CSE331 Introduction to Algorithms
Lecture 3: Merge Sort and Binary Search

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 11, 2017

Introduction

- Reference: Section 2.3 of the textbook Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.
- In Lecture 1, we presented Insertion Sort.
- Insertion Sort follows an *incremental* approach:
  - After sorting $A[1 \ldots j-1]$, we insert $A[j]$ at the proper place.
- In this lecture, we present Merge Sort.
- It follows a *Divide-and-Conquer* approach:
  - Sort recursively $A[1 \ldots n/2]$ and $A[n/2+1 \ldots n]$
  - Combine the results.
- Merge Sort is much more efficient than Insertion Sort.

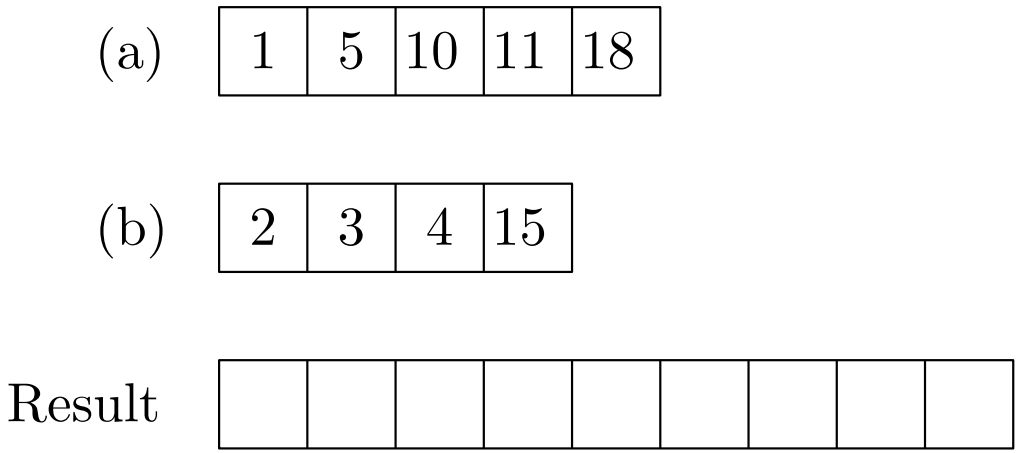
Merging two Sorted Sequences

Problem (Merging two Sorted Sequences)

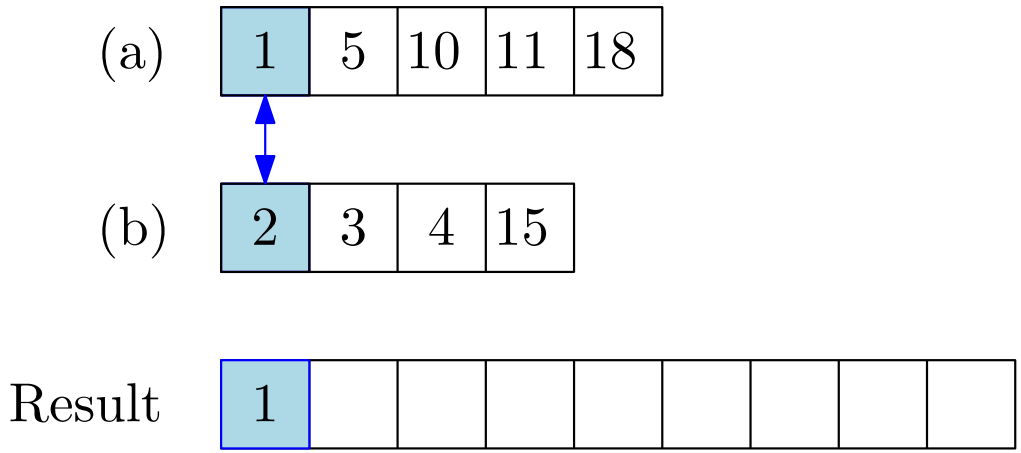
Given two sorted sequences $(a_1, a_2, \ldots, a_{n_1})$ *and* $(b_1, b_2, \ldots, b_{n_2})$*, sort the sequence* $(a_1, a_2, \ldots, a_{n_1}, b_1, b_2, \ldots, b_{n_2})$*.*

- Example:
  - **INPUT:** $(1, 5, 10, 11, 18)$ and $(2, 3, 4, 15)$
  - **OUTPUT:** $(1, 2, 3, 4, 5, 10, 11, 15, 18)$

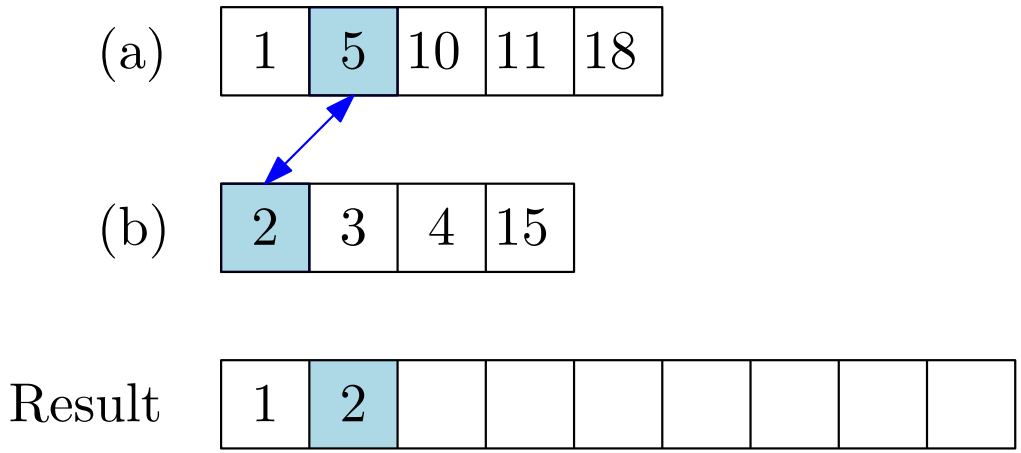
Algorithm for Merging two Sorted Sequences



