

CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #3

➤ Divide-and-Conquer

Prof. Pietro S. Oliveto

Department of Computer Science and Engineering

Southern University of Science and Technology (SUSTech)

`olivetop@sustech.edu.cn`

<https://faculty.sustech.edu.cn/olivetop>

Reading: Section 2.3 and Section 4.5

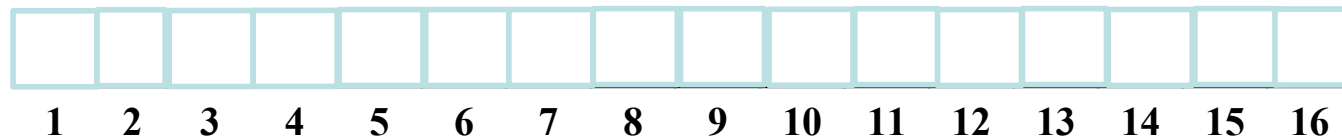
(optional: lots more details in Chapter 4)

Aims of this lecture

- To introduce the divide-and-conquer design paradigm.
- To introduce the MergeSort algorithm – a recursive algorithm using divide-and-conquer.
- To show how to prove correctness for a recursive algorithm
- To show how to analyse the runtime of recursive algorithms using recurrence equations.
- To show how to solve recurrence equations

➤ Problem: Find a number in a sorted array

- I have a sorted array of integers;



- Is the number 40 in the array?
- If we scan the array from the beginning to the end what is the worst case runtime? $\theta(n)$ – linear search
- What if we always check the middle point and discard the “wrong” half of the subarray? $2^k = n \Rightarrow \theta(\log n)$ – binary search
- By **dividing** the problem size by half at each step we have reduced the runtime of the algorithm from linear to **logarithmic!**

➤ Design Paradigms

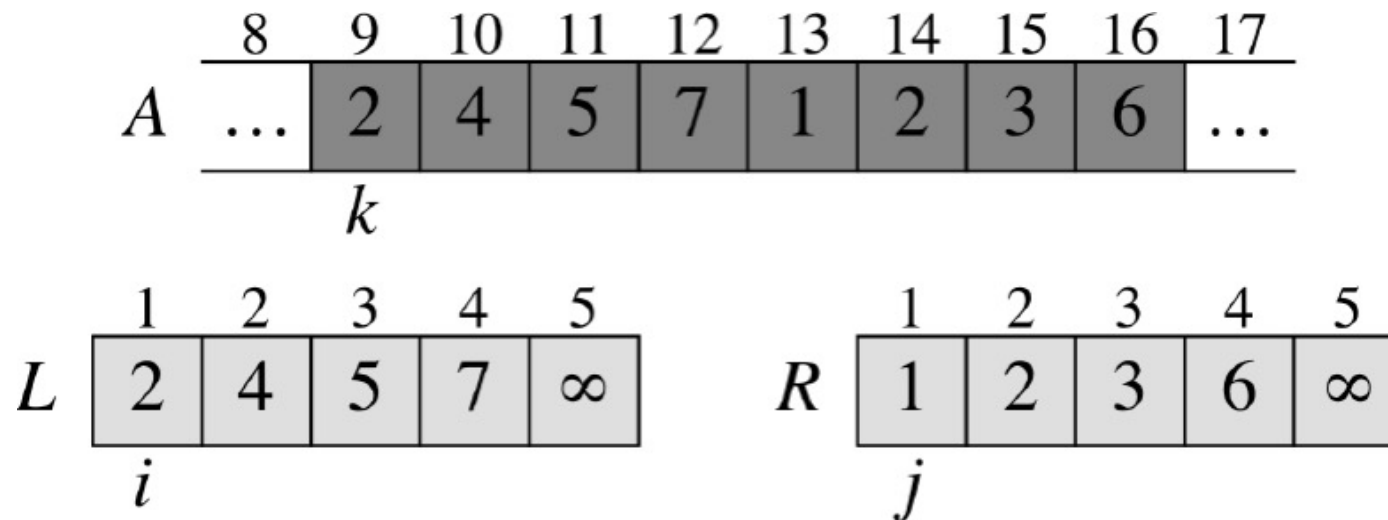
- InsertionSort used an incremental approach:
 - Having sorted the subarray $A[1..j-1]$, we inserted $A[j]$ into its proper place, yielding the sorted subarray $A[1..j]$.
 - **Idea: incrementally build up** a solution to the problem.
- Alternative design approach: **divide-and-conquer**
 1. **Divide:** Break the problem into smaller subproblems, smaller instances of the original problem.
 2. **Conquer:** Solve these problems recursively.
 3. **Combine** the solutions to subproblems into the solution for the original problem.

