Last Week

Asymptotic Notations and Recurrences

- Asymptotic Notations（渐近记号）
  - Big-Oh
  - Big-Omega
  - Big-Theta
  - Algorithm Design and Algorithm Turing
- Solving Recurrences
  - Recursion-tree Method（递归树法）
  - Substitution Method（代入法/替代法）
  - Master Method and Master Theorem (主方法)

Divide & Conquer

- Introduction to Divide & Conquer
- Maximum Contiguous Subarray Problem
  - Problem definition
  - A brute force algorithm
  - A data-reuse algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Pseudo-code

提纲

算法表示

案例分析

撰写工具

提纲

算法表示

案例分析

撰写工具

算法的表示

- 如何表示一个算法?

算法的表示

- 如何表示一个算法?

算法的表示

- 如何表示一个算法?
  - 自然语言

算法的表示

- 自然语言
  - 方法优势
    - 贴近人类思维，易于理解主旨

选择排序

- 第一次遍历找到最小元素
- 第二次在剩余数组中遍历找到次小元素
- ...
- 第n次在剩余数组中遍历找到第n小元素

算法的表示

- 自然语言
  - 方法优势
    - 贴近人类思维，易于理解主旨
  - 不便之处
    - 语言描述繁琐，容易产生歧义
    - 使用了“...”等不严谨的描述

选择排序
- 第一次遍历找到最小元素
- 第二次在剩余数组中遍历找到次小元素
- ...
- 第n次在剩余数组中遍历找到第n小元素

算法的表示

- 如何表示一个算法?
  - 自然语言
  - 编程语言

算法的表示

- 编程语言
  - 方法优势
    - 精准表达逻辑，规避表述歧义

选择排序

```c
void select_sort(int *a, int n) {
    for (int i = 0; i < n - 1; i++) {
        int cur_min = a[i];
        int cur_min_pos = i;
        for (int j = i; j < n; j++) {
            if (cur_min > a[j]) {
                cur_min = a[j];
                cur_min_pos = j;
            }
        }
        int temp = a[i];
        a[i] = a[cur_min_pos];
        a[cur_min_pos] = temp;
    }
    return;
}
```

C语言

```python
def select_sort(a, n):
    for i in range(0, n - 1):
        cur_min = a[i]
        cur_min_pos = i
        for j in range(i, n):
            if cur_min > a[j]:
                cur_min = a[j]
                cur_min_pos = j
        temp = a[i]
        a[i] = a[cur_min_pos]
        a[cur_min_pos] = temp
```

Python语言

算法的表示

- 编程语言
  - 方法优势
    - 精准表达逻辑，规避表述歧义
  - 不便之处
    - 不同编程语言间语法存在差异
    - 过于关注算法实现的细枝末节

选择排序

```c
void select_sort(int *a, int n) {
    for (int i = 0; i < n - 1; i++) {
        int cur_min = a[i];
        int cur_min_pos = i;
        for (int j = i; j < n; j++) {
            if (cur_min > a[j]) {
                cur_min = a[j];
                cur_min_pos = j;
            }
        }
        int temp = a[i];
        a[i] = a[cur_min_pos];
        a[cur_min_pos] = temp;
    }
    return;
}
```

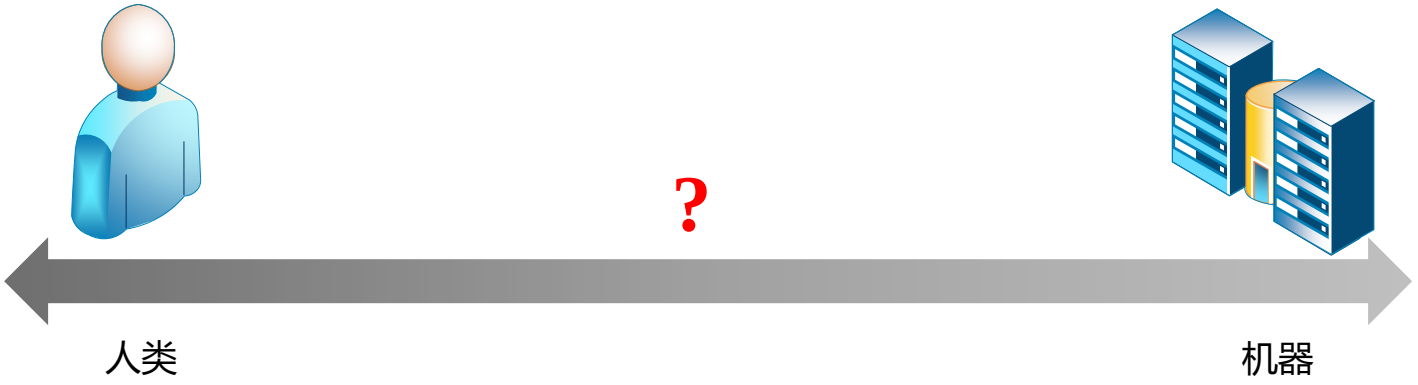
C语言

```python
def select_sort(a, n):
    for i in range(0, n - 1):
        cur_min = a[i]
        cur_min_pos = i
        for j in range(i, n):
            if cur_min > a[j]:
                cur_min = a[j]
                cur_min_pos = j
        temp = a[i]
        a[i] = a[cur_min_pos]
        a[cur_min_pos] = temp
```

Python语言

算法的表示

- 如何表示一个算法?
  - 自然语言：贴近人类思维，易于理解主旨
  - 编程语言：精准表达逻辑，规避表述歧义



问题：可否同时兼顾两类表示方法的优势?

算法的表示

- 如何表示一个算法?
  - 自然语言：贴近人类思维，易于理解主旨
  - 编程语言：精准表达逻辑，规避表述歧义

问题：可否同时兼顾两类表示方法的优势?

算法的表示

- 伪代码
  - 非正式语言
    - 移植编程语言书写形式作为基础和框架
    - 按照接近自然语言的形式表达算法过程

选择排序

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

算法的表示

- 伪代码
  - 非正式语言
    - 移植编程语言书写形式作为基础和框架
    - 按照接近自然语言的形式表达算法过程
  - 兼顾自然语言与编程语言优势
    - 简洁表达算法本质，不拘泥于实现细节
    - 准确反映算法过程，不产生矛盾和歧义

选择排序

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

算法的表示

- 伪代码
  - 书写约定

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

选择排序

算法的表示

- 伪代码
  - 书写约定

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n], 满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

定义算法的输入和输出

选择排序

算法的表示

- 伪代码
  - 书写约定

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

循环
语句块缩进

选择排序

算法的表示

- 伪代码
  - 书写约定

将$i+1$赋值给j

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

选择排序

算法的表示

- 伪代码
  - 书写约定

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n], 满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

注释使用“//”符号

选择排序

算法的表示

- 伪代码
  - 书写约定

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

条件
语句块缩进

选择排序

算法的表示

- 伪代码
  - 书写约定

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

不关注交换过程的实现细节

选择排序

算法的表示

- 伪代码
  - 书写约定

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n], 满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

选择排序

之后出现的算法均使用伪代码描述

算法的表示

- 伪代码
  - 示例解读

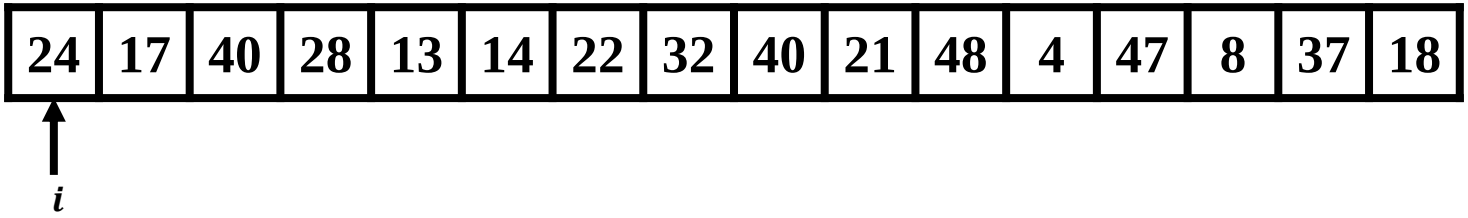
24	17	40	28	13	14	22	32	40	21	48	4	47	8	37	18

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

选择排序

算法的表示

- 伪代码
  - 示例解读



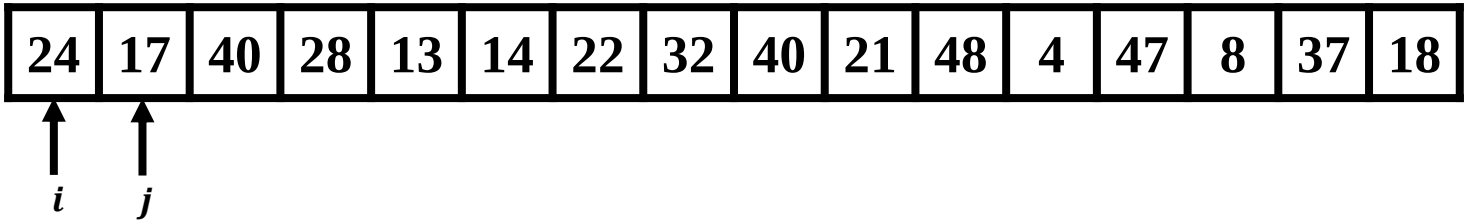

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n], 满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

从数组首部开始枚举i

选择排序

算法的表示

- 伪代码
  - 示例解读



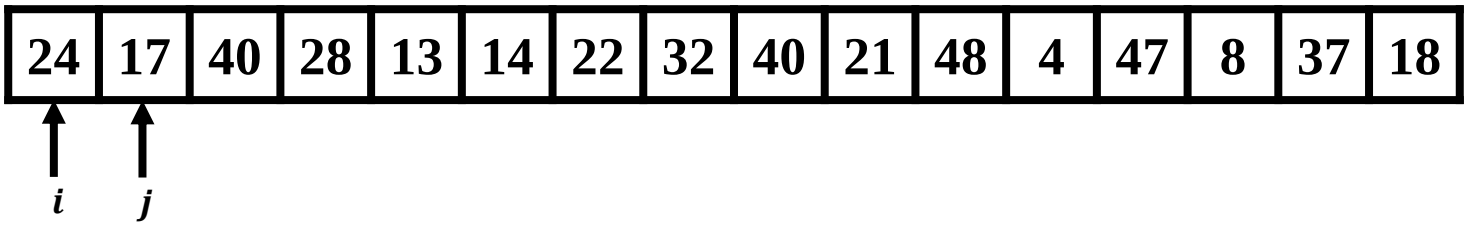

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

从$i+1$开始枚举j

选择排序

算法的表示

- 伪代码
  - 示例解读



cur_min = 17
cur_min_pos = 2

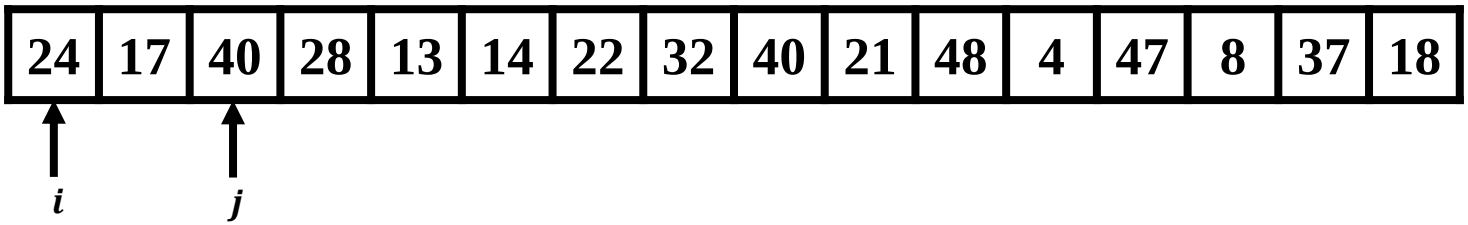
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

不断更新最小值及其位置

选择排序

算法的表示

- 伪代码
  - 示例解读



cur_min = 17
cur_min_pos = 2

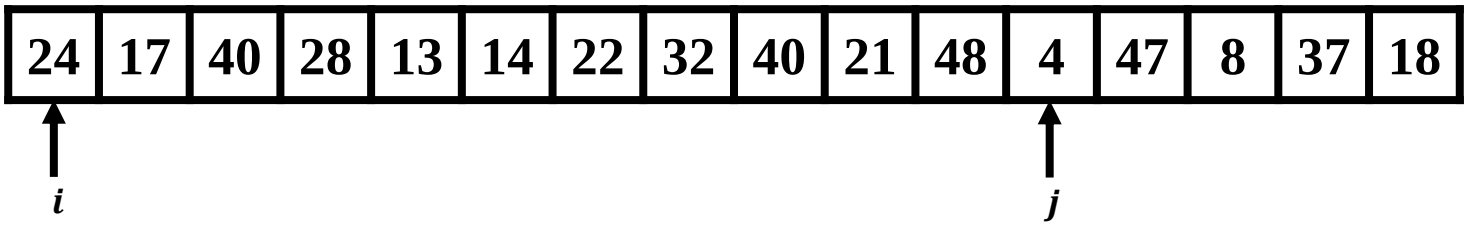
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

不断更新最小值及其位置

选择排序

算法的表示

- 伪代码
  - 示例解读



cur_min = 4
cur_min_pos = 12

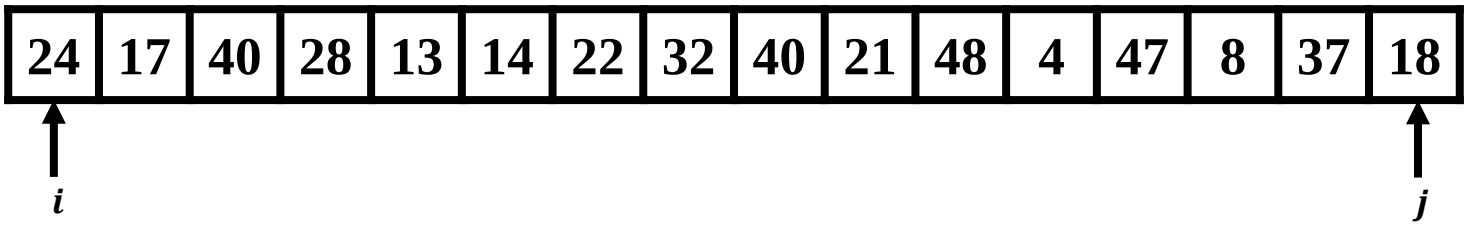
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

不断更新最小值及其位置

选择排序

算法的表示

- 伪代码
  - 示例解读



cur_min = 4
cur_min_pos = 12

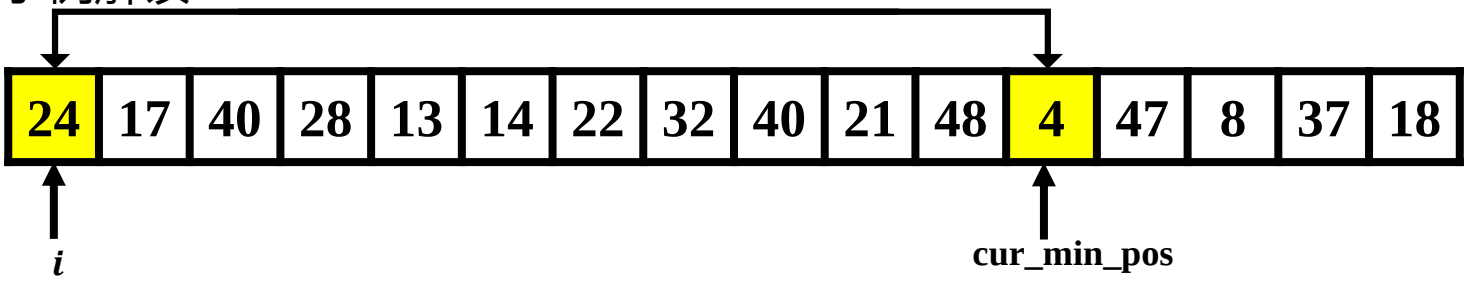
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

不断更新最小值及其位置

选择排序

算法的表示

- 伪代码
  - 示例解读



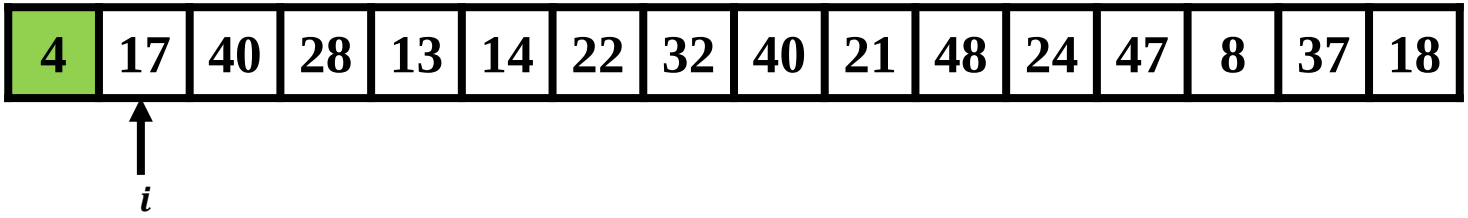
cur_min = 4
cur_min_pos = 12

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

选择排序

算法的表示

- 伪代码
  - 示例解读



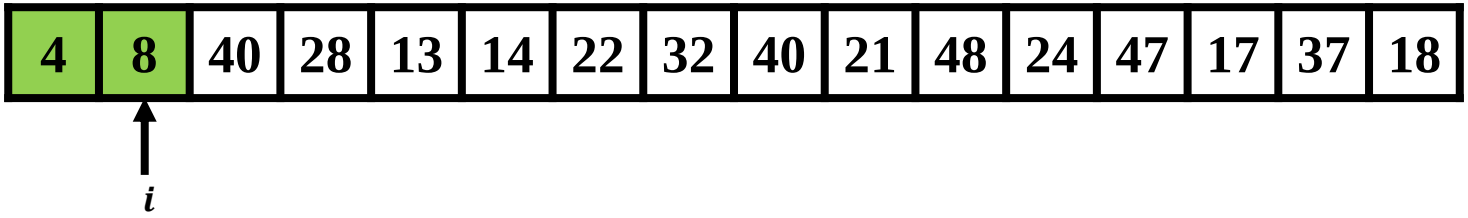

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n], 满足 a'_1 < a'_2 < ... < a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

