know that $T(n) = O(n^0 \log n) = O(\log n)$.
- Consider the recurrence of **merge sort**:
  - $T(1) \leq c1$
  - $T(n) = 2\ T(n/2) + O(n) = 2\ T(n/2) + c2\ n$ (for $n \geq 2$)
  - Hence, $\alpha = 2$, $\beta = 2$, and $\gamma = 1$. Since $\log_\beta \alpha = \log_2 2 = 1 = \gamma$, we know that $T(n) = O(n^1 \log n) = O(n \log n)$.

Quick Sort
RAM with Randomization

Deterministic & Randomized

* So far in CS203, all our algorithms are deterministic, namely, they do not involve any randomization.

* We will introduce randomized algorithms, e.g., quick sort in the sorting problem.

* Randomized algorithms play an important role in computer science, they often simpler, and sometimes can be provably faster as well.

* Recall the core of the RAM model is a set of atomic operations, we extend this set with:

  * RANDOM(x, y): given integers x and y (x <= y), this operation returns an integer chosen uniformly at random in [x,y], i.e., x, x+1, ..., y has the same probability of being returned.

Randomized Algorithm Example

⬥ Find-a-Zero: Given an array of integers with size n, among which there is at least 0. Design an algorithm to report an arbitrary position of A that contains a 0

⬥ Suppose A = (9,18,0,0,15,0), an algorithm can report 3,4 or 6, consider the following randomized algorithm

⬥ 1. do

⬥ 2.   r ← RANDOM(1,n)

⬥ 3. until A[r]=0

⬥ 4. return r

⬥ What is the cost of the algorithm? It depends

　⬥ If all numbers in A are 0, O(1) time. If A has only one 0, O(n) expected time.

　⬥ As before, we care about the worst expected time: O(n)

Quick Sort

- Idea of a quick sort method
  - Randomly pick an integer p in A, call it the pivot
  - Re-arrange the integers in an array A' such that
    - All the integers smaller than p are positioned before p in A'
    - All the integers larger than p are positioned after p in A'
  - Sort the part of A' before p recursively
  - Sort the part of A' after p recursively

Quicksort

- Quick Sort
  - Input: an **array** A of n numbers
  - Output : an **array** A of n numbers in the <u>ascending order</u>
  - Quicksort ($A[1..n]$, lo=1, hi=n)
    1. p ← partition(A, lo, hi)
    2. Quicksort(A,lo,p-1)
    3. Quicksort(A,p+1,hi)

Quicksort

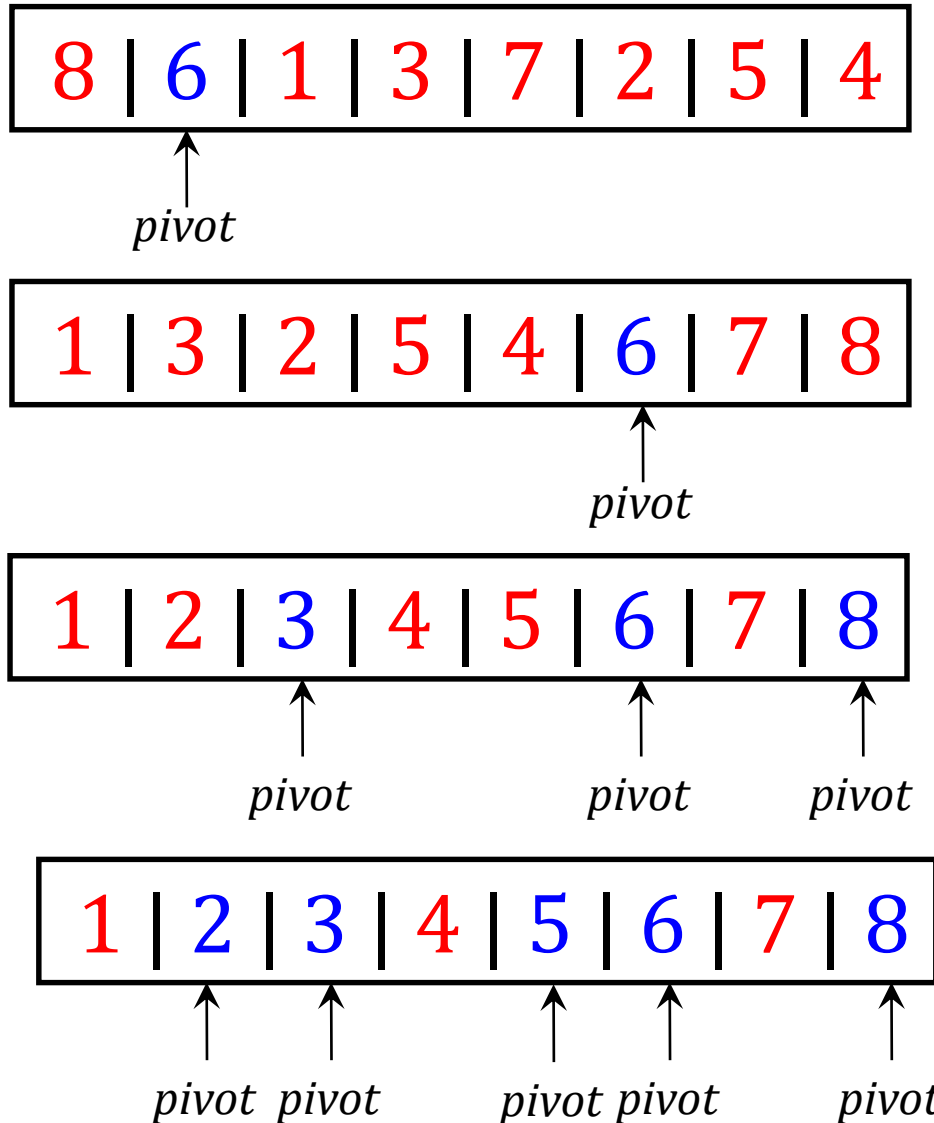◈ Partition(A, lo, hi)