MergeSort

- MergeSort - sorting using **divide-and-conquer**:
 1. **Divide** the n -element sequence to be sorted into two subsequences of $n/2$ elements each.
 2. **Conquer**: Sort the two subsequences **recursively** using MergeSort.
 3. **Combine**: **merge** the two subsequences to produce the sorted answer.
- The recursion stops when the sequence is just 1 element.
- Key here is the procedure **Merge**
- Tedious bit: copying elements between arrays.


➤ Merge(A, p, q, r)

- Assume subarrays $A[p \dots q]$ and $A[q + 1 \dots r]$ are sorted.
- Copy these subarrays to new arrays L and R .
- Both L and R contain an additional element ∞ at the end (“sentinel”), so we don’t have to check for end of array.
- Merge L and R back into A by comparing $L[i]$ and $R[j]$.



MERGE(A, p, q, r)

```
1:  $n_1 = q - p + 1$ 
2:  $n_2 = r - q$ 
3: let  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$  be new arrays
4: for  $i = 1$  to  $n_1$  do
5:    $L[i] = A[p + i - 1]$ 
6: for  $j = 1$  to  $n_2$  do
7:    $R[j] = A[q + j]$ 
8:  $L[n_1 + 1] = \infty$ 
9:  $R[n_2 + 1] = \infty$ 
10:  $i = 1$ 
11:  $j = 1$ 
12: for  $k = p$  to  $r$  do
13:   if  $L[i] \leq R[j]$  then
14:      $A[k] = L[i]$ 
15:      $i = i + 1$ 
16:   else
17:      $A[k] = R[j]$ 
18:      $j = j + 1$ 
```



Set up arrays L
and R (boring)



Actual merge

New book pseudo-code without sentinels

```
MERGE( $A, p, q, r$ )
1   $n_L = q - p + 1$            // length of  $A[p : q]$ 
2   $n_R = r - q$                // length of  $A[q + 1 : r]$ 
3  let  $L[0 : n_L - 1]$  and  $R[0 : n_R - 1]$  be new arrays
4  for  $i = 0$  to  $n_L - 1$  // copy  $A[p : q]$  into  $L[0 : n_L - 1]$ 
5       $L[i] = A[p + i]$ 
6  for  $j = 0$  to  $n_R - 1$  // copy  $A[q + 1 : r]$  into  $R[0 : n_R - 1]$ 
7       $R[j] = A[q + j + 1]$ 
8   $i = 0$                      //  $i$  indexes the smallest remaining element in  $L$ 
9   $j = 0$                      //  $j$  indexes the smallest remaining element in  $R$ 
10  $k = p$                      //  $k$  indexes the location in  $A$  to fill
11 // As long as each of the arrays  $L$  and  $R$  contains an unmerged element,
12 //   copy the smallest unmerged element back into  $A[p : r]$ .
13 while  $i < n_L$  and  $j < n_R$ 
14     if  $L[i] \leq R[j]$ 
15          $A[k] = L[i]$ 
16          $i = i + 1$ 
17     else  $A[k] = R[j]$ 
18          $j = j + 1$ 
19          $k = k + 1$ 
20 // Having gone through one of  $L$  and  $R$  entirely, copy the
21 //   remainder of the other to the end of  $A[p : r]$ .
22 while  $i < n_L$ 
23      $A[k] = L[i]$ 
24      $i = i + 1$ 
25      $k = k + 1$ 
26 while  $j < n_R$ 
27      $A[k] = R[j]$ 
28      $j = j + 1$ 
29      $k = k + 1$ 
```