继续枚举i

选择排序

算法的表示

- 伪代码
  - 示例解读




```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n], 满足 a'_1 < a'_2 < ... < a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

继续枚举*i*

选择排序

算法的表示

- 伪代码
  - 示例解读

……

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n], 满足 a'_1 < a'_2 < ... < a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

枚举i至$n-1$，结束循环

选择排序

算法的表示

- 伪代码
  - 示例解读

4	8	13	14	17	18	21	22	24	28	32	37	40	40	47	48

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for i ← 1 to n − 1 do
    cur_min ← A[i]
    cur_min_pos ← i
    for j ← i + 1 to n do
        if A[j] < cur_min_pos then
            //记录当前最小值及其位置
            cur_min ← A[j]
            cur_min_pos ← j
        end
    end
    交换 A[i] 和 A[cur_min_pos]
end
return A
```

选择排序

算法的表示

- 伪代码
  - 示例解读

17	24	28	40	13	14	22	32	40	21	48	4	47	8	37	18

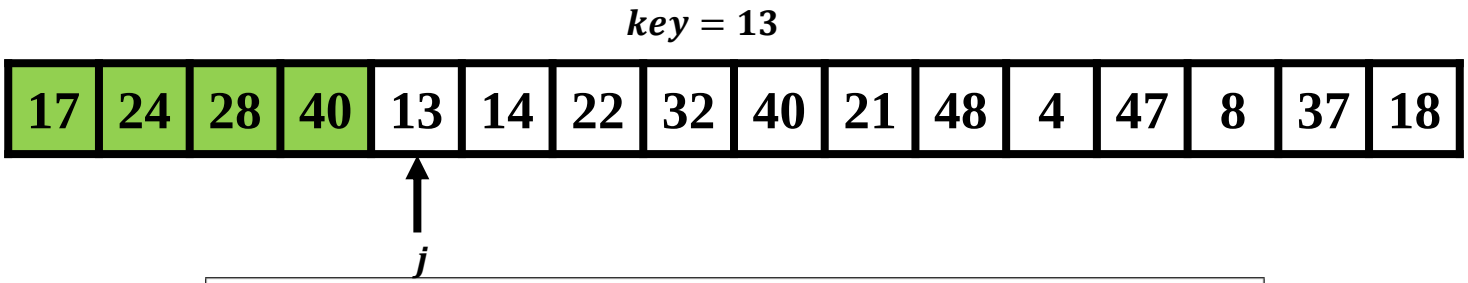
输入: 数组$A[a_1, a_2, ..., a_n]$
输出: 升序数组$A'[a'_1, a'_2, ..., a'_n]$，满足 $a'_1 \leq a'_2 \leq ... \leq a'_n$

```
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1]← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

插入排序

算法的表示

- 伪代码
  - 示例解读



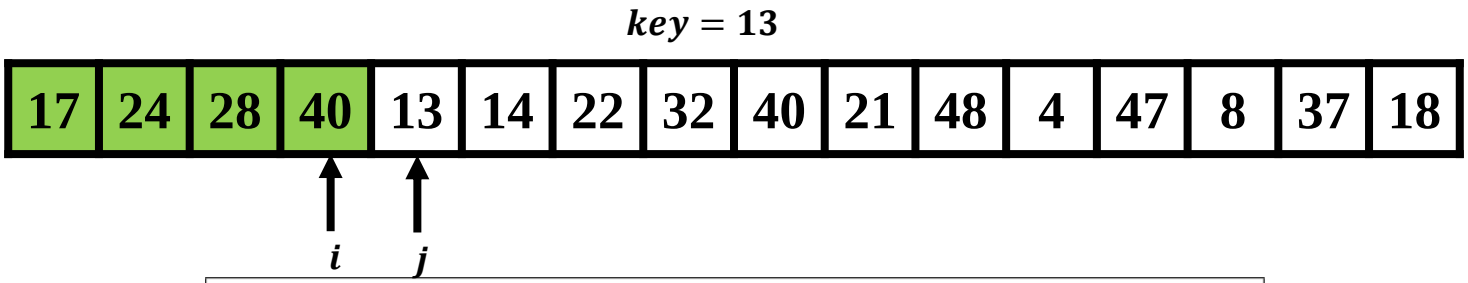

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1]← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将$A[j]$赋值给key

插入排序

算法的表示

- 伪代码
  - 示例解读



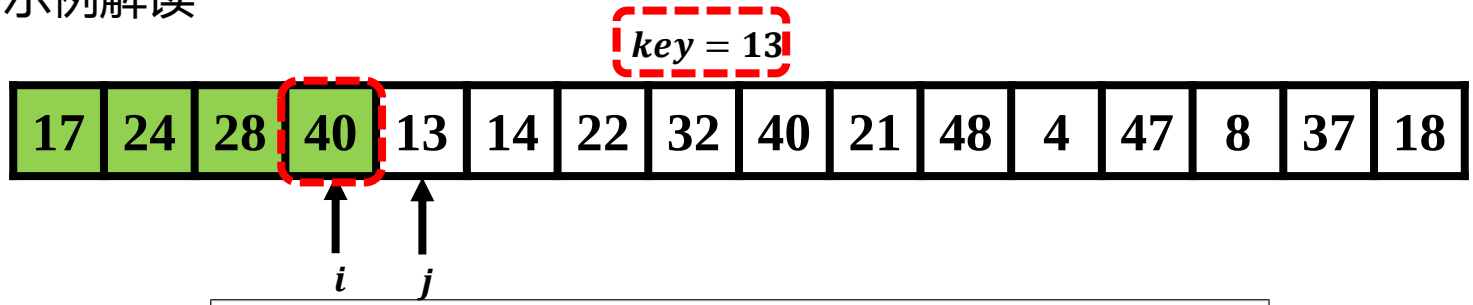

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1]← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

从$j-1$开始枚举i

插入排序

算法的表示

- 伪代码
  - 示例解读



$i > 0$
$A[i] > key$

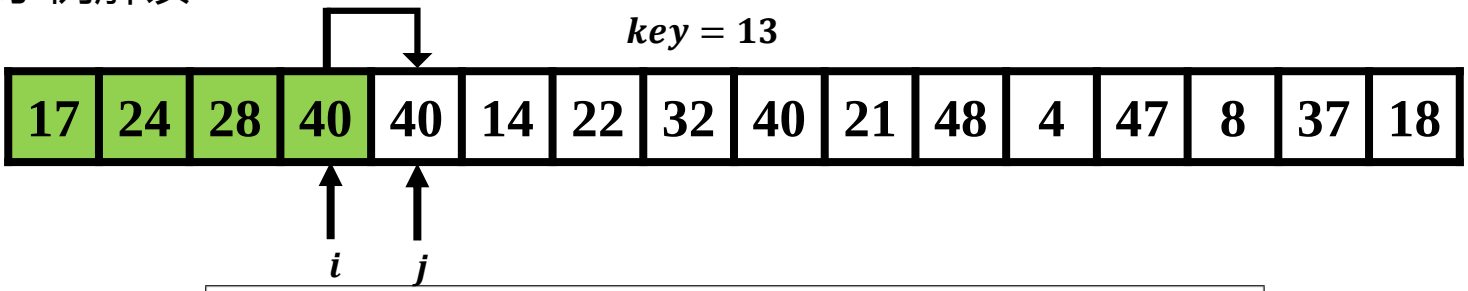
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

判断循环条件

插入排序

算法的表示

- 伪代码
  - 示例解读



输入: 数组$A[a_1, a_2, ..., a_n]$
输出: 升序数组$A'[a'_1, a'_2, ..., a'_n]$，满足 $a'_1 \le a'_2 \le ... \le a'_n$

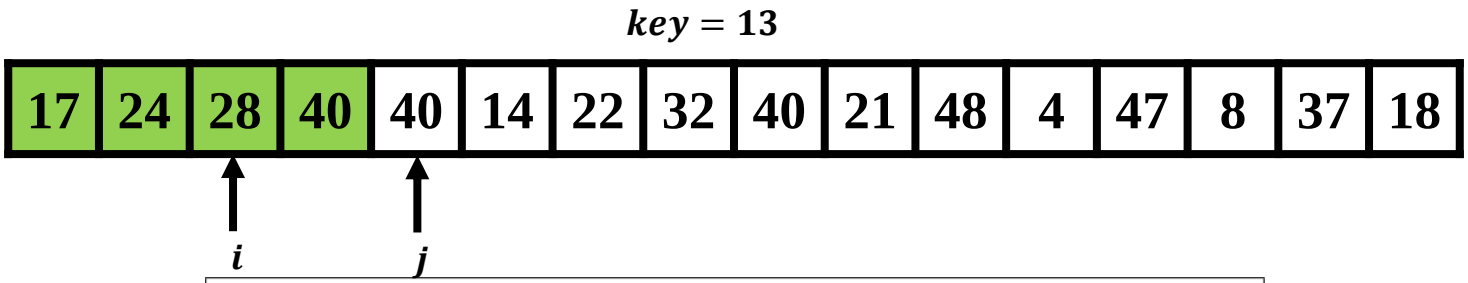
```
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将$A[i]$赋值给$A[i+1]$

插入排序

算法的表示

- 伪代码
  - 示例解读



输入: 数组$A[a_1, a_2, ..., a_n]$
输出: 升序数组$A'[a'_1, a'_2, ..., a'_n]$，满足 $a'_1 \le a'_2 \le ... \le a'_n$

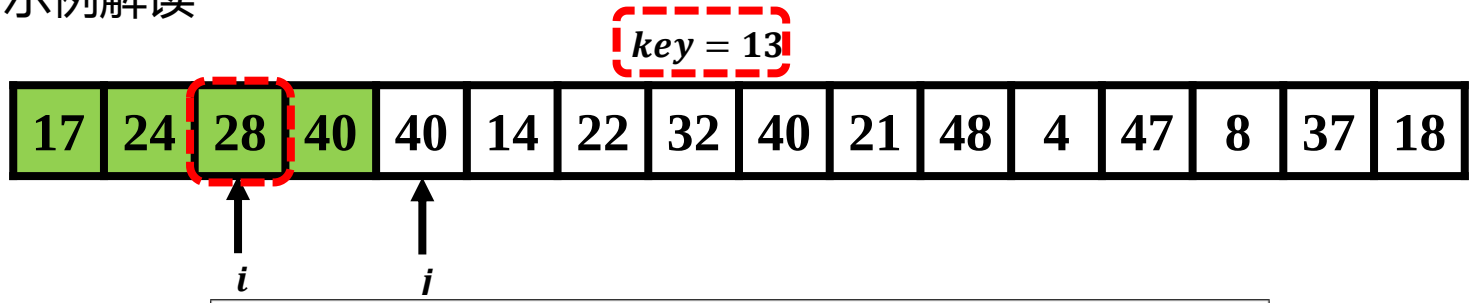
```
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将*i*左移一步

插入排序

算法的表示

- 伪代码
  - 示例解读



$i > 0$
$A[i] > key$

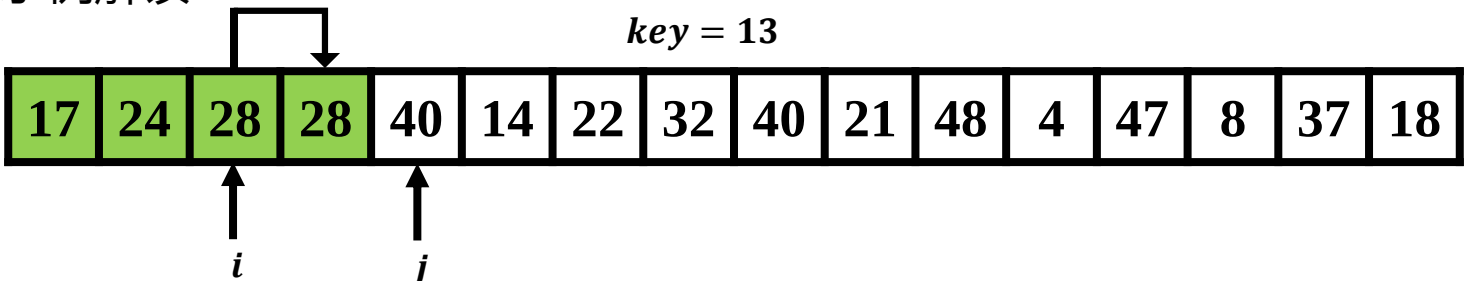
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

判断循环条件

插入排序

算法的表示

- 伪代码
  - 示例解读



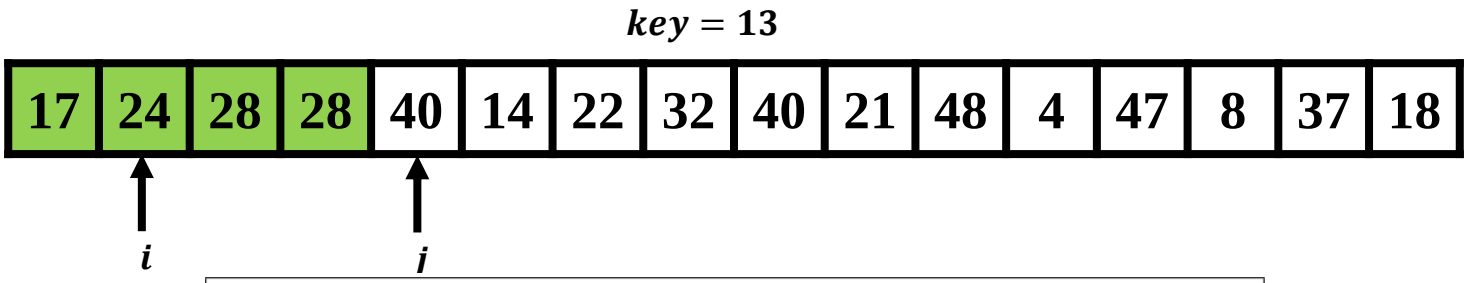

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将$A[i]$赋值给$A[i+1]$

插入排序

算法的表示

- 伪代码
  - 示例解读



输入: 数组$A[a_1, a_2, ..., a_n]$
输出: 升序数组$A'[a'_1, a'_2, ..., a'_n]$，满足 $a'_1 \le a'_2 \le ... \le a'_n$

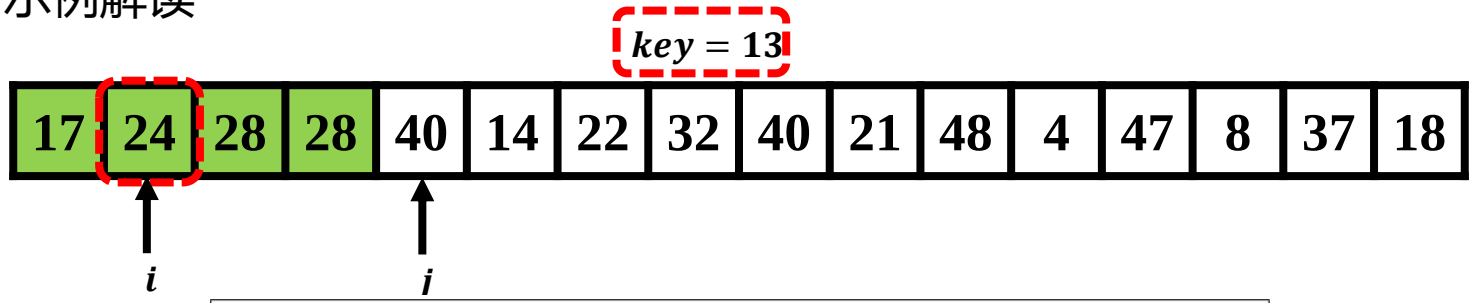
```
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将*i*左移一步

插入排序

算法的表示

- 伪代码
  - 示例解读



$i > 0$
$A[i] > key$

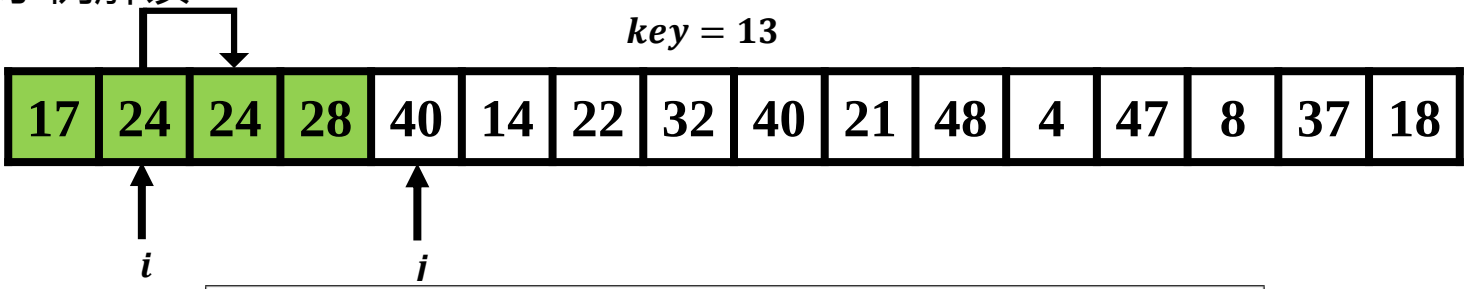
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

判断循环条件

插入排序

算法的表示

- 伪代码
  - 示例解读



输入: 数组$A[a_1, a_2, ..., a_n]$
输出: 升序数组$A'[a'_1, a'_2, ..., a'_n]$，满足 $a'_1 \le a'_2 \le ... \le a'_n$