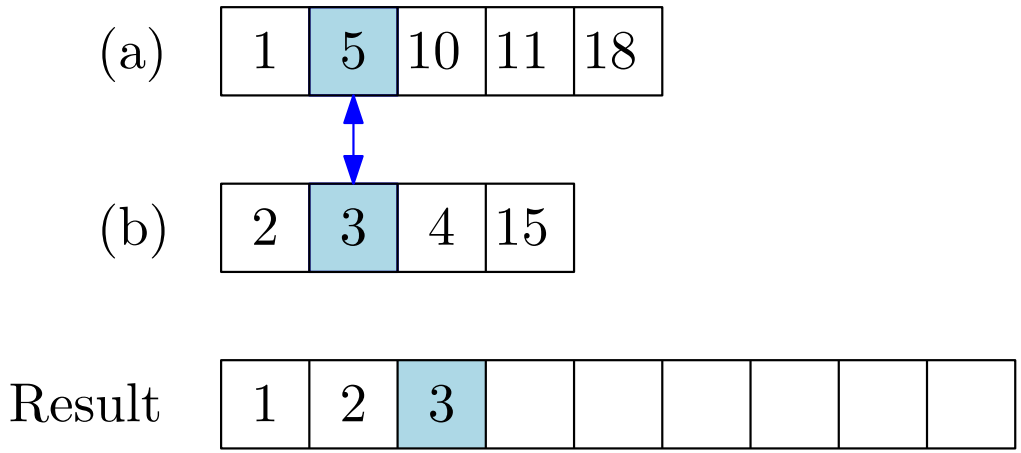
Algorithm for Merging two Sorted Sequences



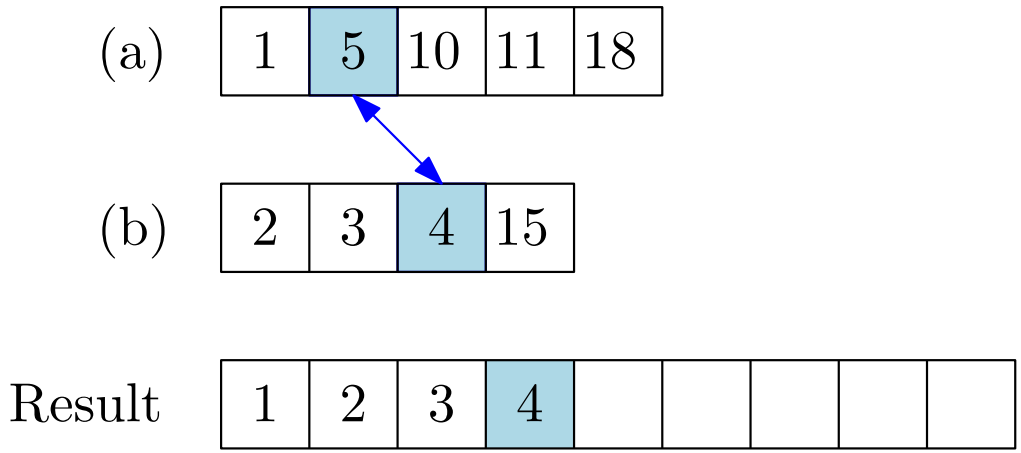
Algorithm for Merging two Sorted Sequences



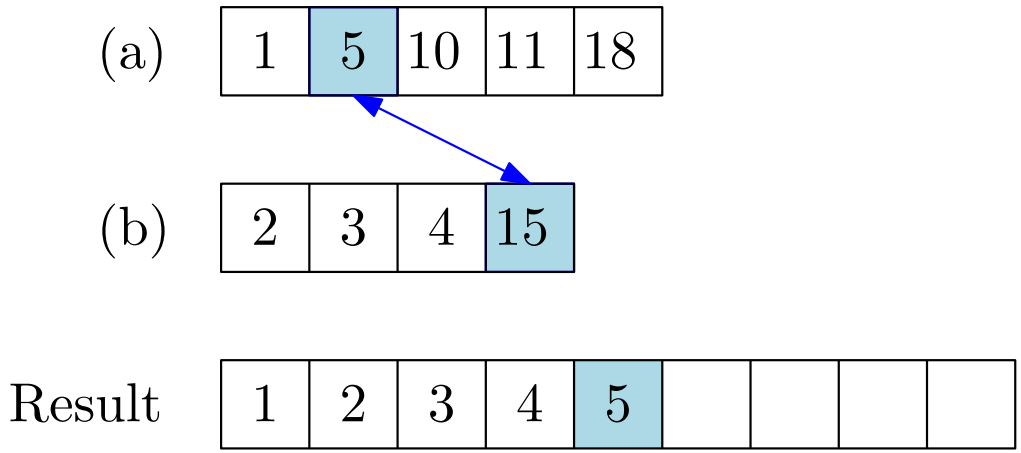
Algorithm for Merging two Sorted Sequences



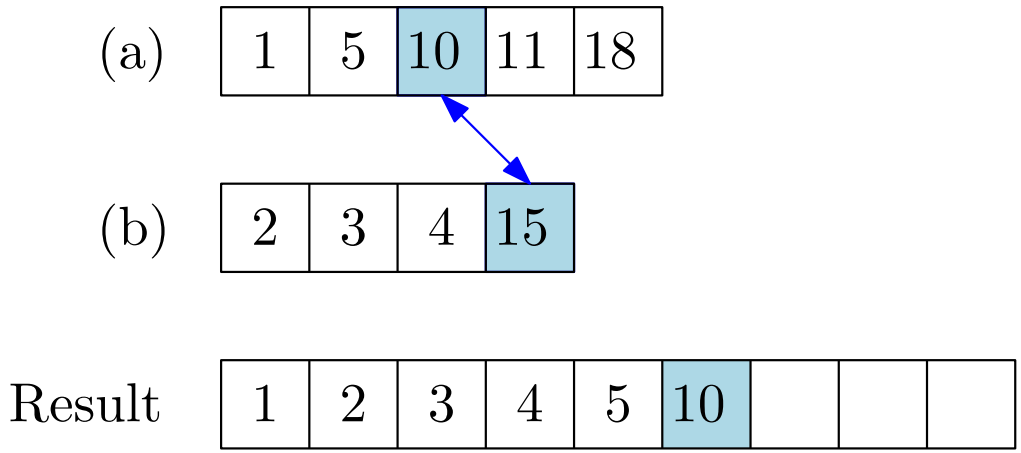
Algorithm for Merging two Sorted Sequences