➤ Runtime of Merge

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

MERGE(A, p, q, r)

```
1:  $n_1 = q - p + 1$ 
2:  $n_2 = r - q$ 
3: let  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$  be new arrays
4: for  $i = 1$  to  $n_1$  do
5:    $L[i] = A[p + i - 1]$ 
6: for  $j = 1$  to  $n_2$  do  $\Theta(n)$ 
7:    $R[j] = A[q + j]$ 
8:  $L[n_1 + 1] = \infty$ 
9:  $R[n_2 + 1] = \infty$ 
10:  $i = 1$ 
11:  $j = 1$ 
12: for  $k = p$  to  $r$  do  $\Theta(n)$ 
13:   if  $L[i] \leq R[j]$  then
14:      $A[k] = L[i]$ 
15:      $i = i + 1$ 
16:   else
17:      $A[k] = R[j]$ 
18:      $j = j + 1$ 
```

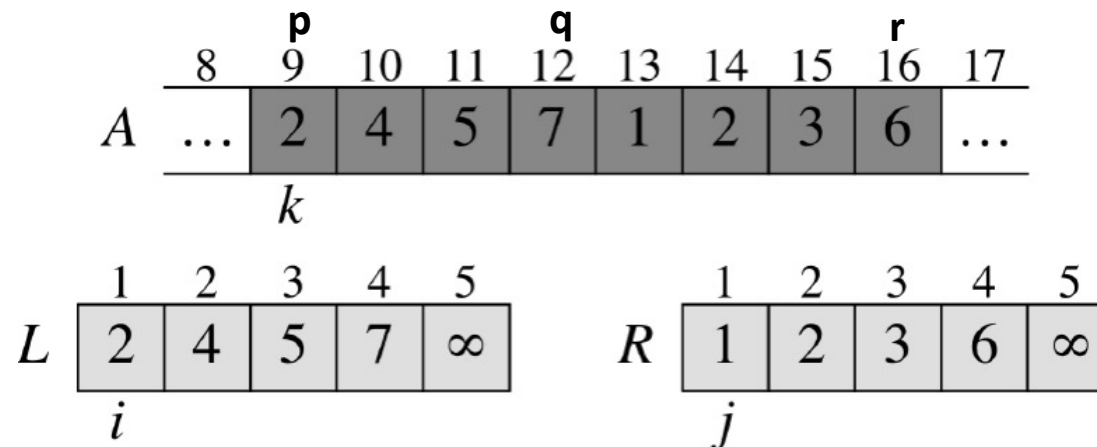
Set up arrays L
and R (boring)

only 1 loop

Actual merge

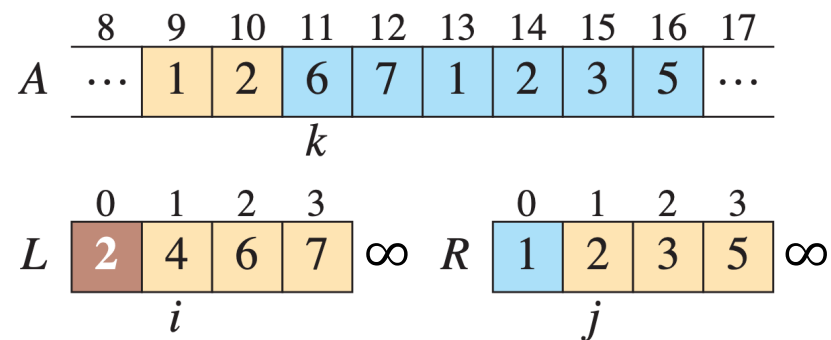
➤ Correctness of Merge (1)

- **Loop invariant:** At the start of the iteration of the last for loop,
 - the subarray $A[p \dots k - 1]$ contains the $k - p$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order and
 - $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to A .
- **Initialisation:** the loop starts with $k = p$, hence $A[p \dots k - 1]$ is empty and contains the $k - p = 0$ smallest elements of L, R . As $i = j = 1$, $L[i]$ and $R[j]$ are the smallest uncopied elements.