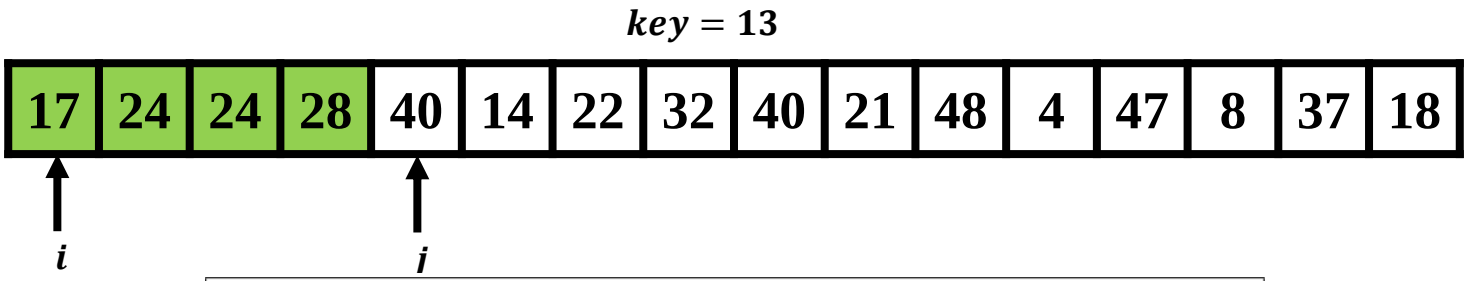
```
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将$A[i]$赋值给$A[i+1]$

插入排序

算法的表示

- 伪代码
  - 示例解读



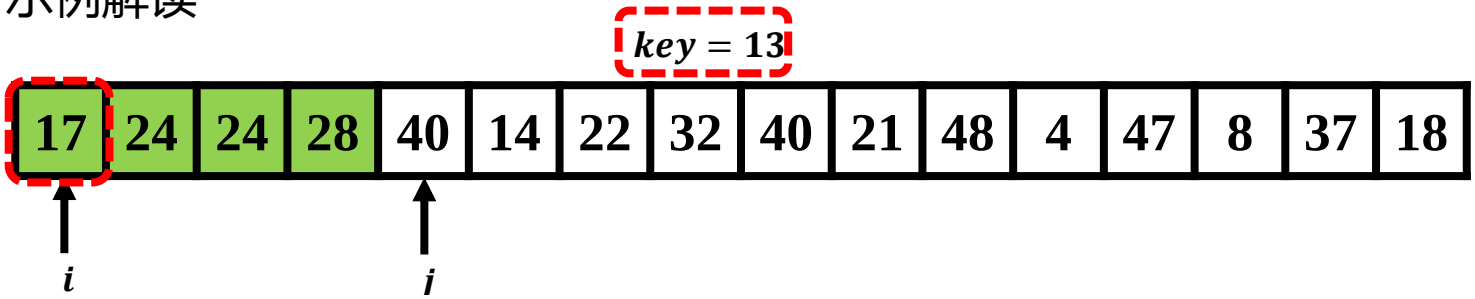

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将*i*左移一步

插入排序

算法的表示

- 伪代码
  - 示例解读



$i > 0$
$A[i] > key$

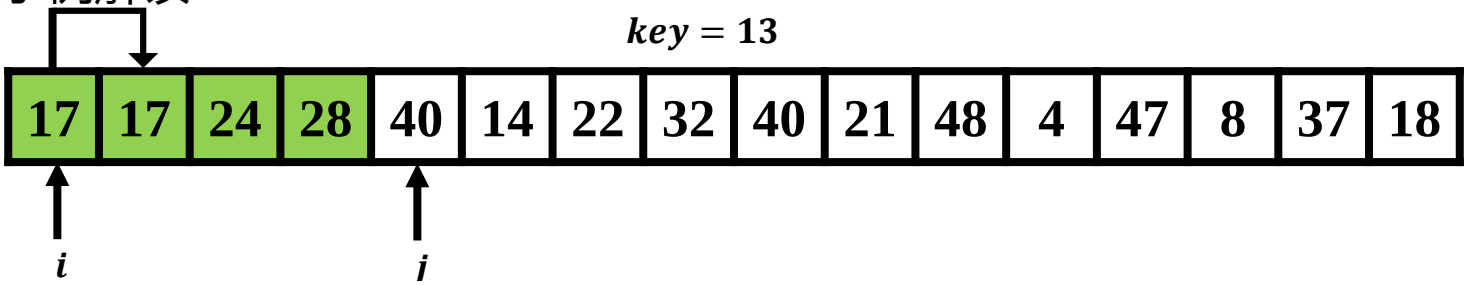
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

判断循环条件

插入排序

算法的表示

- 伪代码
  - 示例解读



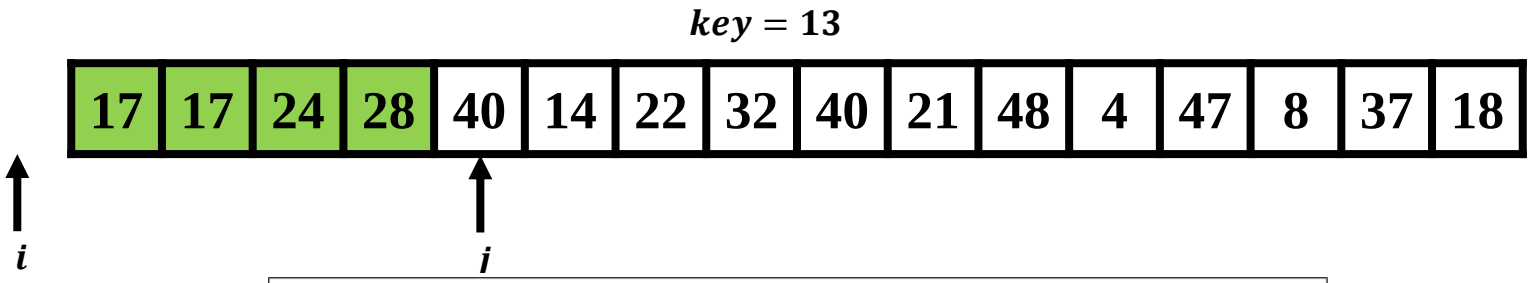

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将$A[i]$赋值给$A[i+1]$

插入排序

算法的表示

- 伪代码
  - 示例解读



输入: 数组$A[a_1, a_2, ..., a_n]$
输出: 升序数组$A'[a'_1, a'_2, ..., a'_n]$，满足 $a'_1 \le a'_2 \le ... \le a'_n$

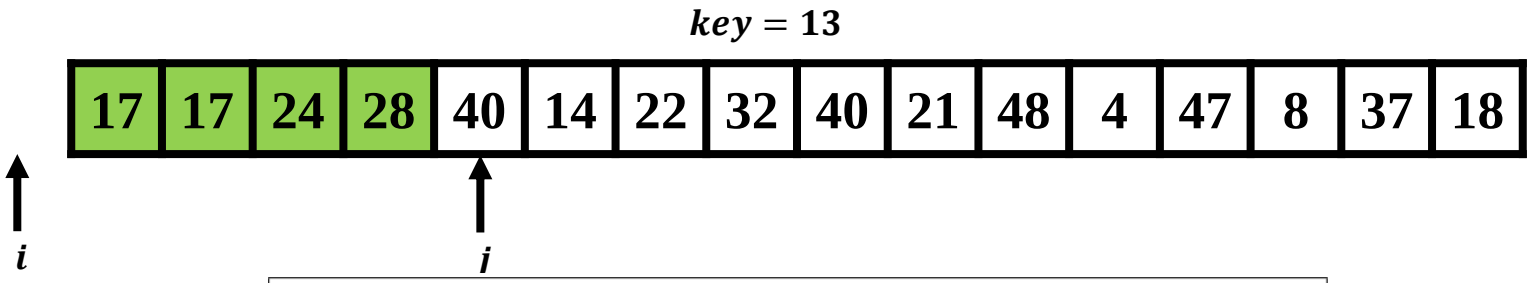
```
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将*i*左移一步

插入排序

算法的表示

- 伪代码
  - 示例解读



$i = 0$
不满足循环条件

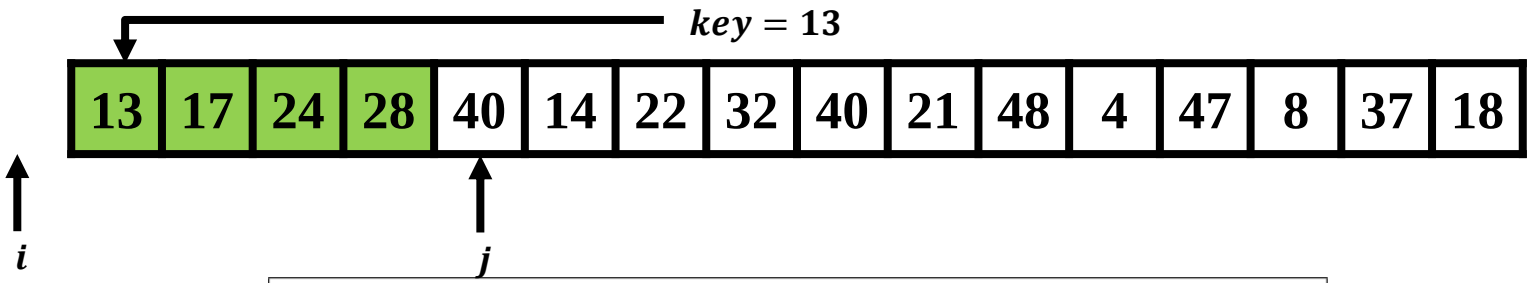
```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

判断循环条件

插入排序

算法的表示

- 伪代码
  - 示例解读



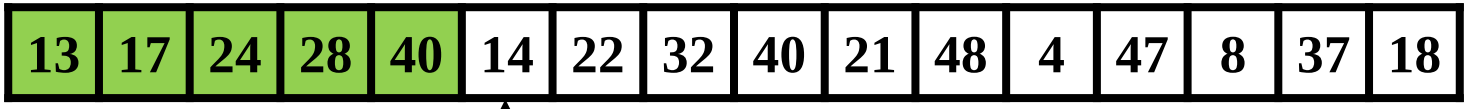

```
输入: 数组A[a_1, a_2, ..., a_n]
输出: 升序数组A'[a'_1, a'_2, ..., a'_n]，满足 a'_1 ≤ a'_2 ≤ ... ≤ a'_n
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1]← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

将**key**赋值给$A[i+1]$

插入排序

算法的表示

- 伪代码
  - 示例解读



j

输入: 数组$A[a_1, a_2, ..., a_n]$
输出: 升序数组$A'[a'_1, a'_2, ..., a'_n]$，满足 $a'_1 \leq a'_2 \leq ... \leq a'_n$

```
for j ← 2 to n do
    key ← A[j]
    //将 A[j] 插入到已排序的数组A[1..j − 1]中
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1]← A[i]
        i ← i − 1
    end
    A[i + 1] ← key
end
return A
```

插入排序

算法的表示方式比较

表示方式	语言特点
自然语言	贴近人类思维，易于理解主旨 表述不够精准，存在模糊歧义
编程语言	精准表达逻辑，规避表述歧义 受限语法细节，增大理解难度
伪代码	关注算法本质，便于书写阅读

提纲

算法表示

案例分析

撰写工具

优质案例

2 k 重有序问题

2.1 Main Idea

记每段长度为 $l = n/k$。对长度为 tl 的问题而言，首先将其调整成 2 重有序数组，方法如下：

1. 寻找中位数：找到当前长度为 tl 的数组的第 $tl/2$ 小数，记其值为 m。
2. Partition：扫描数组，找到任意一个 m。以它为 pivot，进行 QuickSort 中的 Partition 操作。操作后，左段小于等于 m，右段大于 m。但该值为 m 的 pivot 并不一定在第 $tl/2$ 位置。
3. 调整中位数：扫描左段，将所有值等于 m 的数调整到左段的末尾。由于右段大于 m，因此右段没有等于 m 的数。

此时，左段的末尾全部为值为 m 的数，且左段小于等于 m，右段大于 m。则必有值为 m 的数在第 $tl/2$ 位置，且左段小于等于它，右段大于它。则该数组 2 重有序。之后对每个无序数组先调整成 2 重有序数组，再递归划分长度均等的子数组，对每个子数组重复上述步骤直到当前数组长度为 l，则不进行操作，返回。

2.2 Pseudo Code

Algorithm 1: *DoubleOrder*$(A[1...L], l)$

Input: Array $A[1...L]$ whose length is L
Output: The Rearranged 2-Ordered Array $A[1...L]$
if $L = l$ **then** return;
$median \leftarrow$ `FindKth`$(A[1...L],\ L/2)$;
$pivot \leftarrow 1$;
for $i \leftarrow 1$ **to** L **do**
  if $A[i] = median$ **then** $pivot \leftarrow i$;
end
swap $A[pivot]$ and $A[L]$;
$pivotPosition =$`PartitionWithRightPivot`$(A[1...L])$;
$lowerMedian \leftarrow pivotPosition$;
for $i \leftarrow pivotPosition - 1$ **to** 1 **do**
  if $A[i] = median$ **then**
    $lowerMedian \leftarrow lowerMedian - 1$;
    swap $A[lowerMedian]$ and $A[i]$;
  end
end
DoubleOrder$(A[1...L/2], l)$;
DoubleOrder$(A[L/2 + 1...L], l)$;

已知中位数的基于 Partition 的重整 2 重有序算法：
已知中位数后，进行以中位数为 pivot 的 Partition 操作，算法的期望复杂度和最坏复杂度均为 $O(n)$。
之后进行扫描调整中位数到左段的末尾，复杂度也为 $O(n)$。
因此该阶段的期望复杂度和最坏复杂度均为 $O(n)$。

因此，对于每个长为 L 的数组，将其重排成 2 重有序数组的期望时间复杂度为 $O(L)$。

总时间复杂度：
设每段长度为常数 $l = n/k$。则某个段数为 t 的问题的总运行时间 $T(t, l)$ 有：

$$T(t,l) = \begin{cases} 1 & t = 1 \\ 2T(t/2, l) + O(tl) & t = 2^b, b = 1, 2, ... \end{cases}$$

解得 $T(t, l) = O(tl \log t)$。现在将初始问题的记号 $l = n/k, t = k$ 代回该式，得到：

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

1. 如果$k = 1$，那么$A[1..n]$即是所求
2. 否则先对序列进行 KTH-ELEMENT$(A[1..n], n/2)$，然后分别进行 ROUGHLY-SORT$(A[1..n/2], k/2)$，以及 ROUGHLY-SORT$(A[n/2 + 1..n], k/2)$

伪代码

算法 1 将序列排成k重有序
输入: A数组，k有序数组的重数
输出: 排序后的数组A

function KthElement(A, k)
  randomly choose a number $p \in [1, n]$
  call Partition(A, p), and set the new location of $A[p]$ to p'
  if $p' \geq k$ **then**
    KthElement$(A[1..p'], k)$
  else
    KthElement$(A[p' + 1..n], k - p')$
  end if
end function
function RoughlySort$(A[1..n], k)$
  if $k = 1$ **then**
    return
  else
    call KthElement$(A[1..n], n/2)$
    call RoughlySort$(A[1..n/2], k/2)$
    call RoughlySort$(A[n/2 + 1..1], k/2)$
  end if
end function

复杂度分析

其次分析ROUGHLY-SORT的时间复杂度$T(n, k)$，可以发现递归式：

$$T(n,k) = \begin{cases} 1 & , k = 1 \\ 2T(n/2, k/2) + f(n) & , \text{else} \end{cases}$$

其中$f(n) = O(n)$。所以，可以得出，$T(n, k) = O(n \log k)$。

优质案例

name: **getK**

input

array A: 数组 A

begin: 研究范围的起点

end: 研究范围的终点

output： k

伪代码：

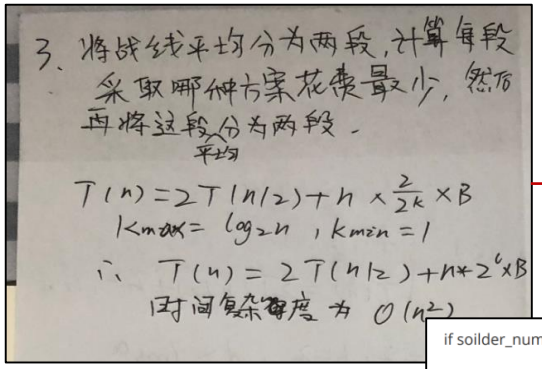
```
//递归情况
if end – begin >1:
    mid = (begin+end)/2
    //左右分治，
    if a[mid] > a[begin]:
        return getK(A, mid, end)
    else:
        return getK(A, begin, mid)
//基本情况
else:
    if A[begin]>A[end]：
        return end
    else：
        return 0
```

设计算法，考虑分治，对于一个区间，折半去判断是否要均分为左右两段按规则计算。先需要准备一个堡垒中士兵总数前缀和 Pre。复杂度为 $T(2^l) = 2T(2^{l-1}) + O(1)$，即 $T(2^l) = O(2^l)$ 预处理的复杂度也是 $O(2^l)$ 故总复杂度为 $O(2^l)$

Algorithm 1 calculate the supply value for [L,R]

```
1:  function CALC(L, R)
2:      mid = (L+R)/2
3:      num ← Pre[R] − Pre[L − 1]
4:      if num = 0 then
5:          now = A
6:      else
7:          now = num × (R − L + 1) × B
8:      end if
9:      if L == R then
10:         result = now
11:     else
12:         result = min( CALC(L, mid) + CALC(mid + 1, R) , now)
13:     end if
14:     return result
15: end function
```

低分案例



文字描述非常笼统，没有说明时间复杂度是怎么得出的，而且计算的结果也是错误的。

3.战线补给问题

已知士兵有 n 个，堡垒有 2l 个。设

情况 1：直接为整条战线[1,2l]提供补给

cost = n * 2l * B

情况 2：均分补给

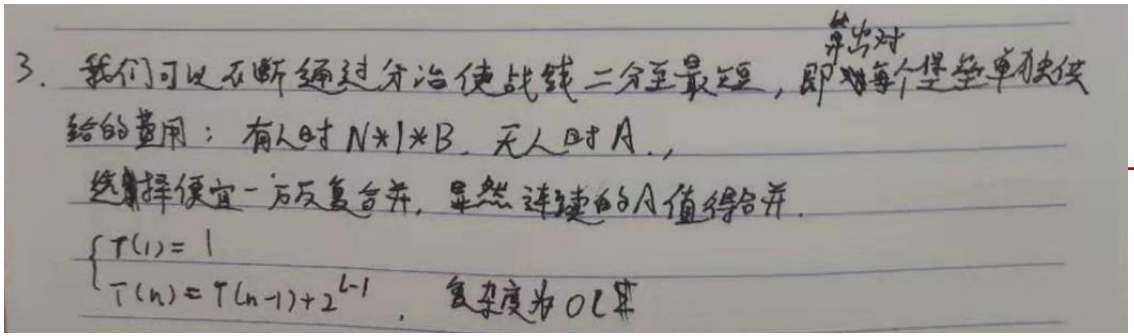
伪代码如下：

```
MinCost(n,l,left,right)
begin
while l != 0 do
if(isnull([left,right ])) then
cost ← cost + A;
end
else begin
if soilder_num([left,right ]) = 1then
cost ← cost + B ;
end
else begin
l ← l - 1;
end
end
end
End

Input:n,l
Output:min
Find cost(n,l)
begin
MinCost(n,l,1,2l-1);
MinCost(n,l,2l-1,2l);
return minimum of the two cost;
end
```

伪码格式不规范，未分析算法时间复杂度。

低分案例



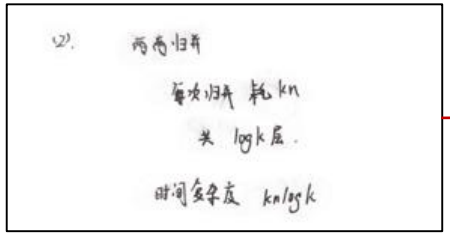
文字描述非常笼统，
没有说明时间复杂度是怎么得出的。

三、战线补给问题

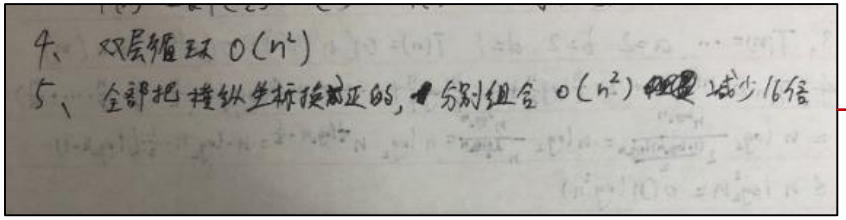
当不分段运输战线补给的时候，最小费用为$\frac{(1+2^l)*2\wedge l}{2}*2^l*B$,此时的时间复杂度为$O(n^2)$。

只用一句话笼统地描述算法，未给出算法流程，
时间复杂度没有分析流程。

低分案例



只提供了没有体现算法思路的潦草的文字描述。



只有简单的一行算法描述且描述不清楚。

低分案例

五.

