Divide and Conquer (Merge Sort)

- Divide and conquer
- Merge sort
- Loop-invariant
- Recurrence relations

# **Divide and Conquer**

- Recursive in structure
  - *Divide* the problem into sub-problems that are similar to the original but smaller in size
  - *Conquer* the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - *Combine* the solutions of the sub-problems to create a global solution to the original problem

### An Example: Merge Sort

**Sorting Problem:** Sort a sequence of *n* elements into non-decreasing order.

- *Divide*: Divide the *n*-element sequence to be sorted into two subsequences of *n*/2 elements each
- *Conquer:* Sort the two subsequences recursively using merge sort.
- *Combine*: Merge the two sorted subsequences to produce the sorted answer.

### <u>Merge Sort – Example</u>



# Merge-Sort (A, p, r)

# **INPUT:** a sequence of *n* numbers stored in array A **OUTPUT:** an ordered sequence of *n* numbers

MergeSort (A, p, r)// sort A[p..r] by divide & conquer1if p < r2then  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 3MergeSort (A, p, q)4MergeSort (A, q+1, r)5Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]

**Initial Call:** *MergeSort*(*A*, 1, *n*)

### Procedure Merge

Merge(A, p, q, r)1  $n_1 \leftarrow q - p + 1$  $2 n_2 \leftarrow r - q$ for  $i \leftarrow 1$  to  $n_1$ 3 **do**  $L[i] \leftarrow A[p+i-1]$ 4 5 for  $j \leftarrow 1$  to  $n_2$ 6 **do**  $R[j] \leftarrow A[q+j]$ 7  $L[n_1+1] \leftarrow \infty$  $R[n_2+1] \leftarrow \infty$ 8 9  $i \leftarrow 1$  $j \leftarrow 1$ 10 **for**  $k \leftarrow p$  **to** r11 **do if**  $L[i] \leq R[j] \leftarrow$ 12 13 then  $A[k] \leftarrow L[i]$ 14  $i \leftarrow i + 1$ else  $A[k] \leftarrow R[j]$ 15  $j \leftarrow j + 1$ 16

Input: Array containing sorted subarrays A[p .. q]and A[q+1 .. r].

Output: Merged sorted subarray in A[p ... r].

**Sentinels**, to avoid having to check if either subarray is fully copied at each step.



# **Correctness of Merge**

Merge(A, p, q, r)1  $n_1 \leftarrow q - p + 1$  $2 n_2 \leftarrow r - q$ for  $i \leftarrow 1$  to  $n_1$ 3 **do**  $L[i] \leftarrow A[p+i-1]$ 4 for  $j \leftarrow 1$  to  $n_2$ 5 **do**  $R[j] \leftarrow A[q+j]$ 6 7  $L[n_1+1] \leftarrow \infty$  $R[n_2+1] \leftarrow \infty$ 8 9  $i \leftarrow 1$  $j \leftarrow 1$ 10 **for**  $k \leftarrow p$  **to** r11 12 **do if**  $L[i] \leq R[j]$ 13 then  $A[k] \leftarrow L[i]$ 14  $i \leftarrow i + 1$ else  $A[k] \leftarrow R[j]$ 15  $j \leftarrow j + 1$ 16

### Loop Invariant for the *for* loop

• At the start of each iteration of the for loop:

subarray A[p . . k - 1]contains the k - p smallest elements of *L* and *R* in sorted order.

• *L*[*i*] and *R*[*j*] are the smallest elements of *L* and *R* that have not been copied back into *A*.

### **Initialization:**

Before the first iteration:

- A[p ... k 1] is empty.
- i = j = 1.
- *L*[1] and *R*[1] are the smallest elements of *L* and *R* not copied to *A*.

# **Correctness of Merge**

Merge(A, p, q, r)1  $n_1 \leftarrow q - p + 1$  $2 n_2 \leftarrow r - q$ for  $i \leftarrow 1$  to  $n_1$ 3 **do**  $L[i] \leftarrow A[p+i-1]$ 4 for  $j \leftarrow 1$  to  $n_2$ 5 6 **do**  $R[j] \leftarrow A[q+j]$ 7  $L[n_1+1] \leftarrow \infty$ 8  $R[n_2+1] \leftarrow \infty$ 9  $i \leftarrow 1$  $j \leftarrow 1$ 10 for  $k \leftarrow p$  to r11 **do if**  $L[i] \leq R[j]$ 12 13 **then**  $A[k] \leftarrow L[i]$ 14  $i \leftarrow i + 1$ else  $A[k] \leftarrow R[j]$ 15  $j \leftarrow j + 1$ 16

#### Maintenance:

(We will prove that if after the *k*th iteration, the Loop Invariant (LI) holds, we still have the LI after the (*k*+1)th iteration.)

**Case 1:**  $L[i] \le R[j]$ •By Loop Invariant, *A* contains k - psmallest elements of *L* and *R* in *sorted order*. •Also, L[i] and R[j] are the smallest elements of *L* and *R* not yet copied into *A*. •Line 13 results in *A* containing k - p + 1smallest elements (again in sorted order). Incrementing *i* and *k* reestablishes the LI for the next iteration. **Similarly for Case 2:** L[i] > R[j].