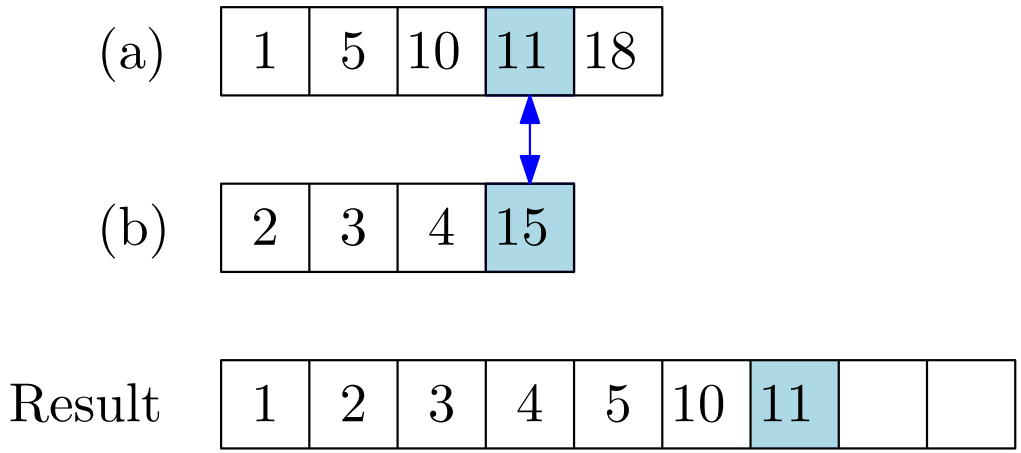
Algorithm for Merging two Sorted Sequences



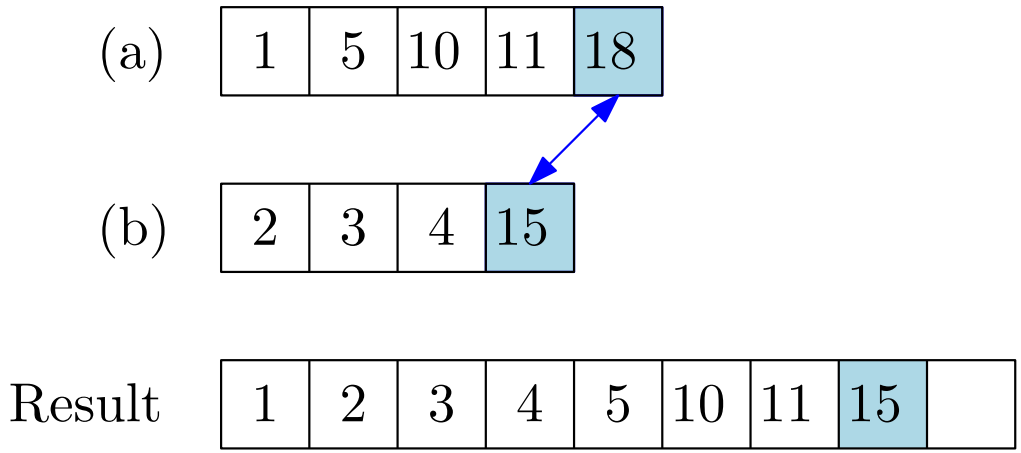
Algorithm for Merging two Sorted Sequences



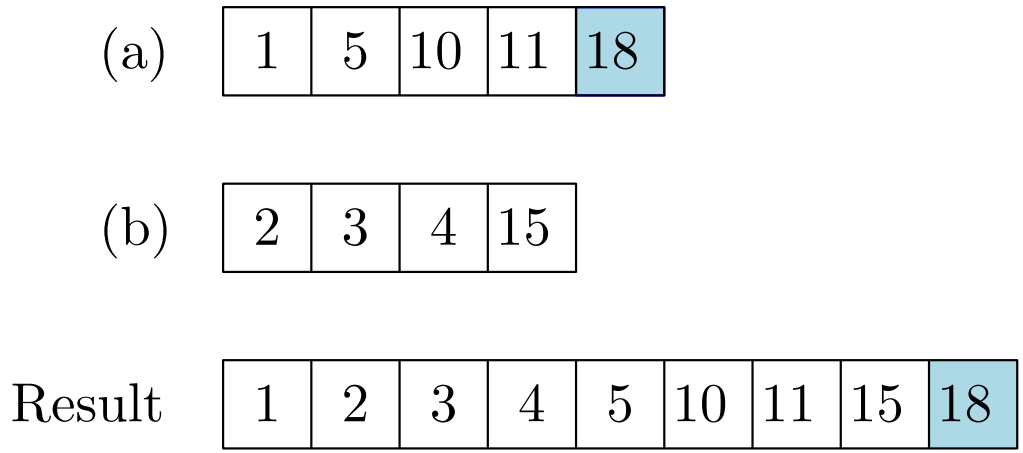
Algorithm for Merging two Sorted Sequences



Algorithm for Merging two Sorted Sequences

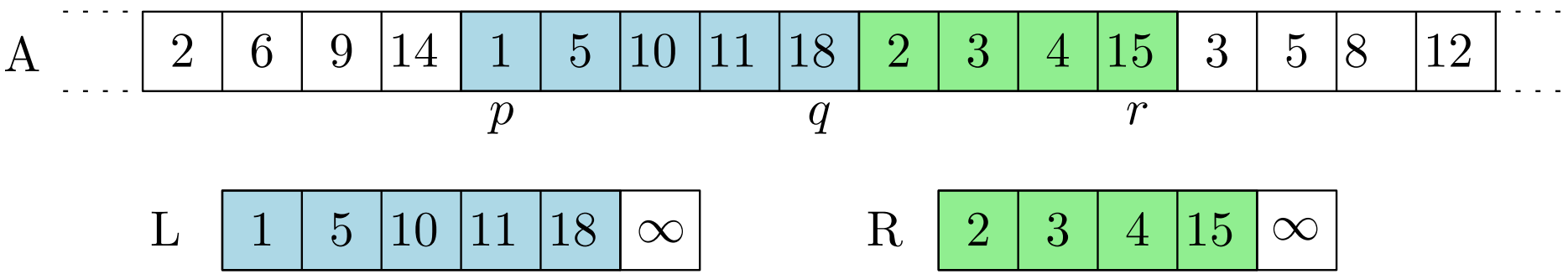


Algorithm for Merging two Sorted Sequences



Algorithm for Merging two Sorted Sequences

- So the algorithm works as follows:
  - Keep a pointer to the current elements a_i and b_j in the input sequences, initialized to a_1 and b_1 respectively.
  - At each step:
    - ★ If $a_i < b_j$, then append a_i to the result, and move the pointer to a_i one position to the right.
    - ★ If $a_i \geqslant b_j$, then append b_j to the result, and move the pointer to b_j one position to the right.
- Next slide:
  - A more detailed pseudocode for the special case where the input sequences are two contiguous subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$.
  - We will place *sentinels*, represented by the value ∞, that we assume to be larger than all the keys.