显然可对所有向量取绝对值即 $v_i=(|x_i|,|y_i|)$

有最短模长为 $(|x_i|+|x_i|)^2+(|y_i|+|y_i|)^2$ 对 x 进行平均分，取一直线 l: x = t;

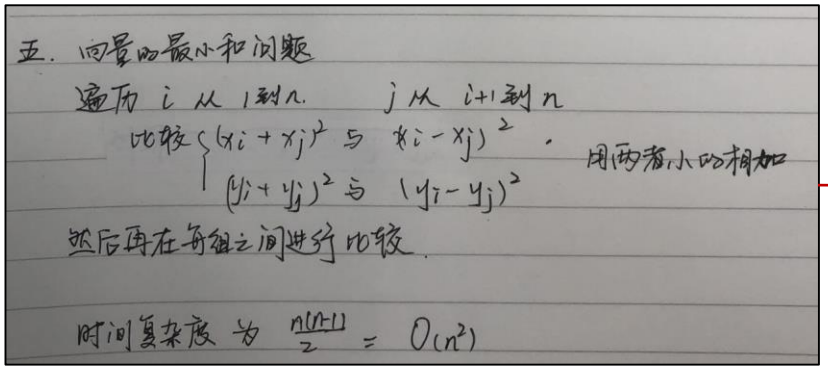
令 t 的值为 x 的中位数，对于 l 两侧分别求最小值，有两侧最小值分别得 d1,d2

令 d=(d1,d2)再设两直线 l1,l2 分别距 l 为 d 有 l1,l2 内点 a,b 若可能为最小点对，则必有 d(a,b)<d,则可知 ab 两点定在一 d*2d 的空间中。又可知任何一侧点的距离都不小于 d，可推出矩形中至多有 6 个平面中的点，即检查 6*n/2 对点距离，y 轴同理分析可得

有 T(n) = 2T(n/2) + n

O(nlogn)

未给出时间复杂度分析流程。



算法流程描述不详细，

5

本题与书例 2.5 解法类似，可将两向量相加模长最小转换为求两点间最短距离，算法时间复杂度为$O(n\log^2 n)$

算法描述过于笼统

提纲

算法表示

案例分析

撰写工具

Overleaf在线编辑器

- 在线多人协作编辑器
  - https://www.overleaf.com/
  - https://www.overleaf.com/latex/templates/tagged/algorithm

```
%最大子数组暴力算法
    \begin{algorithm}
        %\caption{暴力算法}
%       \begin{spacing}{0.8}
            \KwIn{一个大小为$n$的数组$A[1..n]$}
            \KwOut{最大子数组之和$Vmax$}
            $Vmax$ $\leftarrow$ $A[1]$\;
            \For{$i$ $\leftarrow$ $1$ $to$ $n$}
            {

                \For{$j$ $\leftarrow$ $i$ $to$ $n$}{
                    $V$ $\leftarrow$ $0$;{\color{blue}// 计算 $V(i,j)$}
                    \For{$x$ $\leftarrow$ $i$ $to$ $j$}{
                        $V$ $\leftarrow$ $V$ + $A[x]$\;
                    }
                    \If{$V$ >$ $Vmax$ }{
                    $Vmax$ $\leftarrow$ $V$\;
                    }
                }
            }
            \KwRet $Vmax$\;
%       \end{spacing}
    \end{algorithm}

%最大子数组 递推算法
\begin{algorithm}
    %\caption{data-reuse algorithms}

    \KwIn{一个大小为$n$的数组$A[1..n]$}
    \KwOut{最大子数组之和$Vmax$}
    $Vmax$ $\leftarrow$ $A[1]$\;
    \For{$i$ $\leftarrow$ $1$ $to$ $n$}{
        $V$ $\leftarrow$ $0$;{\color{blue}// 计算 $V(i,j)$}\\}
        \For{$j$ $\leftarrow$ $i$ $to$ $n$}{

            $V$ $\leftarrow$ $V$ + $A[j]$\;
        }
        \If{$V$ $>$ $Vmax$ }{
          $Vmax$ $\leftarrow$ $V$\;
        }
    }
    \KwRet $Vmax$\;
\end{algorithm}
```

输入: 一个大小为 n 的数组 $A[1..n]$
输出: 最大子数组之和 $Vmax$
$Vmax \leftarrow A[1]$;
for $i \leftarrow 1$ *to* n **do**
  for $j \leftarrow i$ *to* n **do**
    $V \leftarrow 0$;// 计算 $V(i,j)$
    for $x \leftarrow i$ *to* j **do**
      $V \leftarrow V + A[x]$;
    end
    if $V > Vmax$ **then**
      $Vmax \leftarrow V$;
    end
  end
end
return $Vmax$;

输入: 一个大小为 n 的数组 $A[1..n]$
输出: 最大子数组之和 $Vmax$
$Vmax \leftarrow A[1]$;
for $i \leftarrow 1$ *to* n **do**
  $V \leftarrow 0$;// 计算 $V(i,j)$
  for $j \leftarrow i$ *to* n **do**
    $V \leftarrow V + A[j]$;
    end
    if $V > Vmax$ **then**
      $Vmax \leftarrow V$;
    end
end
return $Vmax$;

Latex相关

- Latex官网
  - https://www.latex-project.org/
- Latex下载
  - https://www.latex-project.org/get/
- 使用教程
  - https://www.latex-tutorial.com/tutorials/
  - https://www.tug.org/twg/mactex/tutorials/ltxprimer-1.0.pdf

Divide-and-Conquer :

Counting Inversion Problem
and
Polynomial Multiplication Problem

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Review to Divide-and-Conquer Paradigm

- Divide-and-conquer (D&C) is an important algorithm design paradigm.

Review to Divide-and-Conquer Paradigm

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**

    Dividing a given problem into two or more subproblems (ideally of approximately equal size)

Review to Divide-and-Conquer Paradigm

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**

    Dividing a given problem into two or more subproblems (ideally of approximately equal size)

  - **Conquer**

    Solving each subproblem (directly if small enough or recursively)

Review to Divide-and-Conquer Paradigm

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**

    Dividing a given problem into two or more subproblems (ideally of approximately equal size)

  - **Conquer**

    Solving each subproblem (directly if small enough or recursively)

  - **Combine**

    Combining the solutions of the subproblems into a global solution

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Counting inversions

Music site tries to match your song preferences with others.

- You rank *n* songs.
- Music site consults database to find people with similar tastes.

Counting inversions

Music site tries to match your song preferences with others.

- You rank *n* songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Counting inversions

Music site tries to match your song preferences with others.

- You rank *n* songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: 1, 2, …, n.

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Counting inversions

Music site tries to match your song preferences with others.

- You rank *n* songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: 1, 2, …, n.
- Your rank: a_1, a_2, …, a_n.

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Counting inversions

Music site tries to match your song preferences with others.

- You rank *n* songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: 1, 2, …, n.
- Your rank: a_1, a_2, …, a_n.
- Songs i and j are inverted if i < j, but $a_i > a_j$

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

A Brute Force Algorithm

List each pair i < j and count the inversions.

Input: L
Output: r
$r \leftarrow 0;$

A Brute Force Algorithm

List each pair i < j and count the inversions.

Input: L
Output: r
$r \leftarrow 0$;
for $i \leftarrow 1$ *to* $L.length$ **do**

A Brute Force Algorithm

List each pair i < j and count the inversions.

```
Input: L
Output: r
r ← 0;
for i ← 1 to L.length do
    for j ← i + 1 to L.length do
        if L[i] > L[j] then
```

A Brute Force Algorithm

List each pair i < j and count the inversions.

```
Input: L
Output: r
r ← 0;
for i ← 1 to L.length do
    for j ← i + 1 to L.length do
        if L[i] > L[j] then
            r ← r + 1;
```

A Brute Force Algorithm

List each pair i < j and count the inversions.

```
Input: L
Output: r
r ← 0;
for i ← 1 to L.length do
    for j ← i + 1 to L.length do
        if L[i] > L[j] then
            r ← r + 1;
        end
    end
end
return
```

A Brute Force Algorithm

List each pair i < j and count the inversions.

```
Input: L
Output: r
r ← 0;
for i ← 1 to L.length do
    for j ← i + 1 to L.length do
        if L[i] > L[j] then
            r ← r + 1;
        end
    end
end
return r;
```

A Brute Force Algorithm

List each pair i < j and count the inversions.

```
Input: L
Output: r
r ← 0;
for i ← 1 to L.length do
    for j ← i + 1 to L.length do
        if L[i] > L[j] then
            r ← r + 1;
        end
    end
end
return r;
```

$O(n^2)$ comparisons and additions.

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Review to Merge Sort

Mergesort(A, left, right)

```
if left < right then
    center ← ⌊(left + right)/2⌋;
    Mergesort(A, left, center);
    Mergesort(A, center+1, right);
    "Merge" the two sorted arrays;
end
```

- To sort the entire array $A[1 \ldots n]$, we make the initial call Mergesort(A, 1, n).
- Key subroutine: "Merge"

Counting inversions: divide-and-conquer

Input

14	7	18	3	10	19	11	23	2	25	16	17

Counting inversions: divide-and-conquer

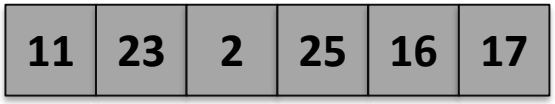
- Divide: separate list into two halves A and B.

Input

Counting inversions: divide-and-conquer

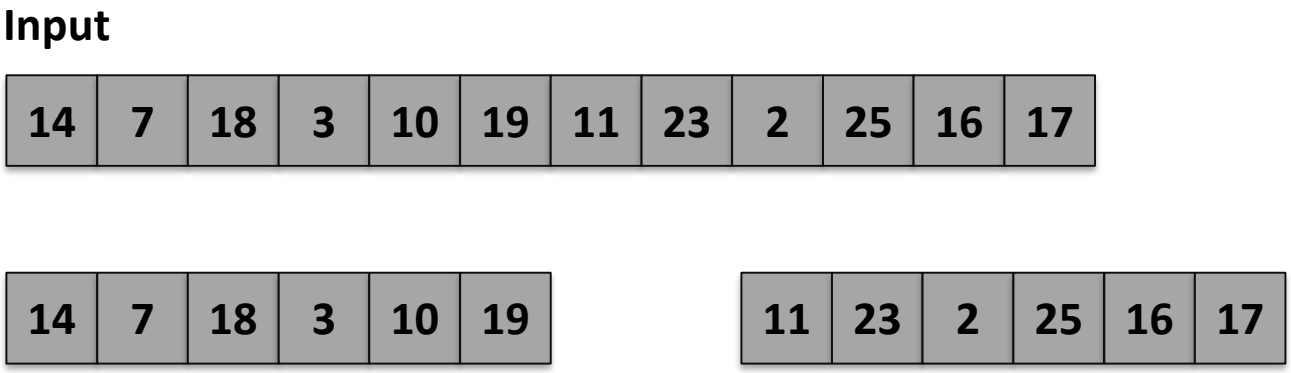
- Divide: separate list into two halves A and B.

Input



Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.



Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.

Input

Count inversions in left half A

14-7,14-3,14-10,7-3,18-3,18-10

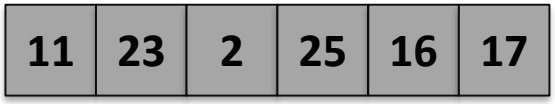
Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.

Input

Count inversions in left half A

14-7,14-3,14-10,7-3,18-3,18-10



Count inversions in right half B

11-2,23-2,23-16,23-17,25-16,25-17

Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with a∈A and b∈B.

Input

14	7	18	3	10	19	11	23	2	25	16	17

14	7	18	3	10	19

Count inversions in left half A

14-7,14-3,14-10,7-3,18-3,18-10

11	23	2	25	16	17

Count inversions in right half B

11-2,23-2,23-16,23-17,25-16,25-17

Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with a∈A and b∈B.

Input

14	7	18	3	10	19	11	23	2	25	16	17

14	7	18	3	10	19

Count inversions in left half A

14-7,14-3,14-10,7-3,18-3,18-10

11	23	2	25	16	17

Count inversions in right half B

11-2,23-2,23-16,23-17,25-16,25-17

Count inversions (a,b) with a∈A and b∈B

14-11,14-2,7-2,18-11,18-2,18-16,18-17,3-2,10-2,19-11,19-2,19-16,19-17

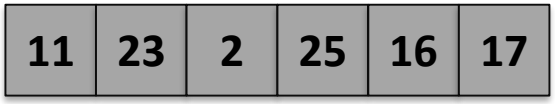
Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with a∈A and b∈B.
- Return sum of three counts.

Input

Count inversions in left half A

14-7,14-3,14-10,7-3,18-3,18-10



Count inversions in right half B

11-2,23-2,23-16,23-17,25-16,25-17

Count inversions (a,b) with a∈A and b∈B

14-11,14-2,7-2,18-11,18-2,18-16,18-17,3-2,10-2,19-11,19-2,19-16,19-17

Counting inversions: divide-and-conquer

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with a∈A and b∈B.
- Return sum of three counts.

Input

14	7	18	3	10	19	11	23	2	25	16	17

14	7	18	3	10	19

Count inversions in left half A

14-7,14-3,14-10,7-3,18-3,18-10

11	23	2	25	16	17

Count inversions in right half B

11-2,23-2,23-16,23-17,25-16,25-17

Count inversions (a,b) with a∈A and b∈B

14-11,14-2,7-2,18-11,18-2,18-16,18-17,3-2,10-2,19-11,19-2,19-16,19-17

Output

6+6+13 =25

How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

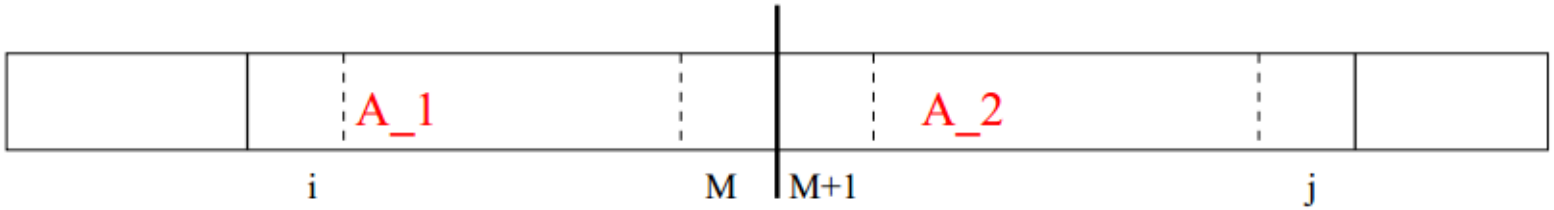
List A

14	7	18	3	10	19

List B

11	23	2	25	16	17

Review to the Conquer Step of MCS Problem



A_1 is in the form $A[i \ldots m]$, $V(i, m) = V(i+1, m) + A[i]$

```
MAX ← A[m];
SUM ← A[m];
for i ← m − 1 downto 1 do
    SUM ← SUM +A[i];
    if SUM > MAX then
        MAX ← SUM;
    end
end
A1=MAX;
```

How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

List A

14	7	18	3	10	19

List B

11	23	2	25	16	17

How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.

List A

14	7	18	3	10	19

List B

11	23	2	25	16	17

How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.

Sort A

3	7	10	14	18	19

List B

11	23	2	25	16	17

How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.

Sort A

3	7	10	14	18	19

Sort B

2	11	16	17	23	25

How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,

How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.

Sort A

3	7	10	14	18	19

Sort B

2	11	16	17	23	25

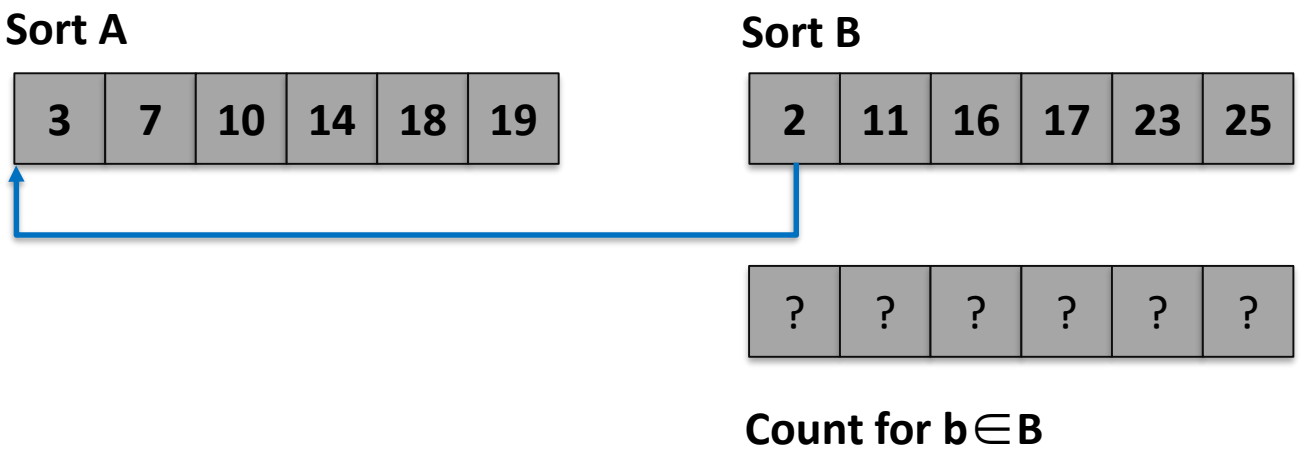
How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.