### **Correctness of Merge**

Merge(A, p, q, r)	
$1  n_1$	$\leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$	
3	<b>for</b> $i \leftarrow 1$ <b>to</b> $n_1$
4	<b>do</b> $L[i] \leftarrow A[p+i-1]$
5	<b>for</b> $j \leftarrow 1$ <b>to</b> $n_2$
6	<b>do</b> $R[j] \leftarrow A[q+j]$
7	$L[n_1+1] \leftarrow \infty$
8	$R[n_2+1] \leftarrow \infty$
9	$i \leftarrow 1$
10	$j \leftarrow 1$
11	<b>for</b> $k \leftarrow p$ <b>to</b> $r$
12	<b>do if</b> $L[i] \leq R[j]$
13	<b>then</b> $A[k] \leftarrow L[i]$
14	$i \leftarrow i + 1$
15	else $A[k] \leftarrow R[j]$
16	$j \leftarrow j + 1$

#### **Maintenance:**

**Case 1:**  $L[i] \le R[j]$ •By Loop Invariant (LI), *A* contains k - psmallest elements of *L* and *R* in *sorted order*. •By LI, L[i] and R[j] are the smallest elements of *L* and *R* not yet copied into *A*. •Line 13 results in *A* containing k - p + 1smallest elements (again in sorted order). Incrementing *i* and *k* reestablishes the LI for the next iteration. **Similarly for Case 2:** L[i] > R[j].

#### **Termination:**

•On termination, k = r + 1.

•By LI, A contains r - p + 1 smallest elements of L and R in sorted order.

•*L* and *R* together contain r - p + 3 - (r - p + 1) = 2 elements.

All but the two sentinels have been copied back into *A*.

- Reduction of data movements
- Non-recursive Algorithm

### Y. Chen, and R. Su, Merge Sort Revisited, ACTA Scientific Computer Sciences, Vol. 4, No. 5, pp. 49 - 52, 2022.

• Reduction of data movements

We notice that in the procedure merge() of Merge sort the copying of A[q + 1 .. r] into R is not necessary, since we can directly merge L and A[q + 1 .. r] and store the merged, but sorted sequence back into A.







9/11/2024





### Why does it work?

- Denote by A' the sorted version of A. Denote by A'(i, j) a prefix of A' which contains the first i elements from L and first j elements from A[q + 1 .. r].
- Obviously, we can store A'(i, j) in A itself since after the *j*th element (from A[q + 1 .. r]) has been inserted into A', the first q - p + j + 1 entries in A are empty and q - p + 1 ≥ i (thus, q - p + j + 1 ≥ i + j).

Algorithm: mergeImpr (A, p, q, r)Input: Both A[p .. q] and A[q + 1 .. r] are sorted; but A as a whole is not sorted

**Output :** sorted *A* 

1.  $n_1 := q - p + 1; n_2 := r - p + 1; k := p;$ 2. let  $L[1 ... n_1]$  be a new array; When going out of while-loop, 3. for i = 1 to  $n_1$  do we distinguish between two cases: 4. L[i] := A[p + i - 1] $i > n_1$ , 5. i := p; j := q + 1; $j > n_2$ . while  $i \leq n_1$  and  $j \leq n_2$  do **6**. if  $L[i] \leq A[j]$  then  $\{A[k] := L[i]; i := i + 1;\}$ 7. else {A[k] := A[j]; j := j + 1;} 8. 9. k := k + 1;if  $j > n_2$  then 10.

11. copy the remaining elements in L into A[k ... r];

### Non-recursive algorithm

•Merge Sort can be further improved by replacing its recursive calls with a series of merging operations, by which the recursive execution of the algorithm is simulated.

The whole working process can be divided into [log<sub>2</sub> n] phases.
In the first phase, we will make [n/2] merging operations with each merging two single-element sequences together.

In the second phase, we will make [n/4] merging operations with each merging two two-element sequences together, and so on.
Finally, we will make only one operation to merge two sorted subsequences to form a globally sorted sequence. Between the sorted subsequences, one contains [n/2] elements while the other contains [n/2] elements.