Pseudocode

1: **procedure** $\textsc{Merge}(A, p, q, r)$
2: $n_1 \leftarrow q - p + 1,\ n_2 \leftarrow r - q$
3: create new arrays $L[1 \ldots n_1 + 1],\ R[1 \ldots n_2 + 1]$
4: **for** $i \leftarrow 1, n_1$ **do**
5: $L[i] \leftarrow A[p + i - 1]$
6: **for** $j \leftarrow 1, n_2$ **do**
7: $R[j] \leftarrow A[q + j]$
8: $L[n_1 + 1] \leftarrow \infty,\ R[n_2 + 1] \leftarrow \infty$
9: $i \leftarrow 1,\ j \leftarrow 1$
10: **for** $k \leftarrow p, r$ **do**
11: **if** $L[i] \leqslant R[j]$ **then**
12: $A[k] \leftarrow L[i]$
13: $i \leftarrow i + 1$
14: **else**
15: $A[k] \leftarrow R[j]$
16: $j \leftarrow j + 1$

Proof of Correctness

Loop Invariant

At the start of each iteration of the for loop of lines 10–16, the subarray $A[p \ldots k-1]$ contains the $k-p$ smallest elements of $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$ in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- Proof of Initialization, Maintenance and Termination done in class. See pages 32–33 of the textbook.

Analysis

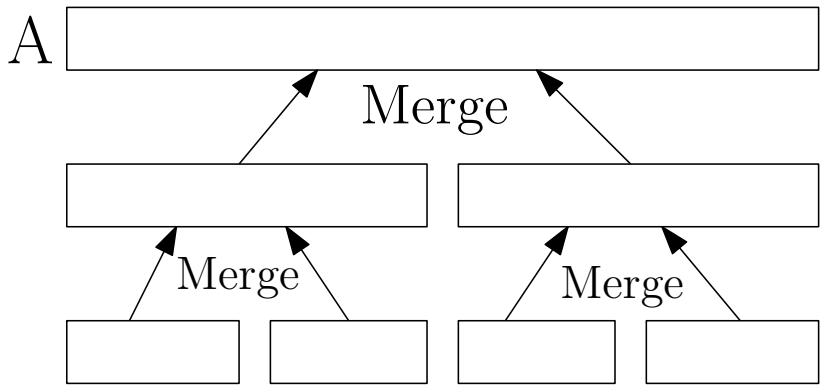
- There are respectively n_1, n_2, and $n_1 + n_2$ iterations in the three loops.
- So the running time is $c_1(n_1 + n_2) + c_2$ for some constants c_1, c_2.
- In terms of the input size $n = r - p + 1$, this is $c_1 n + c_2$. So we just proved:

Theorem

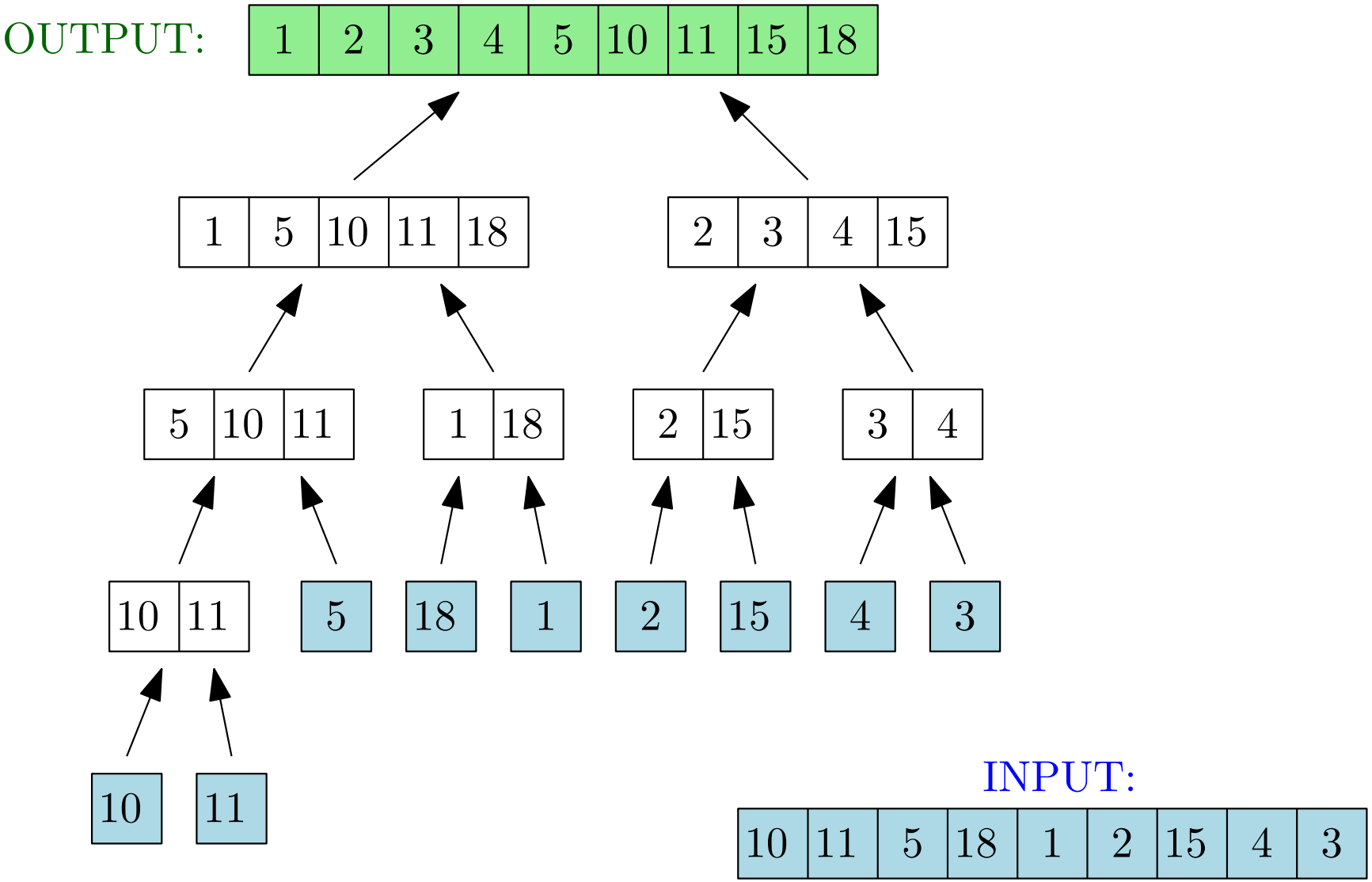
Two sorted sequences can be merged in $c_1 n + c_2$ time, where n is the sum of lengths of the two sequences, and c_1, c_2 are two constants. In other words, two sorted sequences can be merged in linear time.