1. p ← RANDOM(lo, hi); pivot ← A[p];

2. L ← lo, R← hi

3. for integer i from lo to hi

4.　　if(i!=p)

5.　　　　if(A[i] < pivot) A'[L++]←A[i]

6.　　　　else A'[R--]←A[i]

7. A'[L] ← pivot

8. A[lo, hi] ← A'

9. return L;

◈ Question:

◈ If we set p ← lo or hi in Line 1, quick sort is still correct ?

◈ What are the difference between p ← lo/hi and p ← RANDOM(lo, hi)?

Quicksort Example

$$8 \mid 6 \mid 1 \mid 3 \mid 7 \mid 2 \mid 5 \mid 4$$

pivot

$$1 \mid 3 \mid 2 \mid 5 \mid 4 \mid 6 \mid 7 \mid 8$$

pivot

$$1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8$$

pivot   *pivot*   *pivot*

$$1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8$$

pivot pivot   *pivot pivot*   *pivot*

Quicksort Time Complexity

◈ Quicksort's running time is not attractive in the worst case: it is $O(n^2)$ (why?) However, quick sort is fast in expectation, i.e., O(nlogn), remember this holds for every input array A.

◈ Whether quicksort has any advantage over merge sort? which guarantees O(nlogn) in the worst case.

◈ No in theory, but there is an advantage in practice

◈ Quicksort permits a faster implementation that leads to a smaller hidden constant compared to merge sort. (why?)

Quicksort Time Complexity

◈ Let X be the number of comparisons in quicksort algorithm. The running is bounded by O(n+x).

◈ We prove that E[X]=O(n log n)

◈ Denote e_i be the i-th smallest integer in A, consider e_i and e_j for any i,j such that i!=j

◈ What is the probability that quicksort compares e_i and e_j?

  ◈ Every element will be selected as pivot precisely once

  ◈ e_i and e_j are not compared, if any element between them gets selected as a pivot before them.

  ◈ Therefore, e_i and e_j are compared if and only if either one is the first among e_i , e_{i+1} ,…, e_j picked as a pivot

  ◈ The probability is 2/(j-i+1) (random pivot selection)

Quicksort Time Complexity

- Define random variable X_{ij} to be 1, if e_i and e_j are compared. Otherwise, X_{ij} to be 0. Thus, we have

- $\Pr[X_{ij} = 1] = 2/(j-i+1)$, that is $E[X_{ij}] = 2/(j-i+1)$

- Since $X = \sum_{i,j} X_{ij}$, hence:

- $E[X] = \sum_{i,j} E[X_{ij}] = \sum_{i,j} \dfrac{2}{j-i+1}$

- $= 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \dfrac{1}{j-i+1}$

- $= 2 \sum_{i=1}^{n-1} O(\log(j-i+1))$ $(1+1/2+...+1/n = O(\log n))$

- $= 2 \sum_{i=1}^{n-1} O(\log n)$

- $= O(n \log n)$

- Harmonic series: $1+1/2+...+1/n$, which is frequently encountered in computer science.

Summary

Sort	Average	Space
Selection	$O(n^2)$	$O(1)$
Insertion	$O(n^2)$	$O(1)$
Bubble	$O(n^2)$	$O(1)$
Heap	$O(n\log n)$	$O(1)$
Merge	$O(n\log n)$	Depends
Quick	$O(n\log n)$	$O(1)$

◈ Comparison lower bound of sorting algorithm: $\Omega(n \log n)$

◈ We omit the proof here.

Other Sorting Methods

Other Sorting Algorithms

◈ Counting sort (Chapter 8.2)

  ◈ it is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in $O(|S| + n)$ time and $O(|S|)$ memory where n is the length of the input.

◈ Radix sort (Chapter 8.3)

  ◈ radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in $O(n \cdot k)$ time

◈ Bucket sort (Chapter 8.4)

  ◈ Bucket sort is a divide and conquer sorting algorithm that generalizes counting sort by partitioning an array into a finite number of buckets.

Thank You!