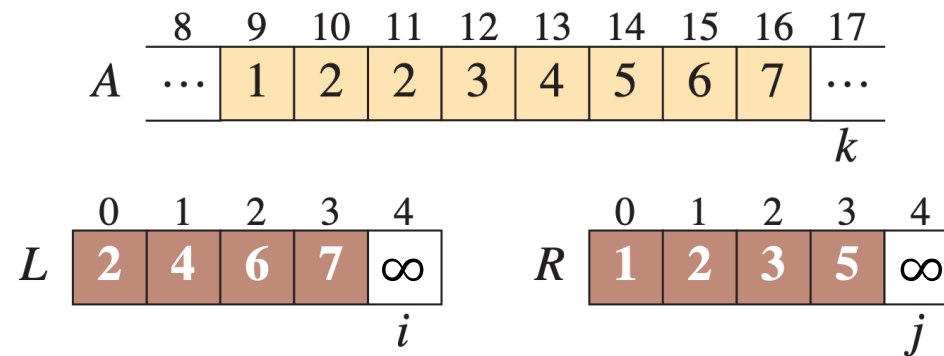
➤ Correctness of Merge (2)

- **Loop invariant:** At the start of the iteration of the last for loop,
 - the subarray $A[p \dots k - 1]$ contains the $k - p$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order and
 - $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to A .
- **Maintenance:** suppose $L[i] \leq R[j]$. Then $L[j]$ is the smallest element not copied back. $A[p \dots k - 1]$ contains the $k - p$ smallest elements, and after copying $L[j]$ into $A[k]$, $A[p \dots k]$ contains the $k - p + 1$ smallest elements. Incrementing k and j re-establishes the loop condition.
Argue similarly for $R[j] < L[i]$.



➤ Correctness of Merge (3)

- **Loop invariant:** At the start of the iteration of the last for loop,
 - the subarray $A[p \dots k - 1]$ contains the $k - p$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order and
 - $L[i]$ and $R[i]$ are the smallest elements of their arrays that have not been copied back to A .
- **Termination:** at termination, $k = r + 1$. By the loop invariant, $A[p \dots k - 1] = A[p \dots r]$ contains the $k - p = r - p + 1 = n_1 + n_2$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order. That's all elements in L and R apart from the two ∞ .



➤ MergeSort: The Complete Algorithm

Notation: $\lfloor x \rfloor$ means “floor of x ” (rounding down).

MERGE_SORT(A, p, r)

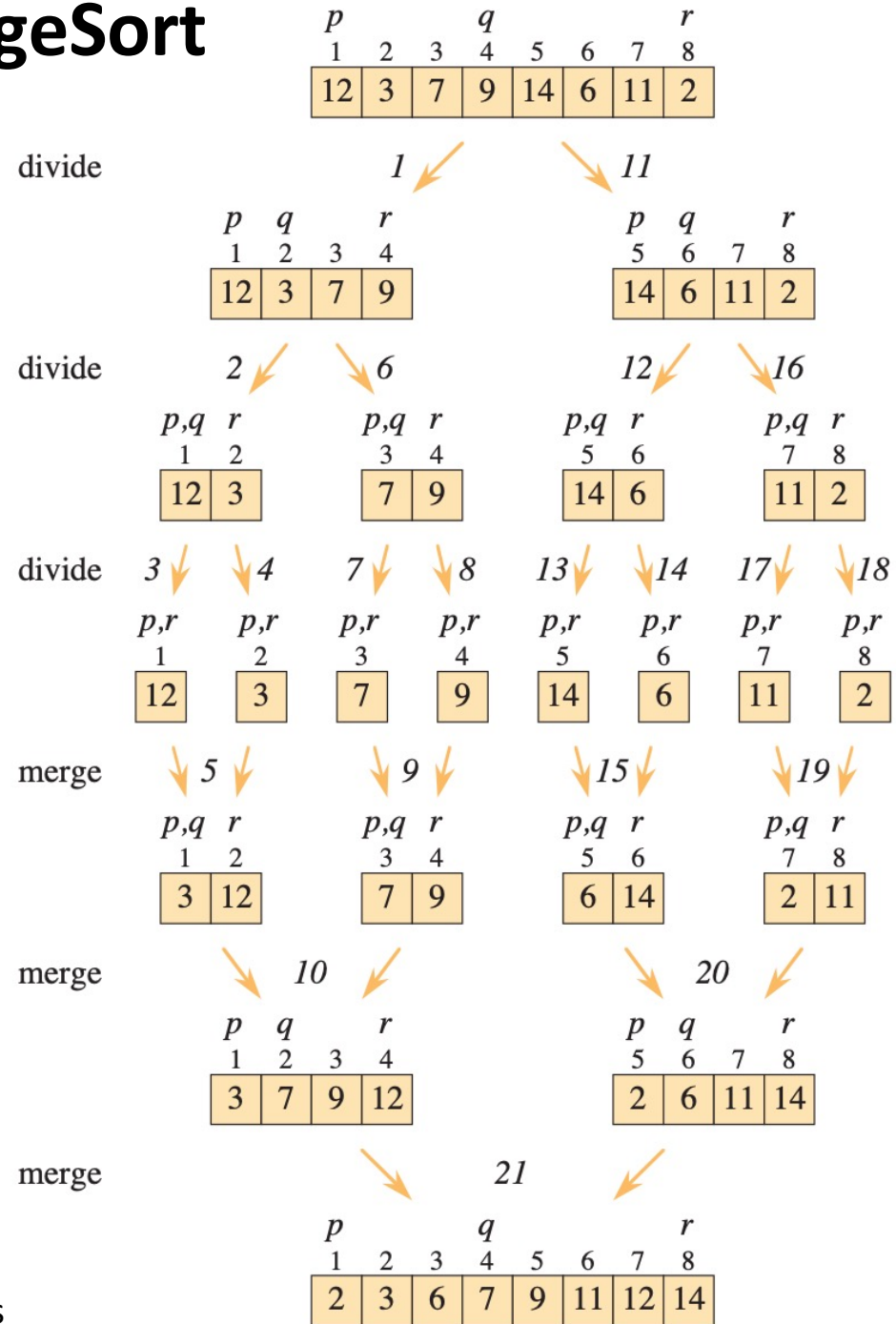
```
1: if  $p < r$  then  
2:    $q = \lfloor (p + r) / 2 \rfloor$   
3:   MERGE_SORT( $A, p, q$ )  
4:   MERGE_SORT( $A, q + 1, r$ )  
5:   MERGE( $A, p, q, r$ )
```