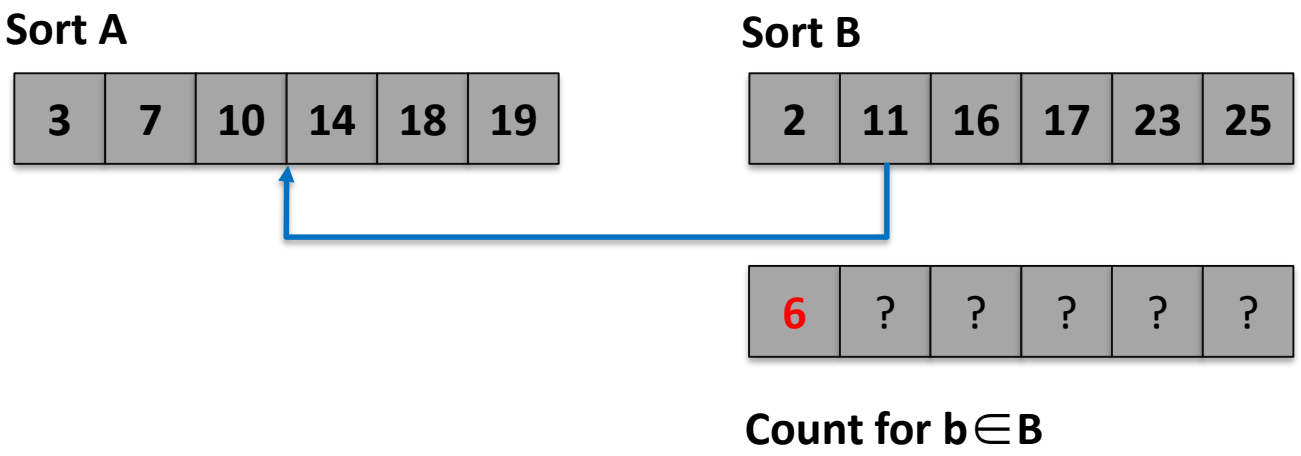
How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.



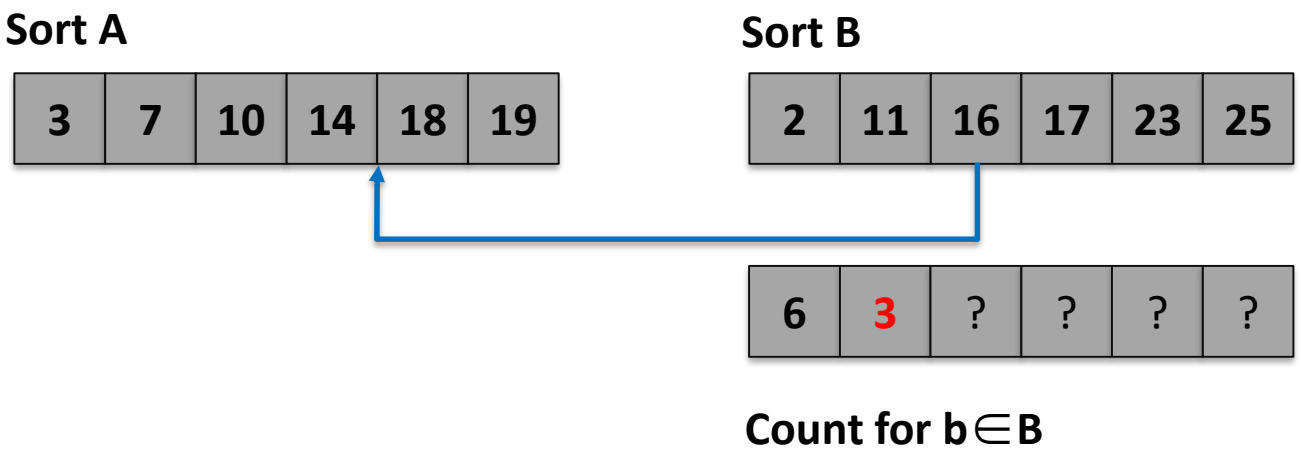
How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.



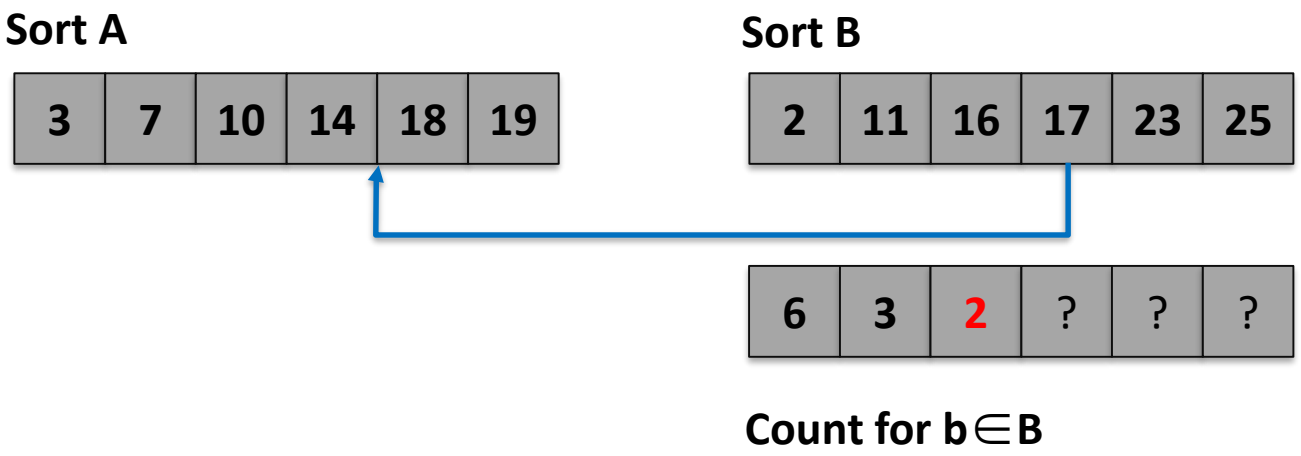
How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.



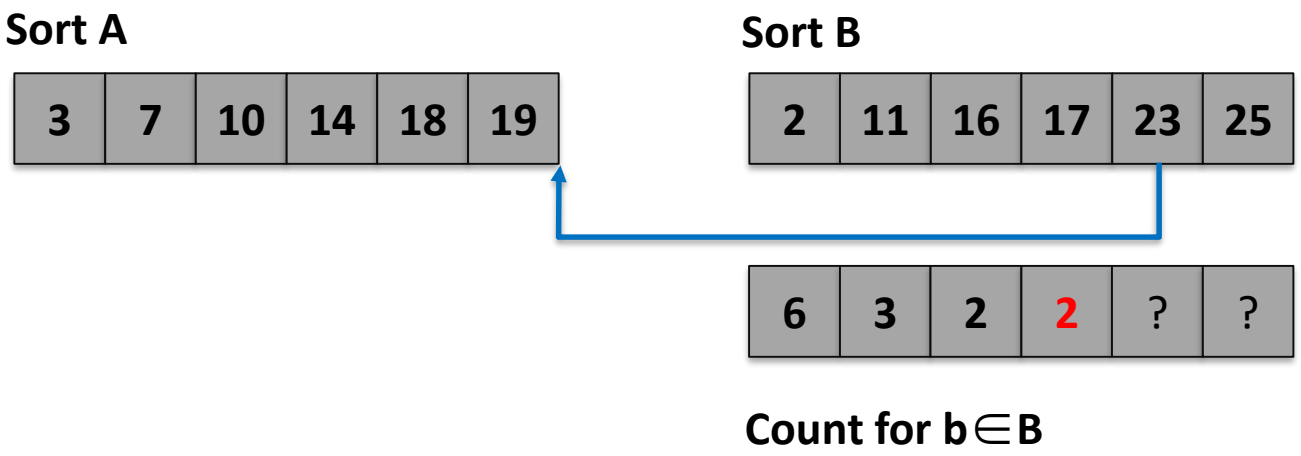
How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.



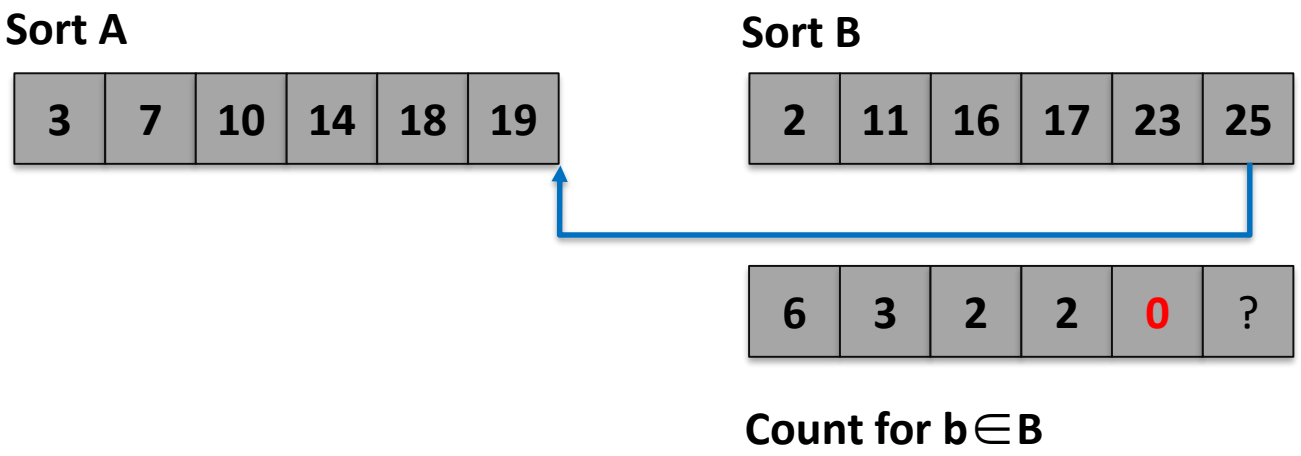
How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.



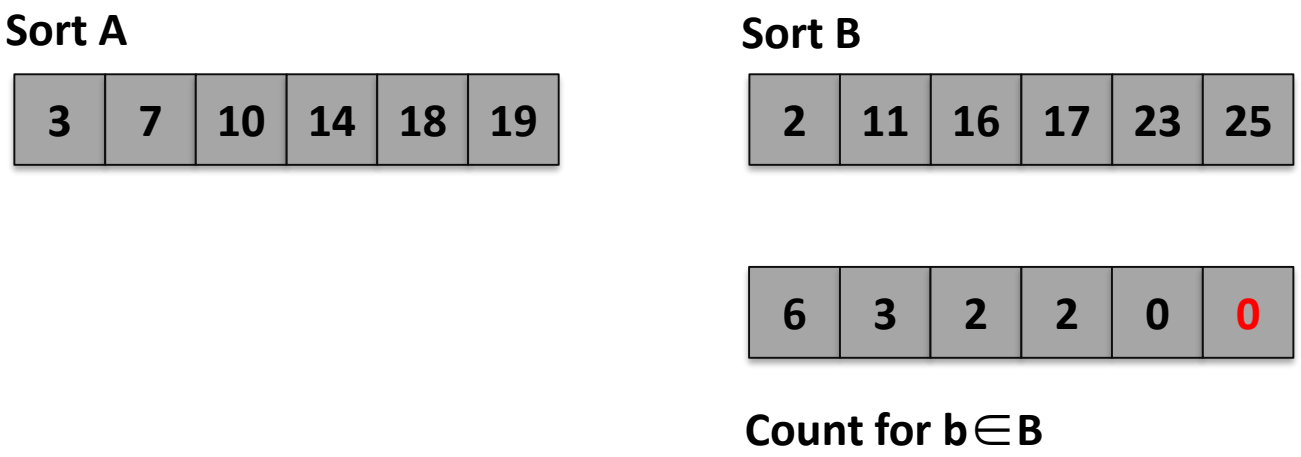
How to combine two subproblems?

Q. How to count inversions (a, b) with a $\in$ A and b $\in$ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b $\in$ B,
  - binary search in A to find how many elements in A are greater than b.



How to combine two subproblems?

Q. How to count inversions (a, b) with a ∈ A and b ∈ B?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element b ∈ B,
  - binary search in A to find how many elements in A are greater than b.

Sort A

3	7	10	14	18	19

Sort B

2	11	16	17	23	25

Inversions between A and B: 6+3+2+2=13

6	3	2	2	0	0

Count for b∈B

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then

Combine two subproblems: Improvement

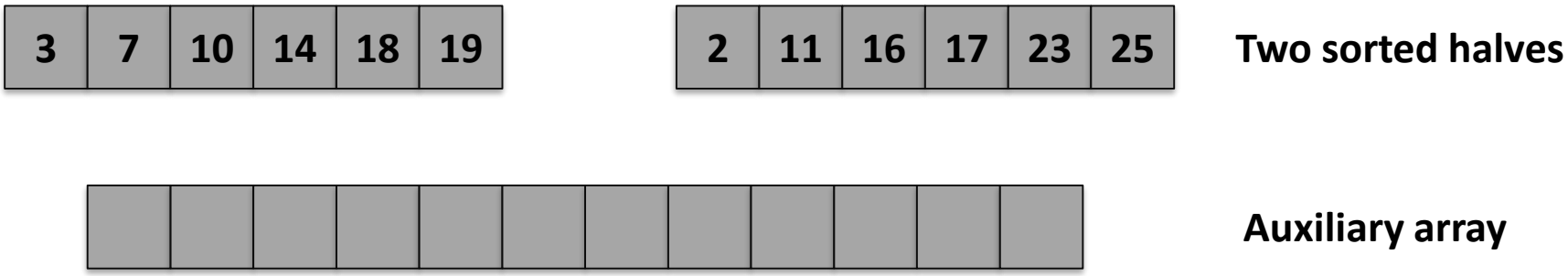
Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

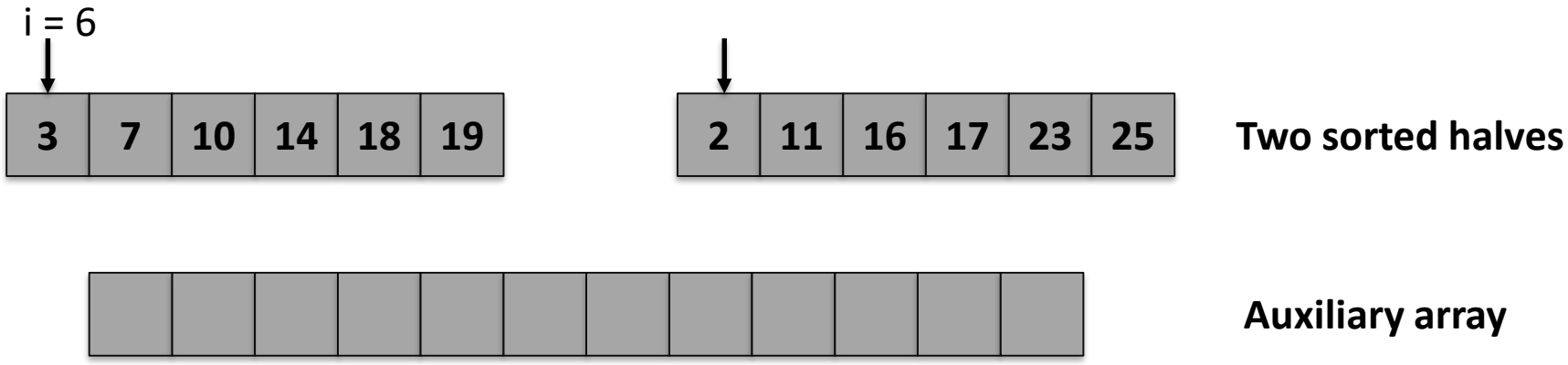
- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.



Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

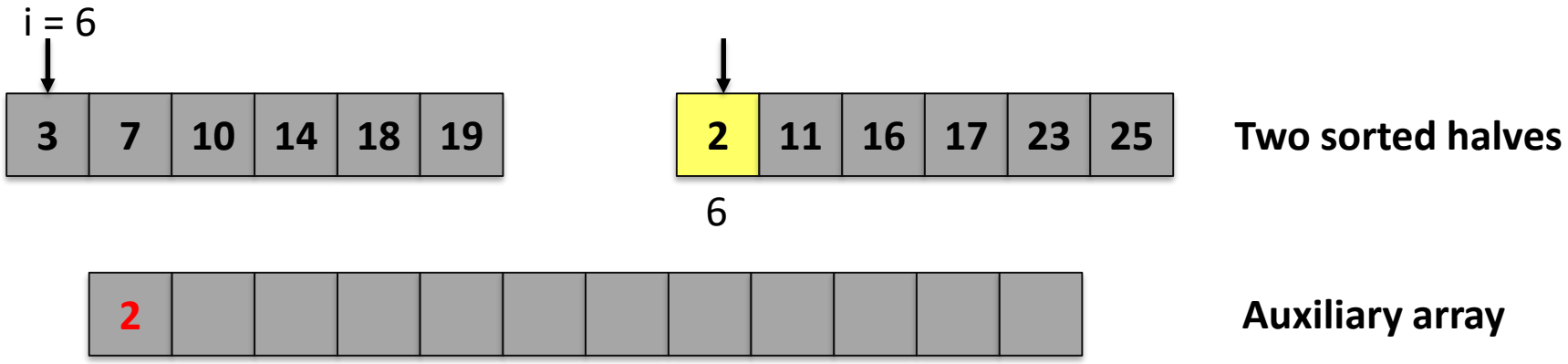


Total:

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

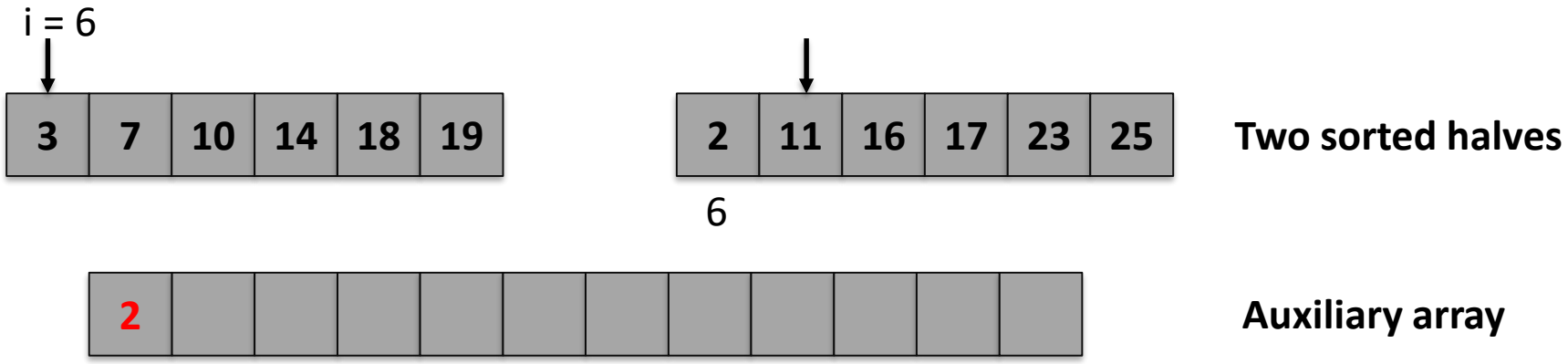


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

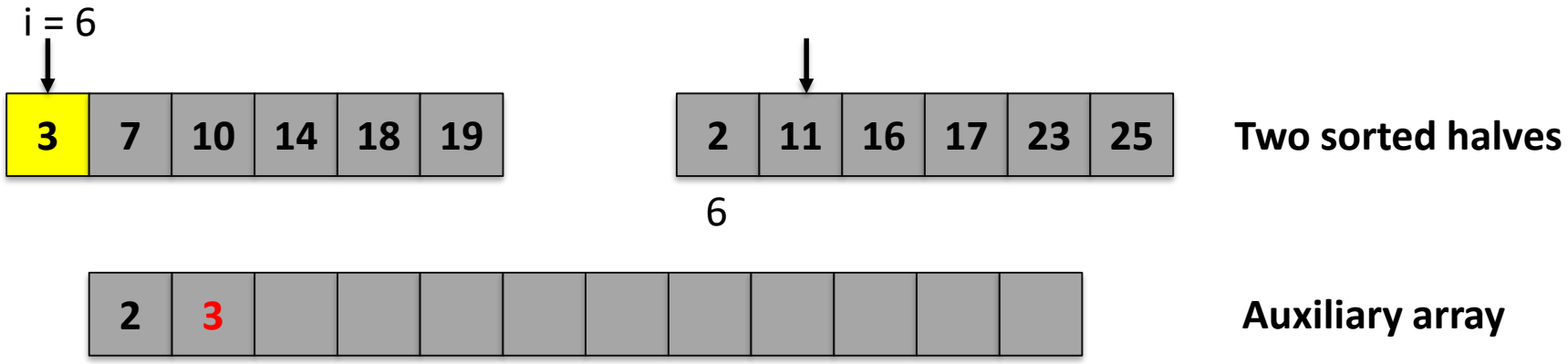


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

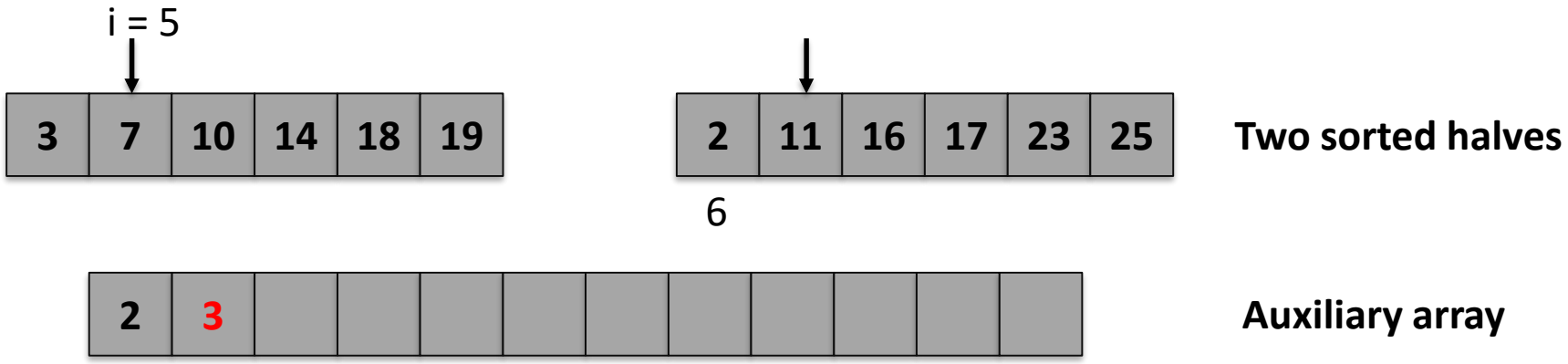


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

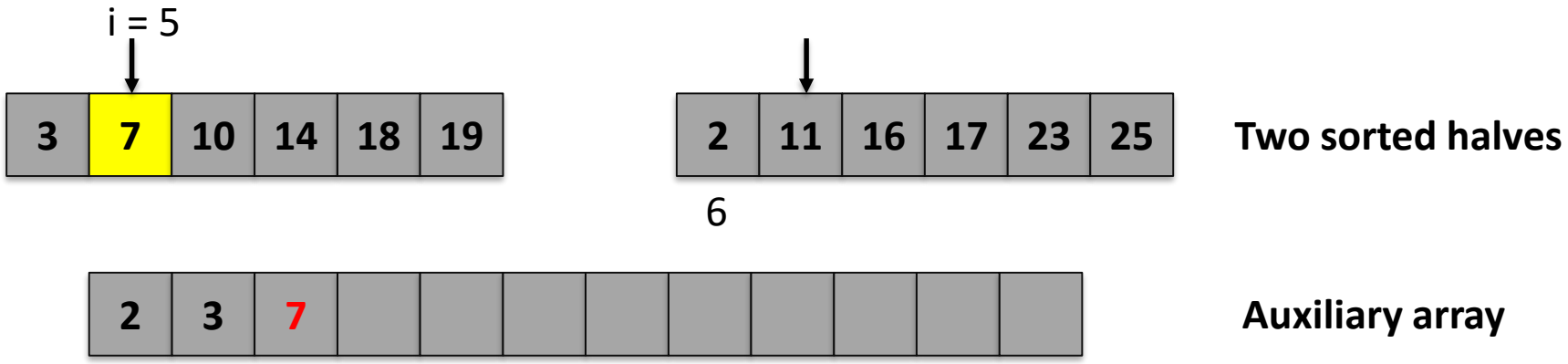


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

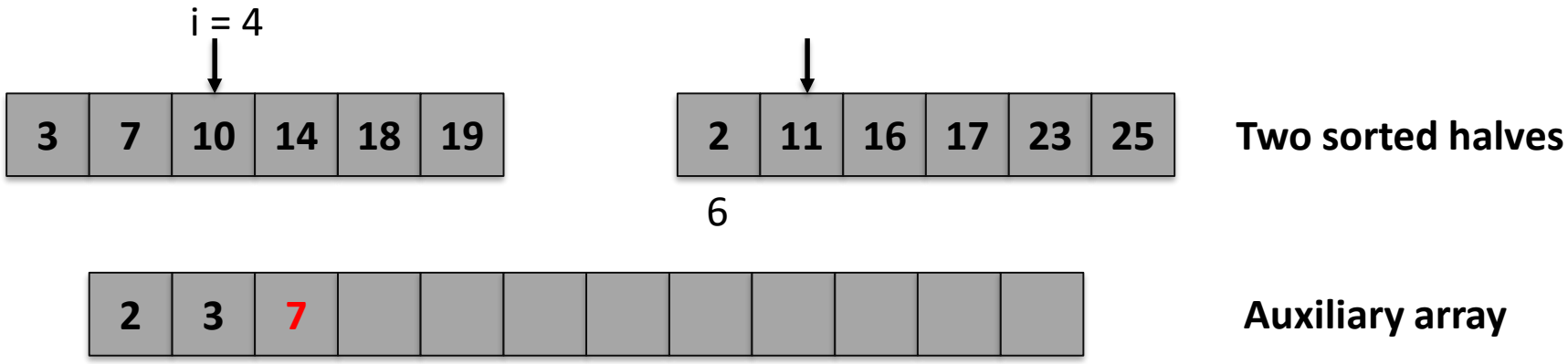


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

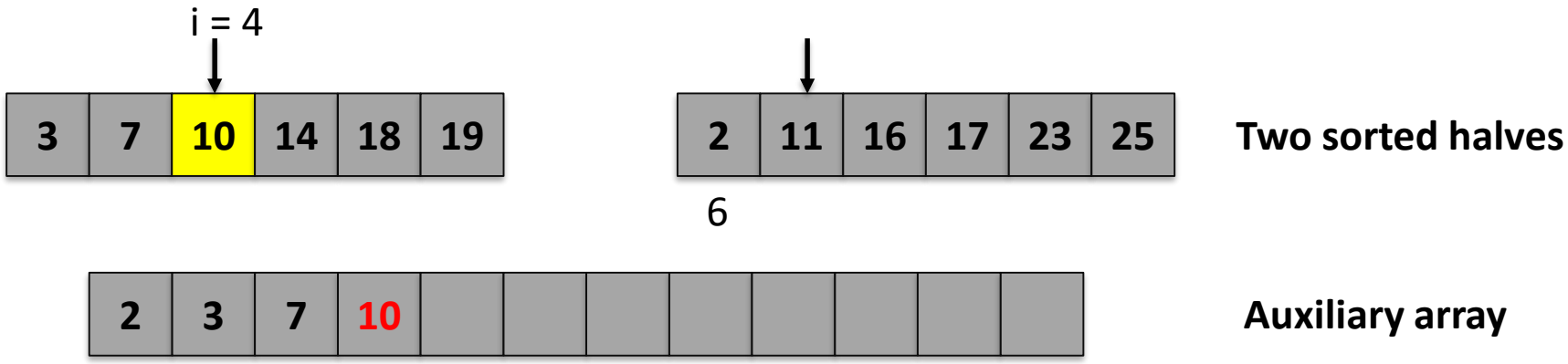


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

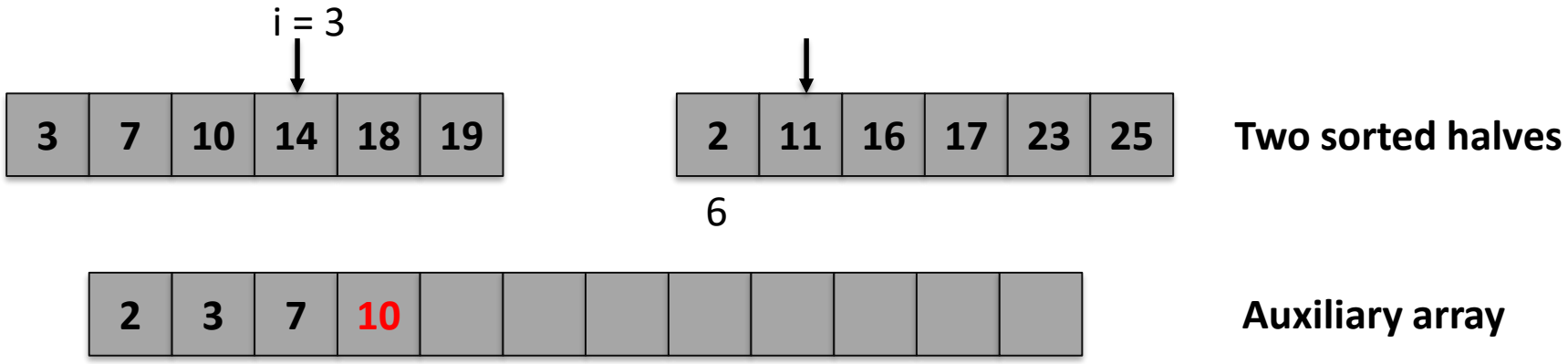


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

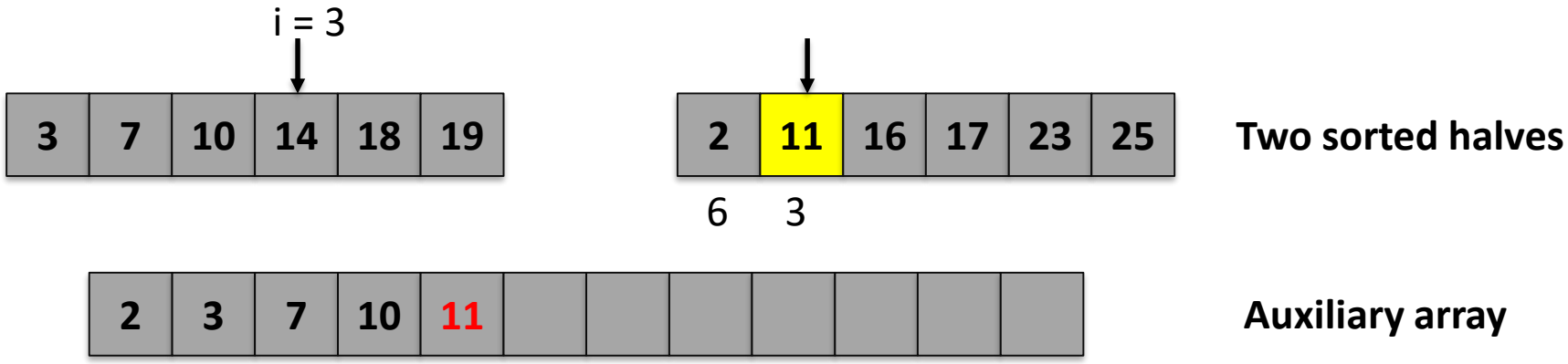


Total: 6

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

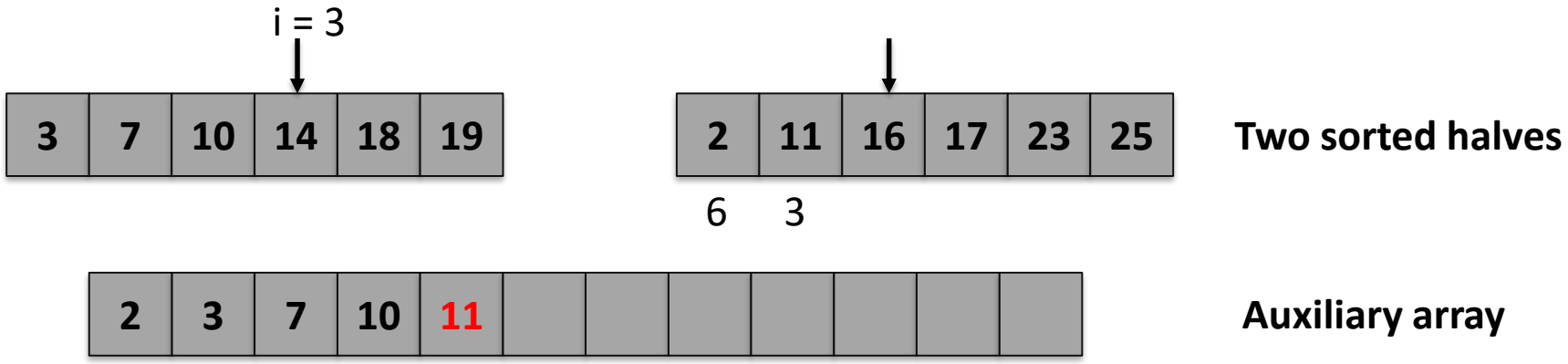


Total: 6+3

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

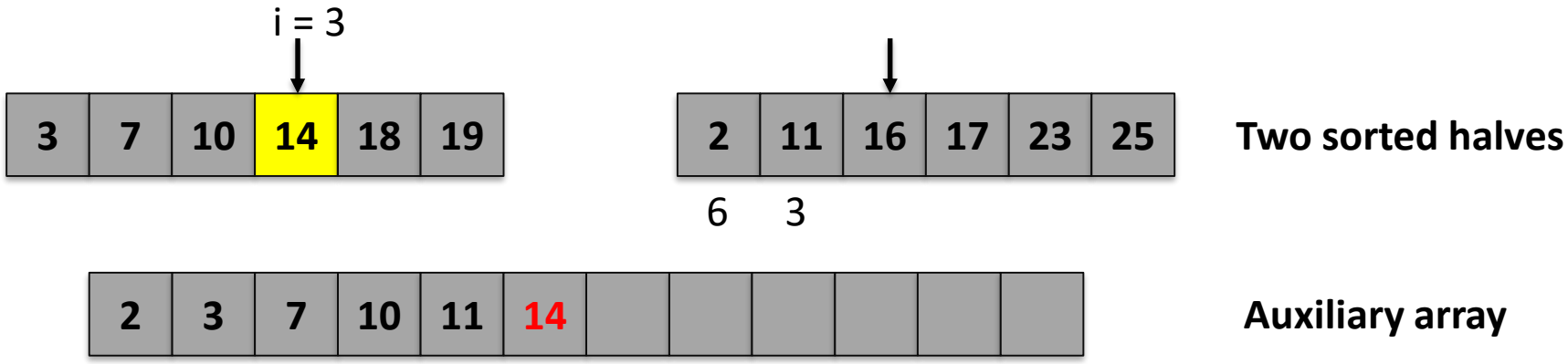


Total: 6+3

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

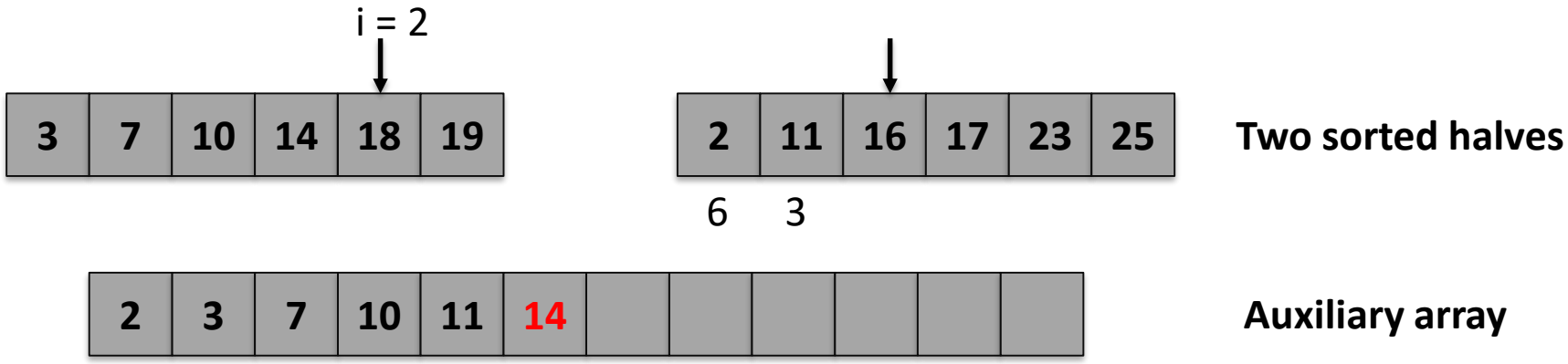


Total: 6+3

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

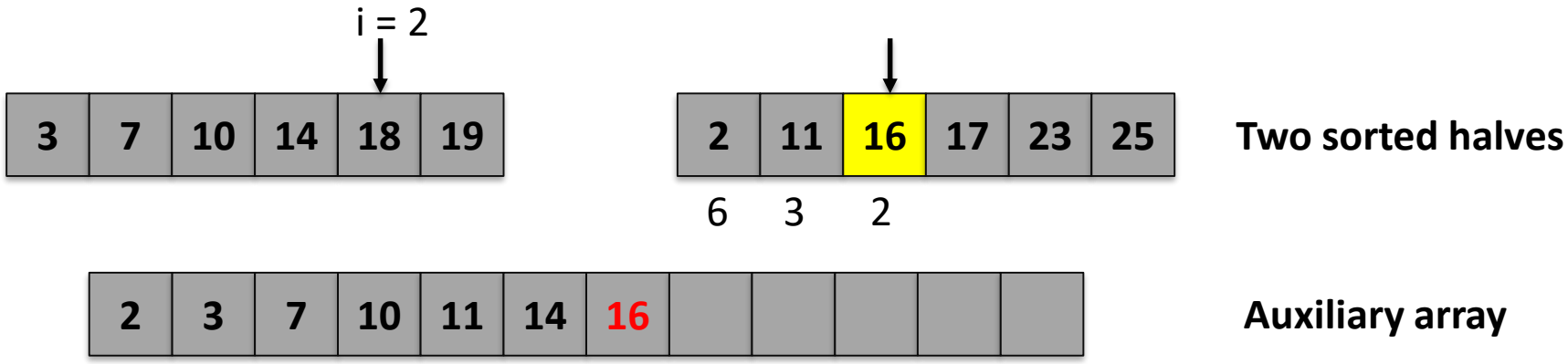


Total: 6+3

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

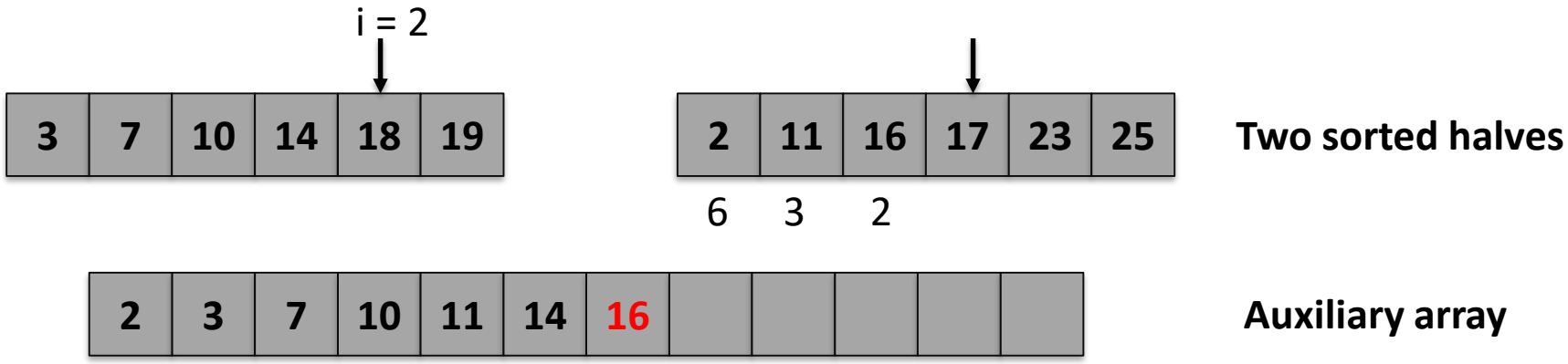


Total: 6+3+2

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

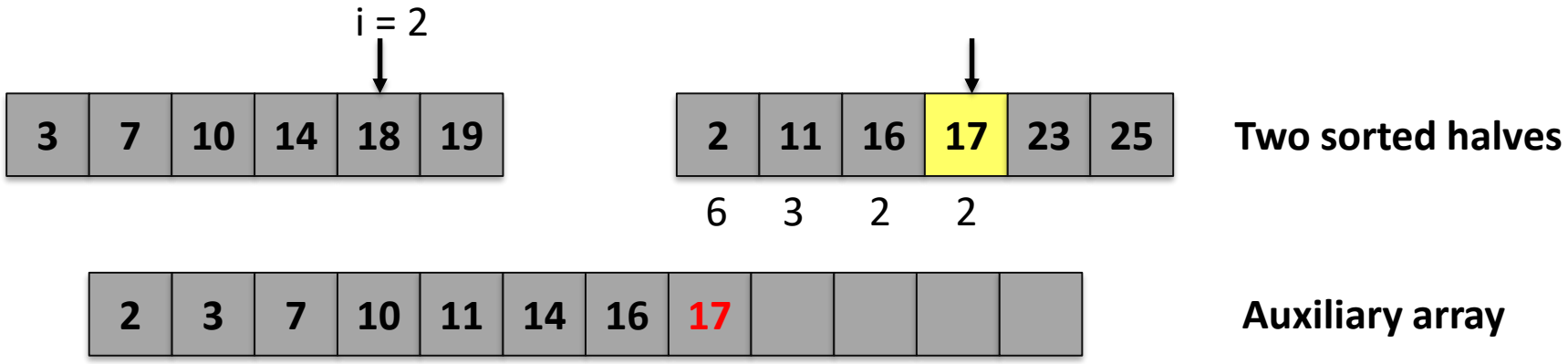


Total: 6+3+2

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

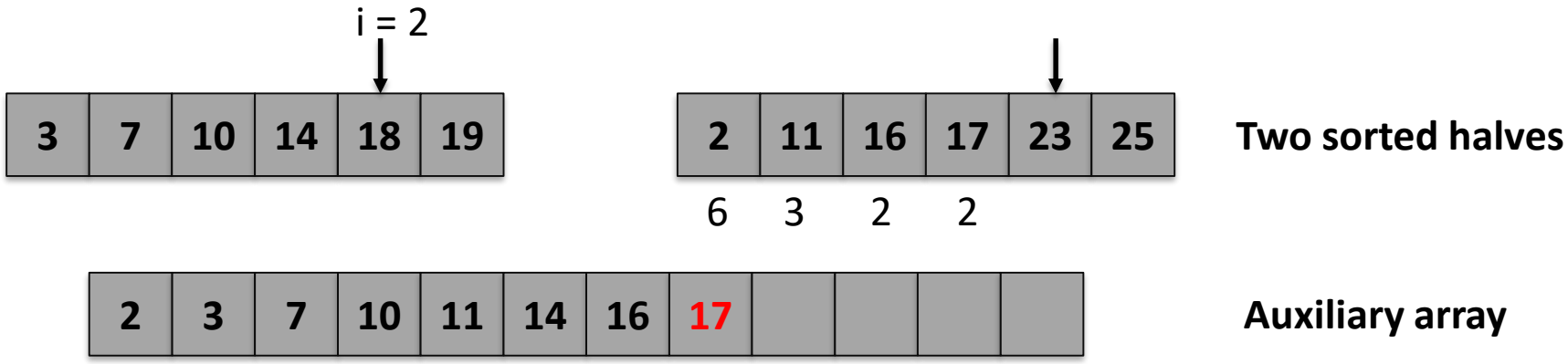


Total: 6+3+2+2

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

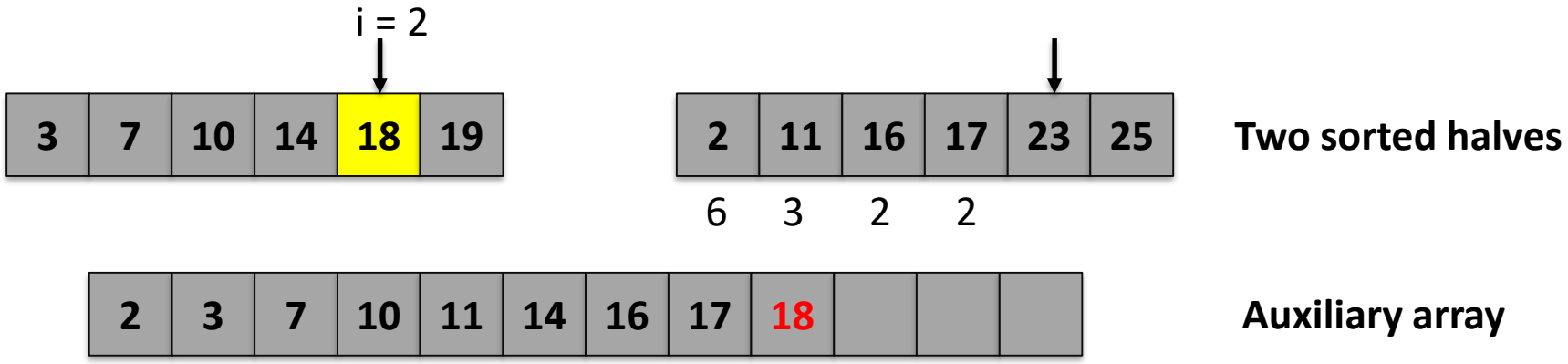


Total: 6+3+2+2

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

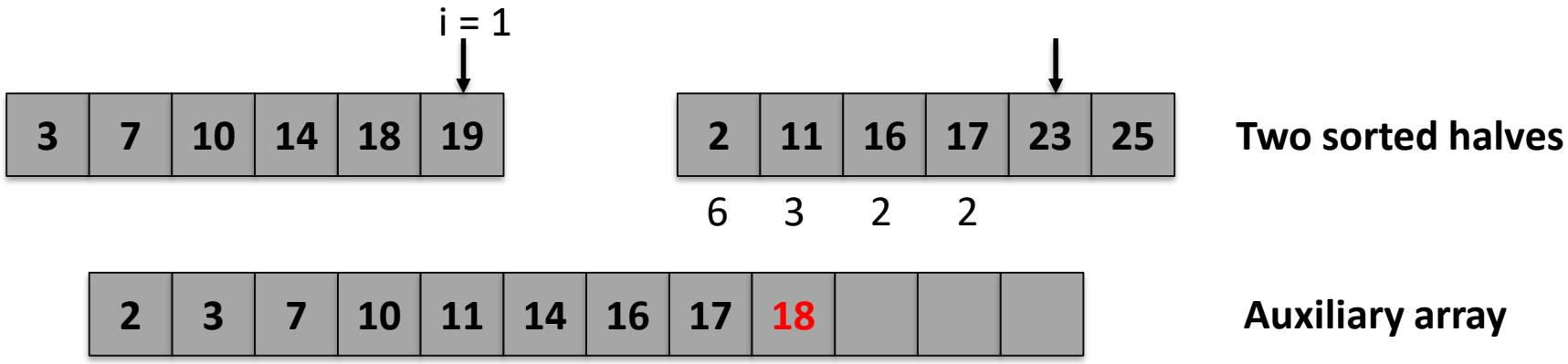


Total: 6+3+2+2

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

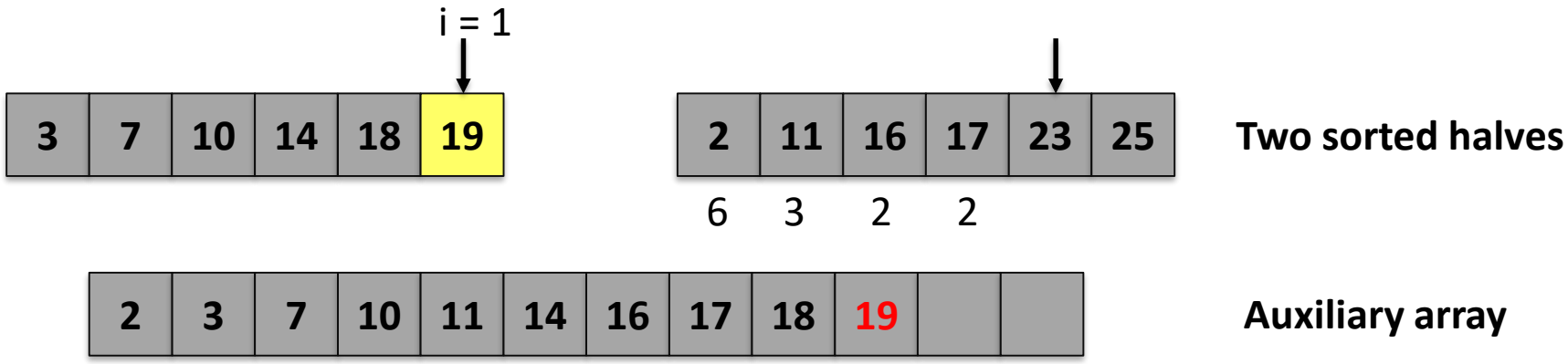


Total: 6+3+2+2

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

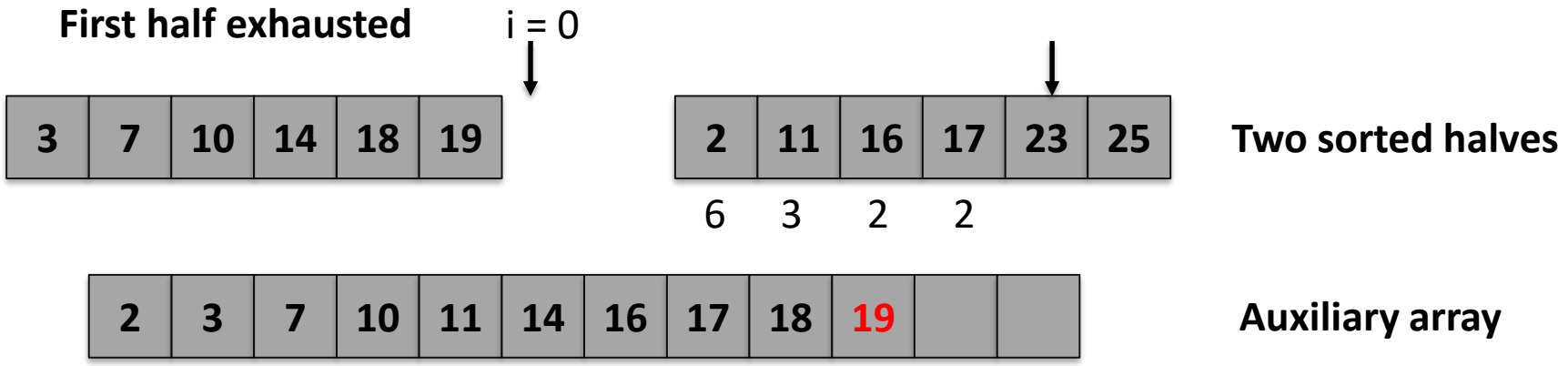


Total: 6+3+2+2

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

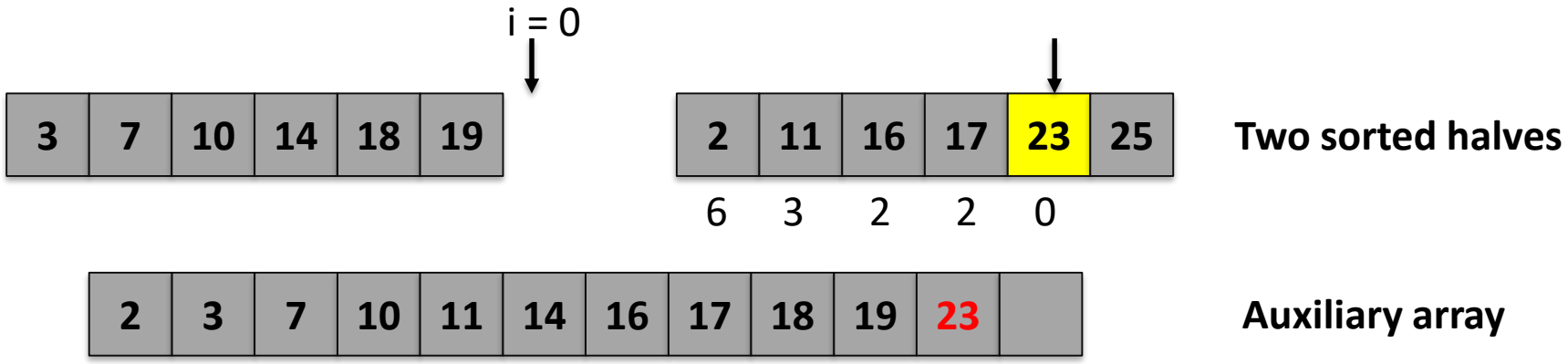


Total: 6+3+2+2

Combine two subproblems: Improvement

Count inversions (a, b) with a ∈ A and b ∈ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