Algorithm: *mSort* (A) **Input** : *A* - a sequence of elements stored as an array; Output : sorted A **1.** if  $|A| \leq 1$  then return A; r: the length of A **2**. r := |A|;3.  $l := \lceil \log_2 r \rceil$ ; *l* : the number of passes *j* : the number of elements involved **4**. j := 2;in a merging process in a pass **5.** for i = 1 to l do 6. for k = 1 to  $\lceil r/j \rceil$ ) do 7. s := |(k - 1)j|;8. mergeImpr(A, s + 1, s + [j/2], s + j);9. j := 2j;



9/11/2024

# Analysis of Merge Sort

- Running time **T**(**n**) of Merge Sort:
- Divide: computing the middle takes  $\Theta(1)$
- Conquer: solving 2 subproblems takes 2T(n/2)
- Combine: merging *n* elements takes  $\Theta(n)$
- Total:

 $T(n) = \Theta(1)$ if n = 1 $T(n) = 2T(n/2) + \Theta(n)$ if n > 1

 $\Rightarrow$  *T*(*n*) =  $\Theta(n \lg n)$  (CLRS, Chapter 4)

### Recurrences – I

9/11/2024

### **Recurrence Relations**

- Equation or an inequality that characterizes a function by its values on smaller inputs.
- Solution Methods (Chapter 4)
  - Substitution Method.
  - Recursion-tree Method.
  - Master Method.
- Recurrence relations arise when we analyze the running time of iterative or recursive algorithms.
  - Ex: Divide and Conquer.  $T(n) = \Theta(1)$ T(n) = a T(n/b) + D(n)

if  $n \le c$ otherwise

# **Substitution Method**

- <u>Guess</u> the form of the solution, then <u>use mathematical induction</u> to show it correct.
  - Substitute guessed answer for the function when the inductive hypothesis is applied to smaller values.
- Works well when the solution is easy to guess.
- No general way to guess the correct solution.

### **Example – Exact Function**

if n = 1Recurrence: T(n) = 1T(n) = 2T(n/2) + n if n > 1•<u>Guess:</u>  $T(n) = n \lg n + n$ . •Induction: •Basis:  $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$ . •Hypothesis:  $T(k) = k \lg k + k$  for all k < n. •**Inductive Step:** T(n) = 2 T(n/2) + n $= 2 ((n/2) \lg(n/2) + (n/2)) + n$ = n (lg(n/2)) + 2n $= n \lg n - n + 2n$  $= n \lg n + n$ 

# **Recursion-tree Method**

- Making a good guess is sometimes difficult with the substitution method.
- Use **recursion trees** to devise good guesses.
- Recursion Trees
  - Show successive expansions of recurrences using trees.
  - Keep track of the time spent on the subproblems of a divide and conquer algorithm.
  - Help organize the algebraic bookkeeping necessary to solve a recurrence.

### <u>Recursion Tree – Example</u>

### • Running time of Merge Sort: $T(n) = \Theta(1)$ if n = 1 $T(n) = 2T(n/2) + \Theta(n)$ if n > 1

• Rewrite the recurrence as

$$T(n) = \mathbf{c} \qquad \qquad \text{if } n = 1$$

T(n) = 2T(n/2) + cn if n > 1

c > 0: Running time for the base case and time per array element for the divide and combine steps.

# **Recursion Tree for Merge Sort**

For the original problem, we have a cost of *cn*, plus two subproblems each of size (n/2) and running time T(n/2).

Each of the size n/2 problems has a cost of cn/2 plus two subproblems, each costing T(n/4).

**C***n* 



### **Recursion Tree for Merge Sort**

Continue expanding until the problem size reduces to 1.



Total: *cn*lg*n*+*cn* 

# **Recursion Tree for Merge Sort**

Continue expanding until the problem size reduces to 1.



Each level has total cost *cn*.
Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves ⇒ *cost per level remains the same*.
There are lg *n* + 1 levels, height is lg *n*. (Assuming *n* is a power of 2.)
Can be proved by induction.
Total cost = sum of costs at each

level =  $(\lg n + 1)cn = cn\lg n + cn = \Theta(n \lg n)$ .

# **Other Examples**

- Use the recursion-tree method to determine a guess for the recurrences
  - $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2).$
  - T(n) = T(n/3) + T(2n/3) + O(n).

### Recursion Trees – Caution Note

- Recursion trees only generate guesses.
  - Verify guesses using substitution method.
- A small amount of "sloppiness" can be tolerated. <u>Why?</u>
- If careful when drawing out a recursion tree and summing the costs, it can be used as direct proof.

# The Master Method

- Based on the Master theorem.
- "Cookbook" approach for solving recurrences of the form
  - T(n) = aT(n/b) + f(n)
  - $a \ge 1, b > 1$  are constants.
  - *f*(*n*) is asymptotically positive.
  - *n/b* may not be an integer, but we ignore floors and ceilings. <u>Why?</u>
- Requires memorization of three cases.

### The Master Theorem

### **Theorem 4.1**

- Let  $a \ge 1$  and b > 1 be constants, let f(n) be a function, and let T(n) be defined on nonnegative integers by the recurrence T(n) = aT(n/b) + f(n), where we can replace n/b by  $\lfloor n/b \rfloor$  or  $\lceil n/b \rceil$ . T(n) can be bounded asymptotically in three cases:
- 1. If  $f(n) = O(n^{\log_b a \varepsilon})$  for some constant  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$ .
- 2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \lg n)$ .
- 3. If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$ , and if, for some constant c < 1 and all sufficiently large n, we have  $a \cdot f(n/b) \le c f(n)$ , then  $T(n) = \Theta(f(n))$ .

We'll return to recurrences as we need them...