Initial call: MERGE_SORT($A, 1, A.length$)

➤ Operation of MergeSort

MERGE_SORT(A, p, r)

- 1: **if** $p < r$ **then**
 - 2: $q = \lfloor (p + r) / 2 \rfloor$
 - 3: MERGE_SORT(A, p, q)
 - 4: MERGE_SORT($A, q + 1, r$)
 - 5: MERGE(A, p, q, r)
-



➤ Correctness of MergeSort

Proof by Induction:

Weak induction

- **Base case:** Show statement true for initial case: $n=a$ (usually $n=0$ or $n=1$)
- **Inductive step:** If assumed true for n and can show true for $n+1$ then true for all $n \geq a$

Strong induction

- **Base case:** Show statement true for initial case: $n=a$ (usually $n=0$ or $n=1$)
- **Inductive step:** If assumed true for all $a \leq k \leq n$ and can show true for $n+1$ then true for all $n \geq a$

Strong induction can be proved using **Weak induction**

MERGE_SORT(A, p, r)

```
1: if  $p < r$  then
2:    $q = \lfloor (p + r) / 2 \rfloor$ 
3:   MERGE_SORT( $A, p, q$ )
4:   MERGE_SORT( $A, q + 1, r$ )
5:   MERGE( $A, p, q, r$ )
```

➤ Correctness of MergeSort

Proof by Induction:

MERGE_SORT(A, p, r)

```
1: if  $p < r$  then
2:    $q = \lfloor (p + r) / 2 \rfloor$ 
3:   MERGE_SORT( $A, p, q$ )
4:   MERGE_SORT( $A, q + 1, r$ )
5:   MERGE( $A, p, q, r$ )
```

Assume MergeSort sorts correctly arrays of size $< n$ and show that it sorts correctly an array of size n


- **Base case:** $n=1 \Rightarrow$ the algorithm returns at line 1 with the sorted array of a single element
- **Inductive step:** by inductive assumption lines 3 and 4 return two sub-arrays sorted correctly. We have already proved that **Merge** is correct hence after its execution the algorithm will return the array A sorted

■

➤ MergeSort: Runtime Analysis

- Looking for time $T(n)$: time for MergeSort to sort n elements.
- Assume for simplicity that n is an exact power of 2.

MERGESORT(A, p, r)		
1: if $p < r$ then		$\Theta(1)$
2: $q = \lfloor (p + r) / 2 \rfloor$		$\Theta(1)$
3: MERGESORT(A, p, q)		$T(n/2)$
4: MERGESORT($A, q + 1, r$)		$T(n/2)$
5: MERGE(A, p, q, r)		$\Theta(n)$



Time for
MergeSort
to sort $n/2$
elements.