First Approach

- Insertion Sort runs in $c_4 n^2 + c_5 n + c_6$ for some constants c_4, c_5, c_6.
- First approach:
  - (We assume $n \in 2\mathbb{N}$.)
  - Sort $A[1 \dots n/2]$ and $A[n/2 + 1 \dots n]$ using Insertion Sort.
  - Merge the two results.
- Running time: $\frac{c_4}{2} n^2 + (c_1 + c_5)n + c_2 + 2c_6$
- The leading coefficient has improved by a factor 2.
- For large values of n, it is about twice faster than insertion sort.
- We can push this idea further and split A into 4 parts:
- It improves the leading coefficient further.



Mergesort


OUTPUT:
1 2 3 4 5 10 11 15 18
1 5 10 11 18
2 3 4 15
5 10 11
1 18
2 15
3 4
10 11
5
18
1
2
15
4
3
10
11
INPUT:
10 11 5 18 1 2 15 4 3

Mergesort

- Mergesort splits recursively until the arrays have size 1.
  - No need to call Insertion Sort.

Pseudocode

```
1: procedure MERGE-SORT(A, p, r)
2:     if p < r then
3:         q ← ⌊(p + r)/2⌋
4:         MERGE-SORT(A, p, q)
5:         MERGE-SORT(A, q + 1, r)
6:         MERGE(A, p, q, r)
```

- In order to sort $A[1 \ldots n]$, call MERGE-SORT$(A, 1, n)$

Analysis

- Not counting the call to MERGE and the two recursive calls, MERGE SORT takes constant time c_3.
- So the running time $T(n)$ of MERGE SORT is given by the recurrence relation:

$$T(n) = \begin{cases} c_3 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_1 n + c_2 + c_3 & \text{if } n \geqslant 2 \end{cases}$$

- An upper bound $U(n) \geqslant T(n)$ is given by the relation

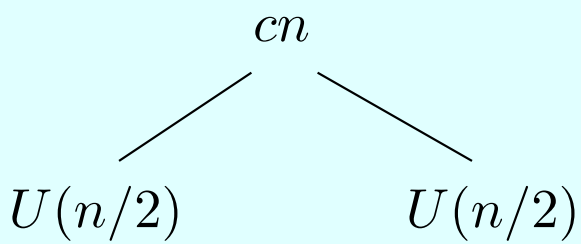
$$U(n) = \begin{cases} c & \text{if } n = 1 \\ U(\lfloor n/2 \rfloor) + U(\lceil n/2 \rceil) + cn & \text{if } n \geqslant 2 \end{cases}$$

  where $c = c_1 + c_2 + c_3$.
- We now show how to solve this recurrence relation using the *recursion tree* method.
- We first assume that n is a power of 2, i.e. $n = 2^h$ for some $h \in \mathbb{N}$.