Yields a **recurrence equation** where $T(n)$ depends on $T(n/2)$:

- If $n=1$, then $p = r$, and the algorithm terminates in constant time $\Theta(1)$


$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 2^0 = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n = 2^k, \text{ for } k \geq 1 \end{cases}$$

- “The time for MergeSort to sort n elements is twice the time for MergeSort to sort $n/2$ elements plus $\Theta(n)$ time (for Merge).”

➤ Recurrence Equation (MergeSort)

- Looking for time $T(n)$: time for MergeSort to sort n elements.
- Assume for simplicity that n is an exact power of 2.

<u>MERGE_SORT(A, p, r)</u>	<u>Time</u>
1: if $p < r$ then	$\Theta(1)$
2: $q = \lfloor (p + r) / 2 \rfloor$	$\Theta(1)$
3: MERGE_SORT(A, p, q)	$T(n/2)$
4: MERGE_SORT($A, q + 1, r$)	$T(n/2)$
5: MERGE(A, p, q, r)	$\Theta(n)$



Time for MergeSort to sort $n/2$ elements.

Yields a **recurrence equation** where $T(n)$ depends on $T(n/2)$:

- If $n=1$, then $p=r$, and the algorithm terminates in constant time $\Theta(1)$
- Otherwise: $T(n) = D(n) + a T(n/b) + C(n)$
 - $D(n)$ - time to *divide* into subproblems: $\Theta(1)$
 - $a T(n/b)$ – time to solve a subproblems each of size n/b : $2 T(n/2)$
 - $C(n)$ – time to *conquer* (to combine the obtained sub-solutions): $\Theta(n)$

➤ How to Solve a Recurrence Equation

$$T(n) = \begin{cases} d & \text{if } n = 2^0 \\ 2T(n/2) + cn & \text{if } n = 2^k, \text{ for } k \geq 1 \end{cases}$$

1. **Substitution method** (Sec 4.3): guess a solution and verify using **induction** (over k).
 - Tutorial exercise.
2. Draw a **recursion tree** (Sec 4.4), add times across the tree.
3. Use the **Master Theorem** (Sec 4.5) to solve a general recurrence equation in the shape of:

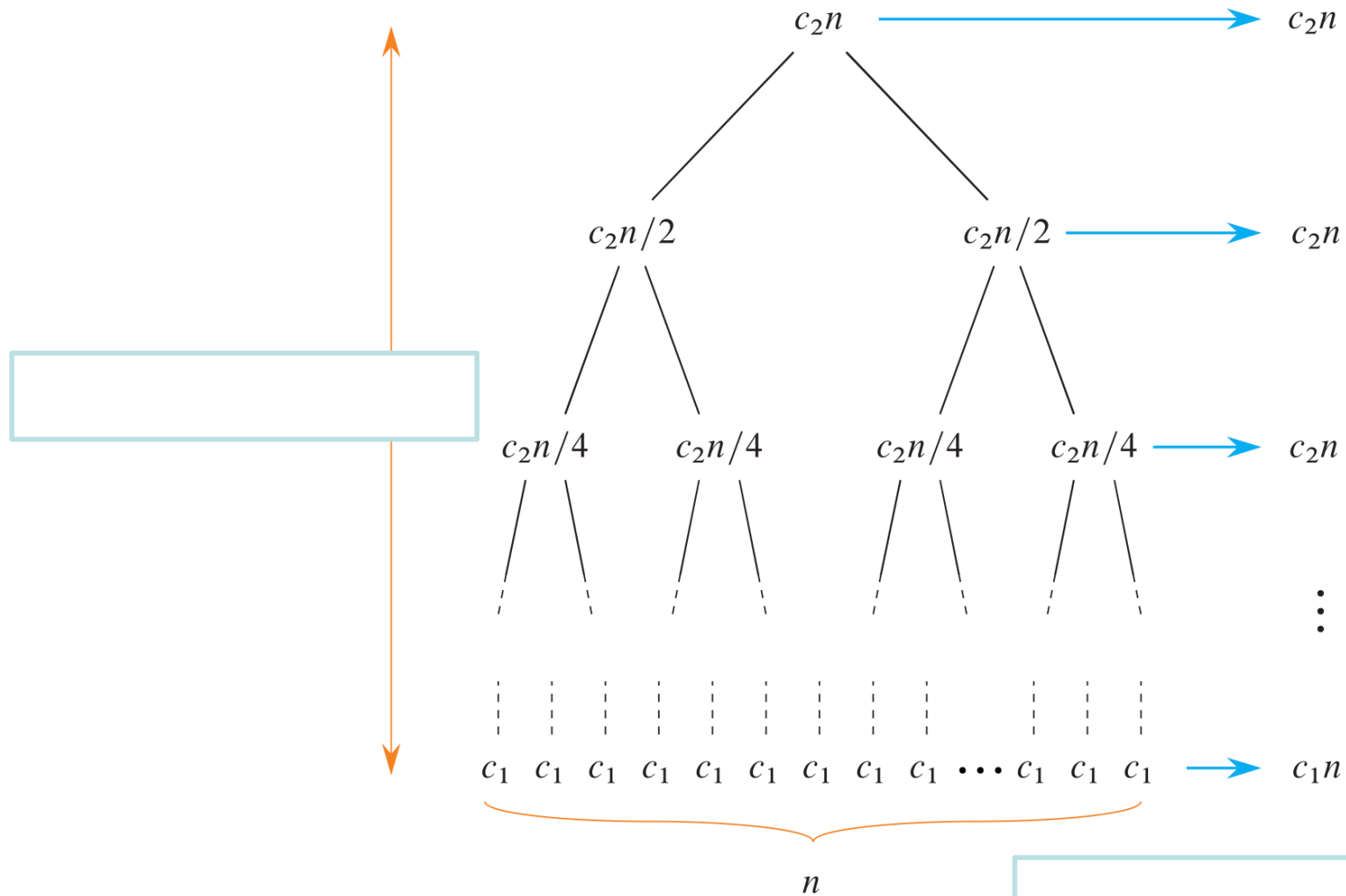
$$T(n) = aT(n/b) + f(n).$$

➤ Runtime Visualised as Recursion Tree

$T(n)$

(a)

➤ Runtime Visualised as Recursion Tree



(d)

➤ Comparison with InsertionSort

- MergeSort **always runs in time** $\Theta(n \log n)$.
- Way better than worst case and average case of $\Theta(n^2)$ for InsertionSort.
- Worse than the best-case time $\Theta(n)$ of InsertionSort.
 - InsertionSort might be faster if your array is almost sorted.
- MergeSort needs **more space** than InsertionSort:
 - MergeSort always stores $\Omega(n)$ elements outside the input.
 - InsertionSort only needs $O(1)$ additional space.
 - We say that InsertionSort sorts **in place**:

A sorting algorithm sorts **in place**
if it only uses $O(1)$ additional space.

➤ The Master Theorem (1)

- Provides a “cookbook” method for solving recurrences of the form $T(n) = aT(n/b) + f(n)$ where $a > 0$ and $b > 1$
- $f(n)$ is called the **driving function** and $T(n)$ is called the **master recurrence**
- The master recurrence $T(n)$ describes the running time of a divide and conquer algorithm that divides a problem of size n into a subproblems each of size $n/b < n$
-> the algorithm solves each subproblem in time $T(n/b)$
- The driving function $f(n)$ describes the cost of dividing the problem before the recursion (**divide**), as well as the cost of combining the results together (**conquer**)

Important term:

- $n^{\log_b a}$ is called the **watershed function**

➤ The Master Theorem (Statement)

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be non-negative for large enough n . Then, the solution of the recurrence function defined over $n \in \mathbb{N}$

$$T(n) = a T(n/b) + f(n)$$

has the following asymptotic behaviour:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = \underline{O(n^{\log_b a - \epsilon})}$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \underline{\Theta(n^{\log_b a} \lg^k n)}$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \underline{\Omega(n^{\log_b a + \epsilon})}$, and if $f(n)$ additionally satisfies the **regularity condition** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

➤ The Master Theorem (Properties)

- Allows you to state the master recurrence $T(n)$ without floors and ceilings even when you don't have problems of exactly the same size

$$\text{eg., } T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \theta(n)$$

- The theorem does not apply to all possible recurrence equations but it does cover the vast majority of those that arise in practice