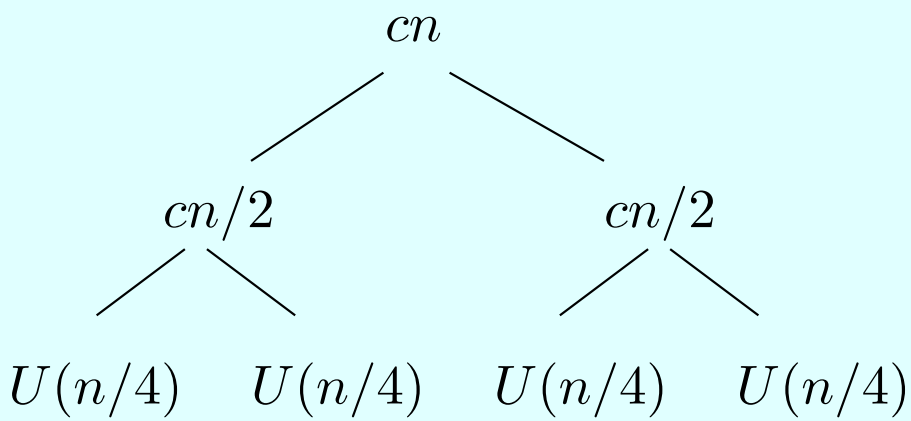
Recursion Tree Method

$$U(n)$$

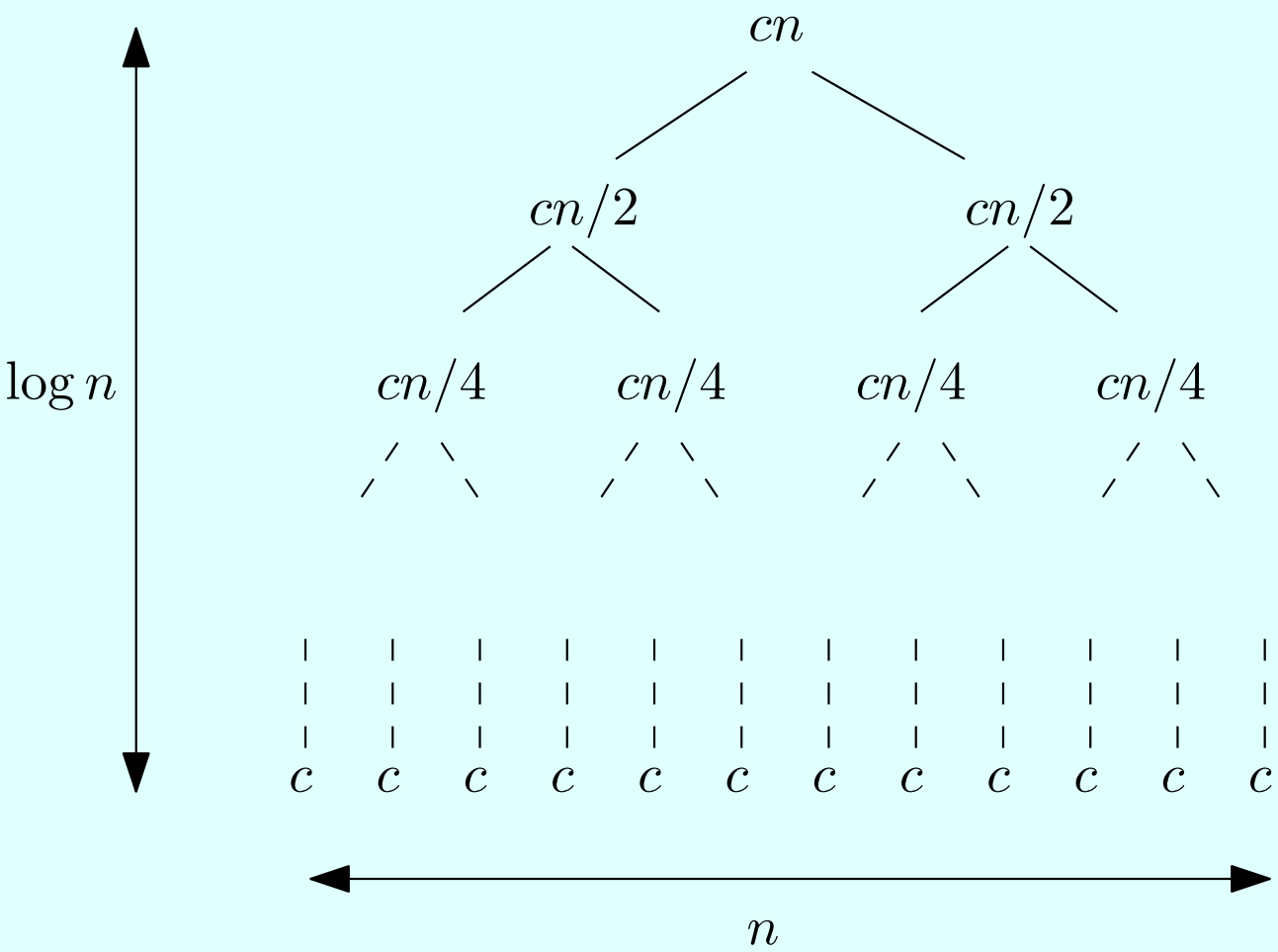
Recursion Tree Method



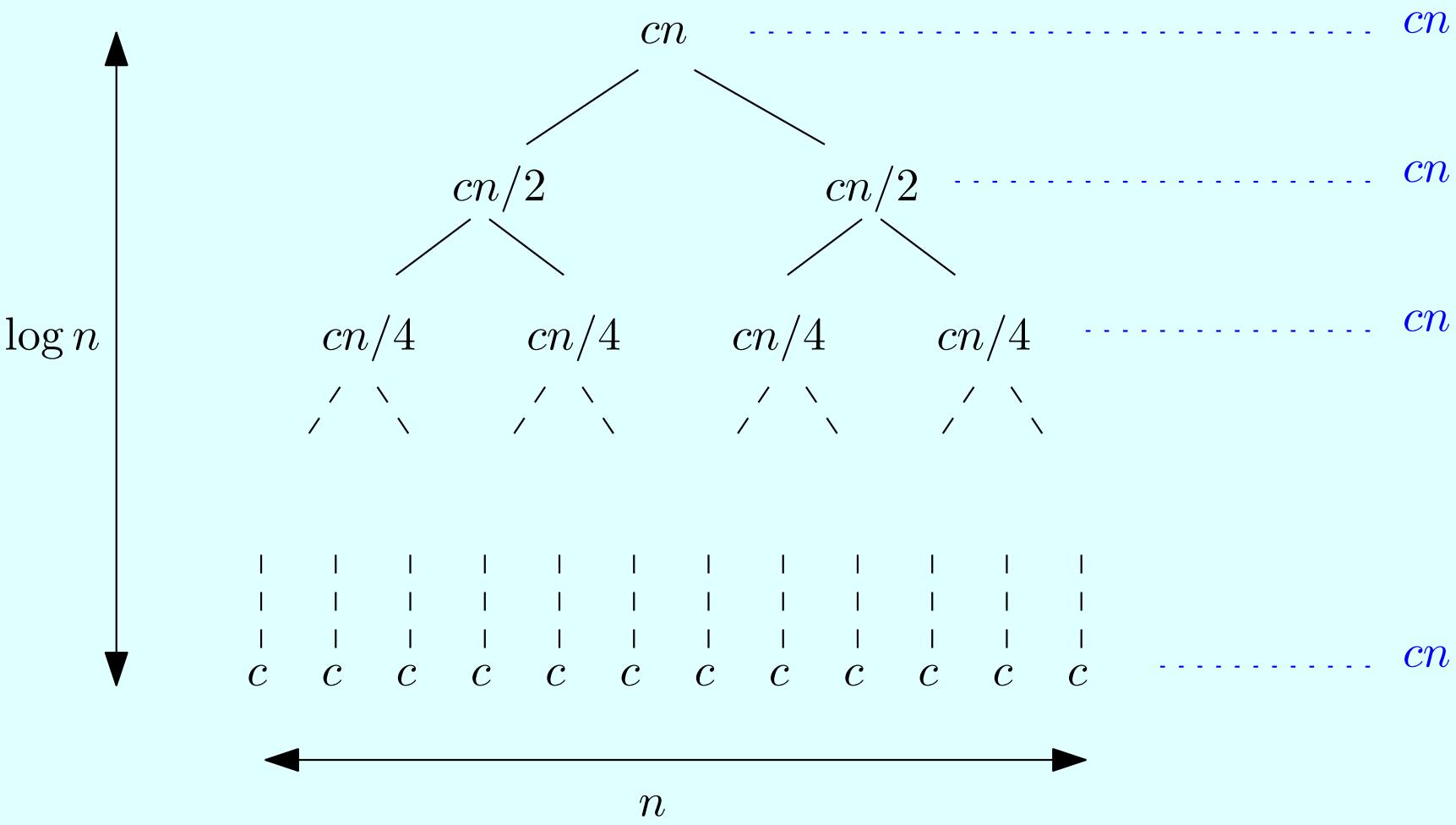
Recursion Tree Method



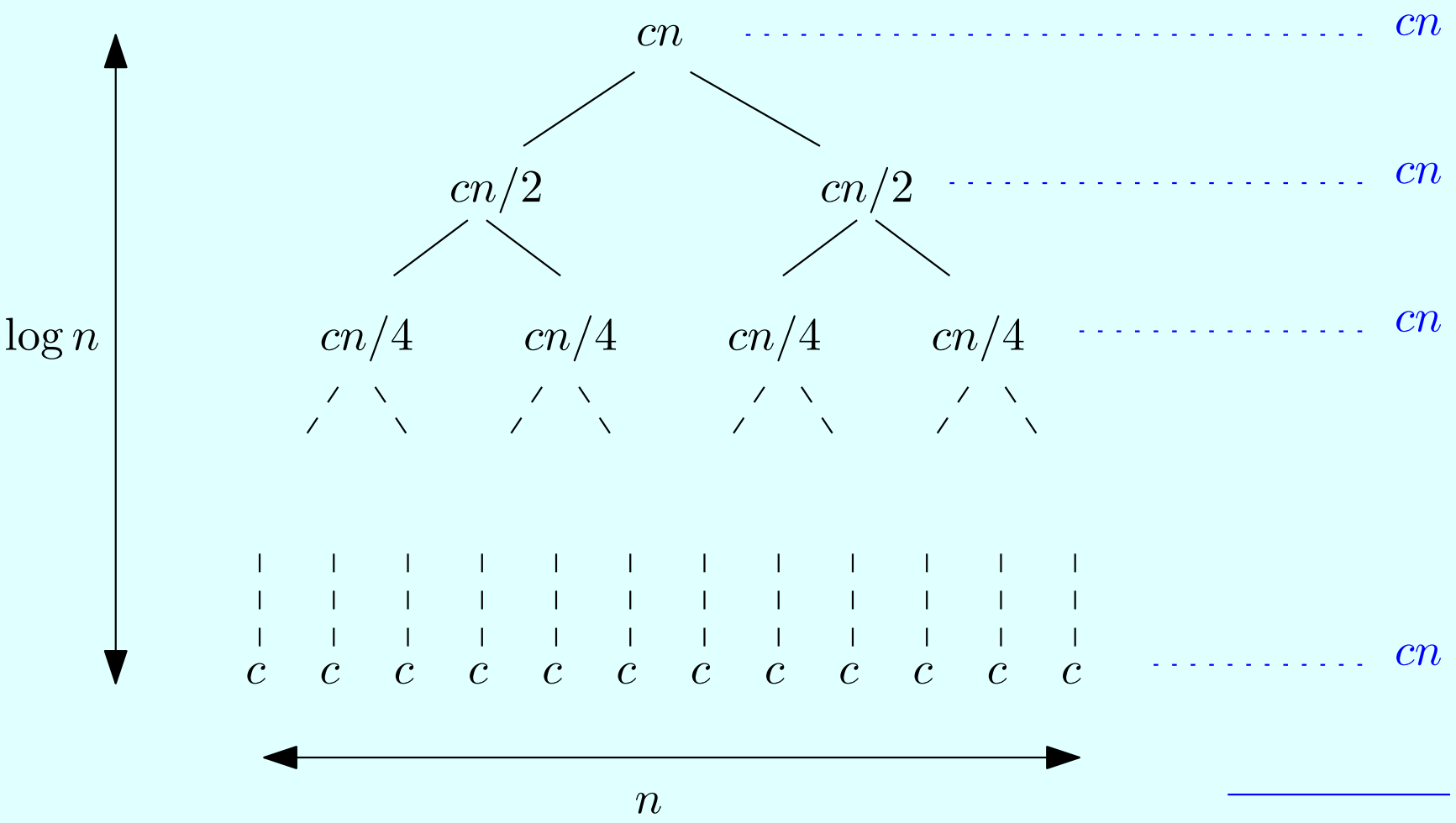
Recursion Tree Method



Recursion Tree Method



Recursion Tree Method



Recursion Tree Method

- Recursion tree method:
  - Expand the recurrence relation into a tree.
  - Add the cost across each level of the tree.
  - Add the costs of all levels.
- Here, the *height* of the tree is $\log n$.
  - In this course, log means $\log_2$
- So there are $1 + \log n$ levels in the tree.
- The cost of each level is cn.
- So $U(n) = cn \log n + cn$.

Technicality

- We only proved $U(n) = cn \log n + cn$ when n is a power of 2.
- Suppose that $2^k < n < 2^{k+1}$.
  - Then the height of the recursion tree is $k + 1$. (Proof?)
  - So $U(n) < cn \log n + 2cn$.

Result

- We have just proved that $T(n) < cn \log n + 2cn$ for some constant c.
- The same approach can be used to show that $T(n) > c'n \log n$ for some constant $c' > 0$.
- It can be summarized as follows.

Theorem

Merge Sort *runs in* $\Theta(n \log n)$ *time.*

- We will argue later this semester that $\Theta(n \log n)$ is the best possible for sorting.

The Divide-and-Conquer Approach

- Merge Sort follows a *divide-and-conquer* approach.
- Divide and Conquer approach:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - **Combine** the solutions to the subproblems into the solution for the original problem.
- Merge Sort follows this paradigm.