➤ The Master Theorem: closer look

1. If there exists a constant $\epsilon > 0$ such that $f(n) = \underline{O(n^{\log_b a - \epsilon})}$, then $T(n) = \Theta(n^{\log_b a})$.
 2. If there exists a constant $k \geq 0$ such that $f(n) = \underline{\Theta(n^{\log_b a} \lg^k n)}$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
 3. If there exists a constant $\epsilon > 0$ such that $f(n) = \underline{\Omega(n^{\log_b a + \epsilon})}$, and if $f(n)$ additionally satisfies the **regularity condition** $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■
- $n^{\log_b a}$ is called the **watershed function**
 - **Case 1:** the watershed function must grow **polynomially faster** than $f(n)$ – by at least a factor $\theta(n^\epsilon)$ for some constant $\epsilon > 0$
 - **Case 2:** watershed and driving ($f(n)$) functions grow asymptotically **nearly at the same rate** (you get the same growth for $k = 0$ – common situation)
 - **Case 3:** the watershed function must grow **polynomially slower** than $f(n)$ – by at least a factor $\theta(n^\epsilon)$ for some constant $\epsilon > 0$ + **regularity condition** must hold

➤ The Master Theorem: MergeSort example

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
 2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
 3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the **regularity condition** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■
- MergeSort: $T(n) = 2T(n/2) + \theta(n)$
 - $a=2, b=2, f(n) = \theta(n)$ watershed function: $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
 - Does **Case 1** hold? Does the watershed function grow **polynomially faster** than $f(n)$?
 - Does **Case 3** hold? Does the watershed function grow **polynomially slower** than $f(n)$?

➤ The Master Theorem: MergeSort example

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
 2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
 3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the **regularity condition** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■
- MergeSort: $T(n) = 2T(n/2) + \theta(n)$
 - $a=2, b=2, f(n) = \theta(n)$ watershed function: $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
 - Does **Case 2** hold?

Yes! for $k = 0$, $f(n) = \Theta(n^{\log_b a} \log^0 n) = \Theta(n)$
 - So the solution is $T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n \log n)$

➤ The Master Theorem: Further examples (1)

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
 2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
 3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the **regularity condition** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■
- $T(n) = 9T(n/3) + n$
 - $a=9, b=3, f(n) = n$ watershed function: $n^{\log_b a} = n^{\log_3 9} = n^2$
 - Does **Case 1** hold? Does the watershed function must grow **polynomially faster** than $f(n)$?

Yes! $f(n) = n = O(n^{\log_b a - \epsilon}) = O(n^{2 - \epsilon})$ for any $\epsilon < 1$

- So the solution is $T(n) = \theta(n^{\log_b a}) = \theta(n^2)$

➤ The Master Theorem: Further examples (2)

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
 2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
 3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the **regularity condition** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■
- $T(n) = 3T(n/4) + n \log n$
 - $a=3, b=4, f(n) = n \log n$, watershed function: $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$
 - Does **Case 1** hold? Does the watershed function must grow **polynomially faster** than $f(n)$?

➤ The Master Theorem: Further examples (2)

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
 2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
 3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the **regularity condition** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■
- $T(n) = 3T(n/4) + n \log n$
 - $a=3, b=4, f(n) = n \log n$, watershed function: $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$
 - Does **Case 3** hold? Does the watershed function must grow **polynomially slower** than $f(n)$?
 - Yes! $f(n) = n \log n = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{0.793 + \epsilon})$ for any $0 < \epsilon < 0.207$
 - and $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)(\log n/4) \leq c n \log n$ for $c = 3/4$
 - So the solution is $T(n) = \theta(f(n)) = \theta(n \log n)$
- CSE217: Data Structures & Algorithm Analysis

➤ Summary

- The divide-and-conquer design paradigm
 - **Divides** a problem into smaller subproblems of the same kind
 - **Solves** these subproblems recursively, and then
 - **Combines** these solutions to an overall solution.
- MergeSort uses divide-and-conquer to sort in time $\Theta(n \log n)$ (best case = worst case).
- It's possible to sort n elements in worst-case time $\Theta(n \log n)$!
- Drawback: MergeSort does not sort in place.
 - “In place”: sorting using only $O(1)$ additional space.
- The runtime of recursive algorithms can be analysed by solving a **recurrence equation**.