  - **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.

The Divide-and-Conquer Approach

- The divide-and-conquer approach is one of the most important algorithm design technique.
- It can often be analyzed using the recursion tree method.
  - We will see other methods later this semester.

Comparison with Insertion Sort

- Insertion Sort runs in $\Theta(n^2)$ time.
  - So for large values of n, MERGE SORT is much faster.
- In the best case, Insertion Sort runs in $\Theta(n)$ time, while MERGE SORT still runs in $\Theta(n \log n)$ time.
  - So Insertion Sort is better in the best case.
  - However, when we analyze algorithms, we usually pay more attention to the worst case running time.
- Insertion Sort is better in terms of memory requirement:
  - The Merge procedure needs to create two auxiliary arrays of linear size.
  - Insertion Sort only stores one key outside the input array.
  - We say that Insertion Sort is *in place*:

Definition

A sorting algorithm is *in place* if it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time.

Comparison with Insertion Sort

- Insertion Sort is *not* a divide and conquer algorithm.
- Insertion sort is an *incremental* algorithm: It inserts the elements one by one, and updates the sorted subarray after each insertion.

Searching in a Sorted Array

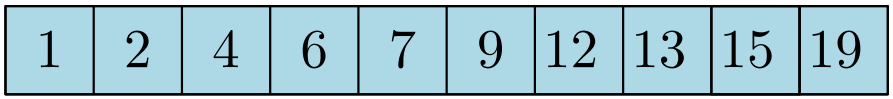
Problem (Searching in a sorted array)

Given a sorted array $A[1 \ldots n]$ of numbers and a query number x, find the position of x in A. In particular, if there exists i such that $x = A[i]$, return i, and otherwise, return NotFound.

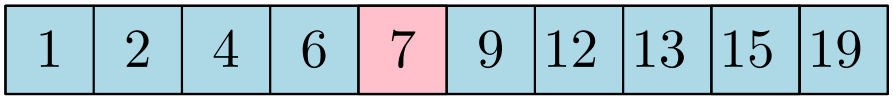
- Can be solved in $\Theta(n)$ by checking whether $x = A[i]$ for all i.
- This is called *brute force search*, because all possible answers are checked.
- It is also called *linear search*.
- Next slides: We introduce *binary search*, which runs in $\Theta(\log n)$ time.

Binary Search

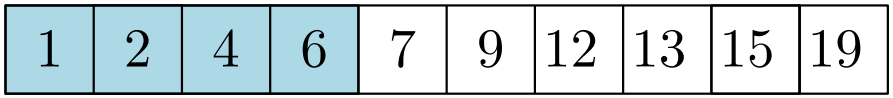
- Example 1: $x = 4$ and $A[1 \ldots 10] = [1, 2, 4, 6, 7, 9, 12, 13, 15, 19]$



- Compare x with the middle element $A[5] = 7$.

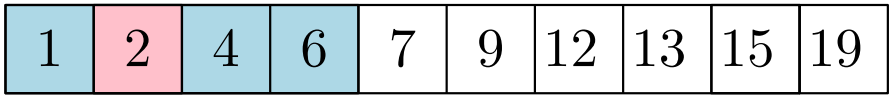


- As $x < A[5]$, recurse on $A[1 \ldots 4]$

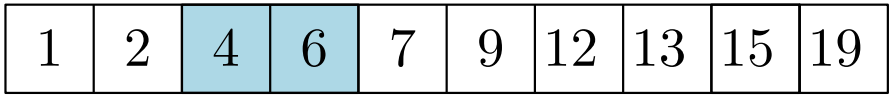


Binary Search

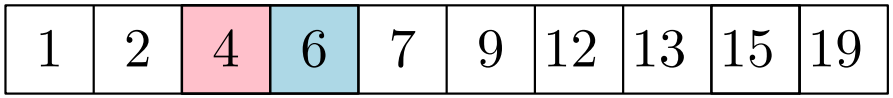
- Compare x with the middle element $A[2] = 2$.



- As $x > A[2]$, recurse on $A[3, 4]$.



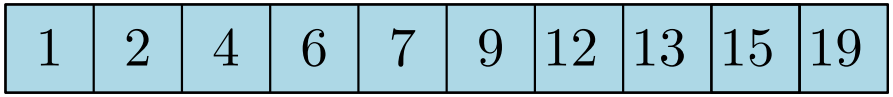
- Compare x with the middle element $A[3] = 4$.



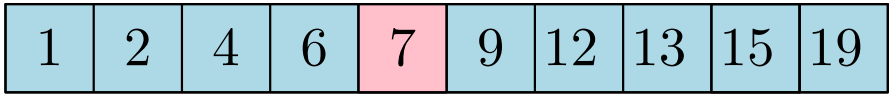
- As $x = A[3] = 4$, we return 3.

Binary Search

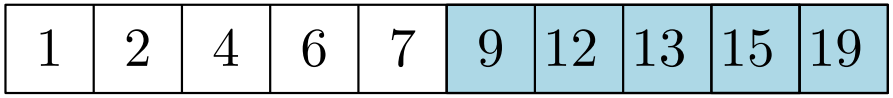
- Example 2: $x = 10$ and $A[1 \ldots 10] = [1, 2, 4, 6, 7, 9, 12, 13, 15, 19]$



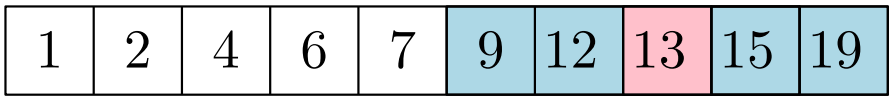
- Compare x with the middle element $A[5] = 7$.



- As $x > A[5]$, recurse on $A[6 \ldots 10]$

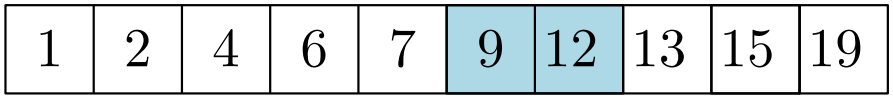


- Compare x with the middle element $A[8] = 13$.

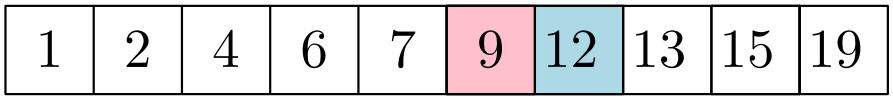


Binary Search

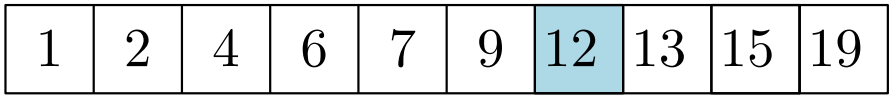
- As $x < A[8]$, recurse on $A[6, 7]$.



- Compare x with the middle element $A[6] = 9$.



- As $x > A[6]$, recurse on $A[7]$.



- As $x \neq A[7]$, return NotFound.

Pseudocode

Pseudocode of Binary Search

```
1:  procedure BinarySearch(A, p, r, x)      ▷ Searches for x in A[p ... r]
2:      if p > r then                        ▷ Test if array is empty
3:          return NotFound
4:      m ← ⌊(p + r)/2⌋
5:      if x = A[m] then
6:          return m
7:      if x < A[m] then
8:          return BinarySearch(A, p, m − 1, x)
9:      else
10:         return BinarySearch(A, m + 1, r, x)
```

Analysis

- The recursion tree is a single path.
- At each level, we spend time c for some constant c.
- Intuition:
  - The recursion tree is similar to a single path to a leaf in the recursion tree of Merge Sort.
  - So there are about $\log n$ levels.
  - So the running time is $\Theta(\log n)$.
- More careful analysis:
  - The size of the current subarray is at most half the size of its parent.
  - So it takes at most $1 + \log\lceil n \rceil$ steps to reach an empty subarray.
  - So the worst-case running time is $\Theta(\log n)$.