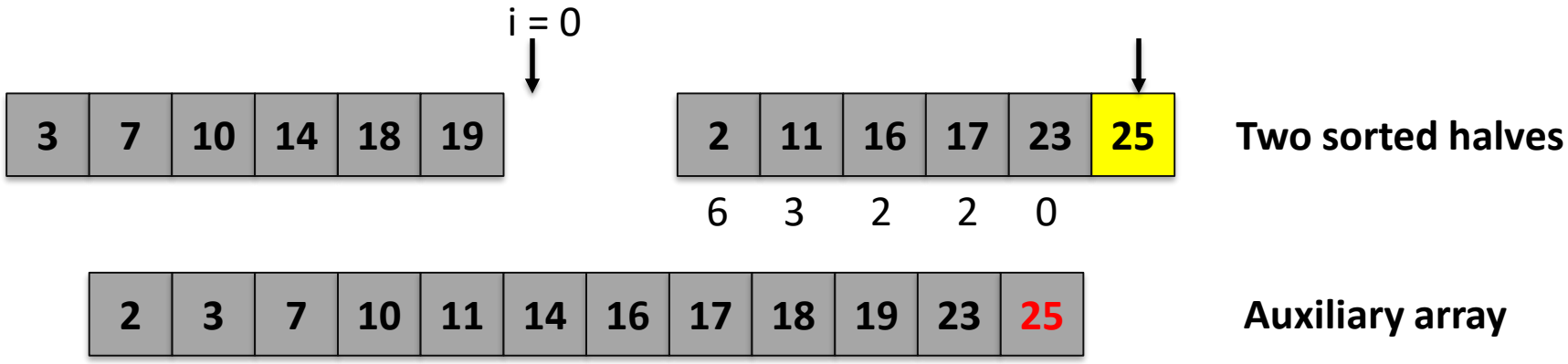


Total: 6+3+2+2+0

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

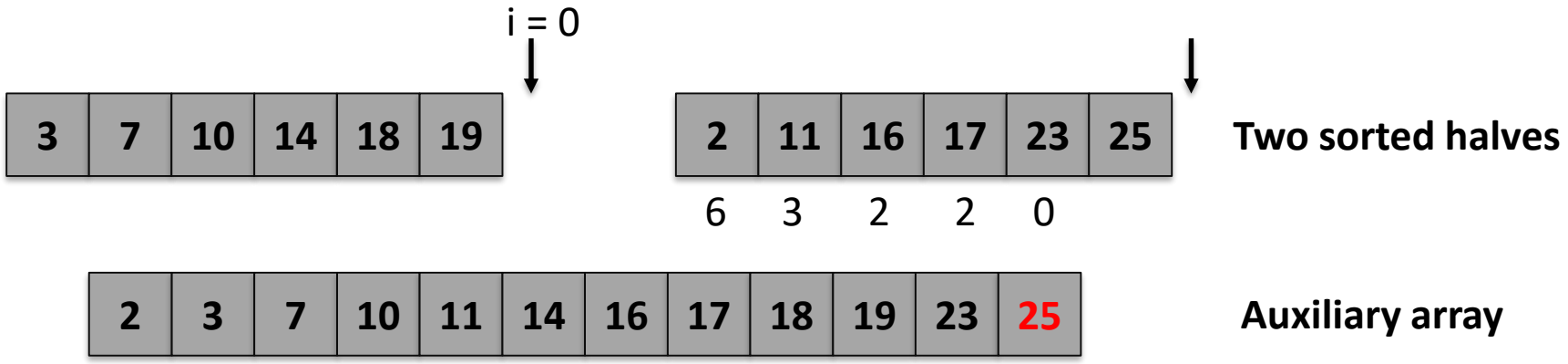


Total: 6+3+2+2+0+0

Combine two subproblems: Improvement

Count inversions (a, b) with a $\in$ A and b $\in$ B, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
  - If $a_i < b_j$, then a_i is not inverted with any element left in B.
  - If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.



Total: 6+3+2+2+0+0 = 13

Combine two subproblems: Improvement

Merge-and-Count(A, B)

Input: A, B
Output: r, L
$r \leftarrow 0, L \leftarrow \emptyset$;

Combine two subproblems: Improvement

Merge-and-Count(A, B)

Input: A, B
Output: r, L
$r \leftarrow 0, L \leftarrow \emptyset$;
while *both* A *and* B *are not empty* **do**
| // Let a and b represent the first element of A and B, repectively

Combine two subproblems: Improvement

Merge-and-Count(A, B)

Input: A, B
Output: r, L
$r \leftarrow 0, L \leftarrow \emptyset$;
while *both* A *and* B *are not empty* **do**
  // Let a and b represent the first element of A and B, repectively
  if $a < b$ **then**

Combine two subproblems: Improvement

Merge-and-Count(A, B)

Input: A, B
Output: r, L
$r \leftarrow 0, L \leftarrow \emptyset$;
while *both* A *and* B *are not empty* **do**
  // Let a and b represent the first element of A and B, repectively
  if $a < b$ **then**
    Move a to the back of L;//$A.length$ is decreased by 1;
  end
  else

Combine two subproblems: Improvement

Merge-and-Count(A, B)

Input: A, B
Output: r, L
$r \leftarrow 0, L \leftarrow \emptyset$;
while *both* A *and* B *are not empty* **do**
  // Let a and b represent the first element of A and B, repectively
  if $a < b$ **then**
    Move a to the back of L;//$A.length$ is decreased by 1;
  end
  else
    Increase r by $A.length$;
    Move b to the back of L;
  end
end

Combine two subproblems: Improvement

Merge-and-Count(A, B)

```
Input: A, B
Output: r, L
r ← 0, L ← ∅;
while both A and B are not empty do
    // Let a and b represent the first element of A and B, repectively
    if a < b then
        Move a to the back of L;//A.length is decreased by 1;
    end
    else
        Increase r by A.length;
        Move b to the back of L;
    end
end
if A is not empty then
```

Combine two subproblems: Improvement

Merge-and-Count(A, B)

```
Input: A, B
Output: r, L
r ← 0, L ← ∅;
while both A and B are not empty do
    // Let a and b represent the first element of A and B, repectively
    if a < b then
        Move a to the back of L;//A.length is decreased by 1;
    end
    else
        Increase r by A.length;
        Move b to the back of L;
    end
end
if A is not empty then
    Move A to the back of L;
end
else
```

Combine two subproblems: Improvement

Merge-and-Count(A, B)

```
Input: A, B
Output: r, L
r ← 0, L ← ∅;
while both A and B are not empty do
    // Let a and b represent the first element of A and B, repectively
    if a < b then
        Move a to the back of L;//A.length is decreased by 1;
    end
    else
        Increase r by A.length;
        Move b to the back of L;
    end
end
if A is not empty then
    Move A to the back of L;
end
else
    Move B to the back of L;
end
return
```

Combine two subproblems: Improvement

Merge-and-Count(A, B)

```
Input: A, B
Output: r, L
r ← 0, L ← ∅;
while both A and B are not empty do
    // Let a and b represent the first element of A and B, repectively
    if a < b then
        Move a to the back of L;//A.length is decreased by 1;
    end
    else
        Increase r by A.length;
        Move b to the back of L;
    end
end
if A is not empty then
    Move A to the back of L;
end
else
    Move B to the back of L;
end
return L, r;
```

Combine two subproblems: Improvement

- For every element in A and B,

Combine two subproblems: Improvement

- For every element in A and B,
  - Only O( ) times operations are executed.

Combine two subproblems: Improvement

- For every element in A and B,
  - Only O(1) times operations are executed.

Combine two subproblems: Improvement

- For every element in A and B,
  - Only O(1) times operations are executed.

- Function *Sort-and-Count(A,B)* can be executed in O( ) time where n is the number of elements in A and B.

Combine two subproblems: Improvement

- For every element in A and B,
  - Only O(1) times operations are executed.

- Function *Sort-and-Count(A,B)* can be executed in O(n) time where n is the number of elements in A and B.

The Complete Divide-and-Conquer Algorithm

Review of The Complete MCS Algorithm

$MCS(A, s, t)$

Input: $A[s \ldots t]$ with $s \leq t$
Output: MCS of $A[s \ldots t]$
begin
  if $s = t$ **then return** $A[s]$;
  else
    $m \leftarrow \lfloor \frac{s+t}{2} \rfloor$;
    Find $MCS(A, s, m)$;
    Find $MCS(A, m+1, t)$;
    Find MCS that contains both $A[m]$ and $A[m+1]$;
    return maximum of the three sequences found
  end
end

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L
if L *is empty* **then**
  return $0, L$;
end
Divide L into two halves A and B;

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L
if L *is empty* **then**
  return $0, L$;
end
Divide L into two halves A and B;
$(r_A, A) \leftarrow$Sort-and-Count(A);//$T(\lceil \frac{n}{2} \rceil)$

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L
if L *is empty* **then**
  return $0, L$;
end
Divide L into two halves A and B;
$(r_A, A) \leftarrow$Sort-and-Count(A);//$T(\lceil \frac{n}{2} \rceil)$
$(r_B, B) \leftarrow$Sort-and-Count(B);//$T(\lfloor \frac{n}{2} \rfloor)$
$(r_L, L) \leftarrow$

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L
if L *is empty* **then**
  return $0, L$;
end
Divide L into two halves A and B;
$(r_A, A) \leftarrow$Sort-and-Count(A);//$T(\lceil \frac{n}{2} \rceil)$
$(r_B, B) \leftarrow$Sort-and-Count(B);//$T(\lfloor \frac{n}{2} \rfloor)$
$(r_L, L) \leftarrow$Merge-and-Count(A, B);//$O(n)$
return

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L
if L *is empty* **then**
  return $0, L$;
end
Divide L into two halves A and B;
$(r_A, A) \leftarrow$Sort-and-Count(A);//$T(\lceil \frac{n}{2} \rceil)$
$(r_B, B) \leftarrow$Sort-and-Count(B);//$T(\lfloor \frac{n}{2} \rfloor)$
$(r_L, L) \leftarrow$Merge-and-Count(A, B);//$O(n)$
return $r_A + r_B + r_L, L$;

The Complete Divide-and-Conquer Algorithm

Sort-and-Count(L)

Input: L
Output: r_L, L
if L *is empty* **then**
  return $0, L$;
end
Divide L into two halves A and B;
$(r_A, A) \leftarrow$Sort-and-Count(A);//$T(\lceil \frac{n}{2} \rceil)$
$(r_B, B) \leftarrow$Sort-and-Count(B);//$T(\lfloor \frac{n}{2} \rfloor)$
$(r_L, L) \leftarrow$Merge-and-Count(A, B);//$O(n)$
return $r_A + r_B + r_L, L$;

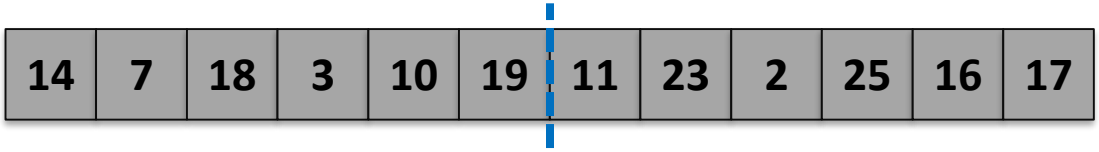
$$T(n) = \begin{cases} O(1), & if\ n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n) & otherwise \end{cases}$$

Example

Divide

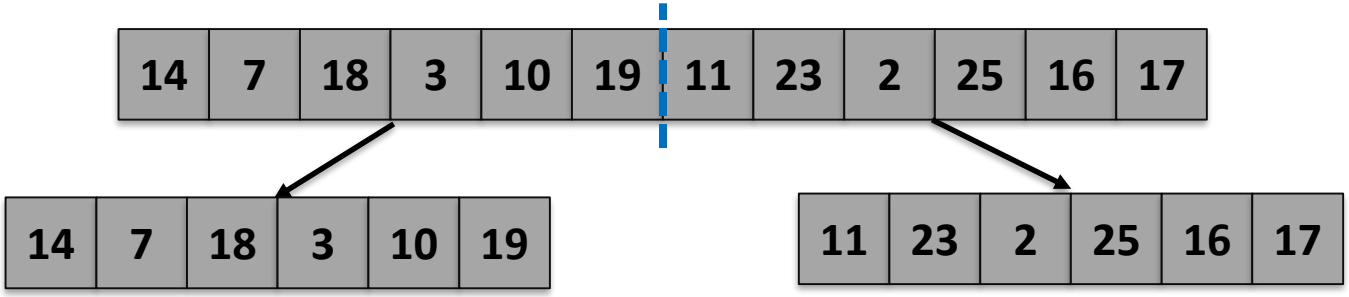
Example

Divide



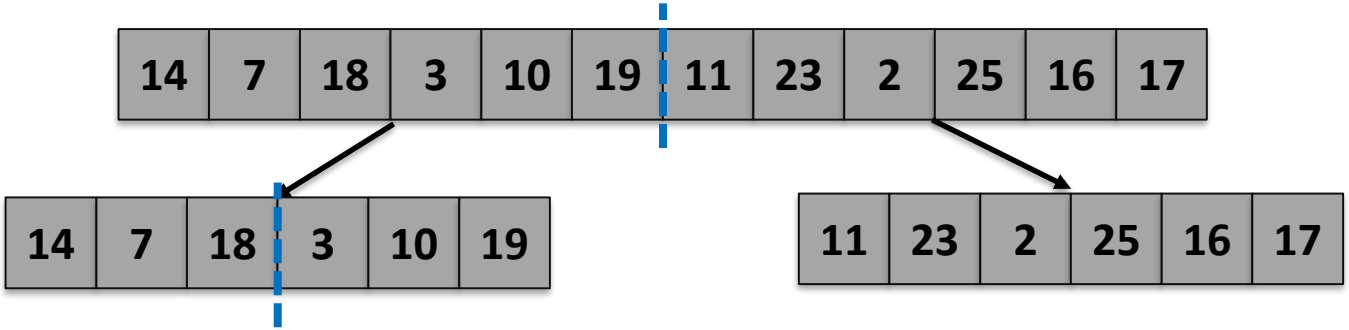
Example

Divide



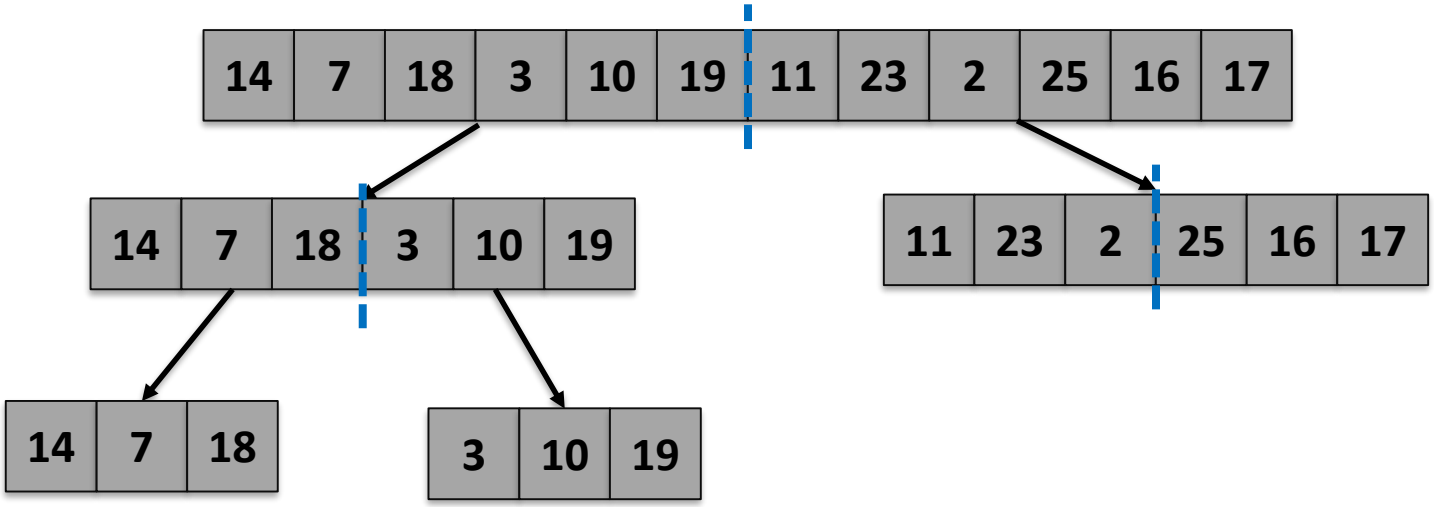
Example

Divide



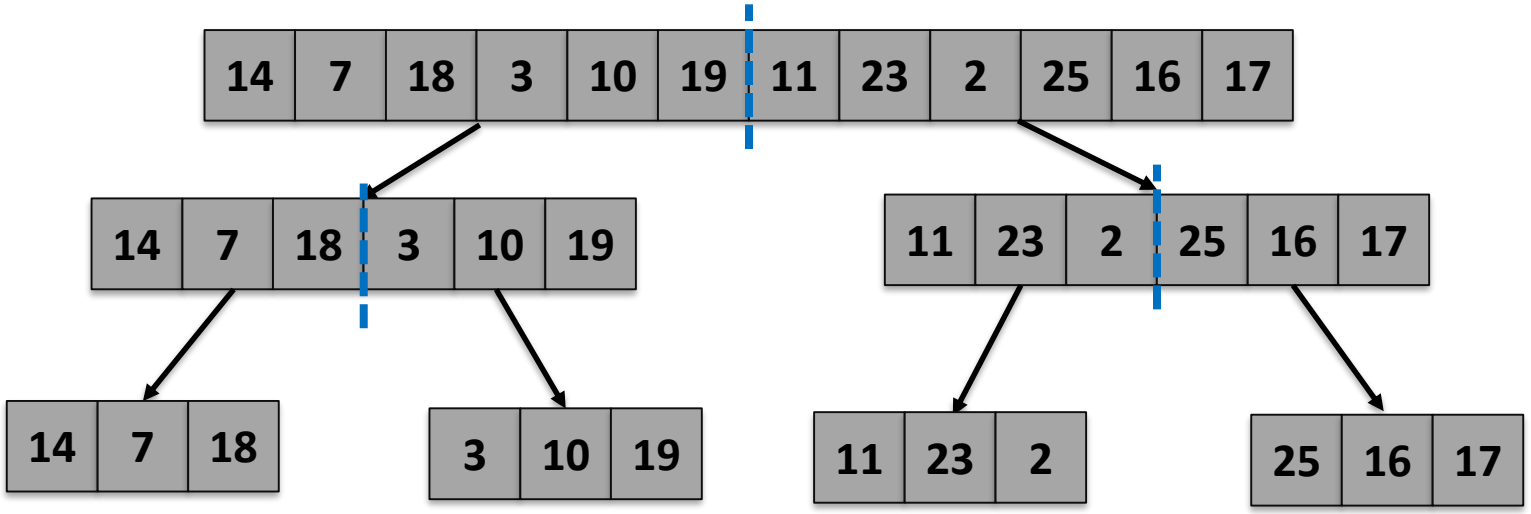
Example

Divide



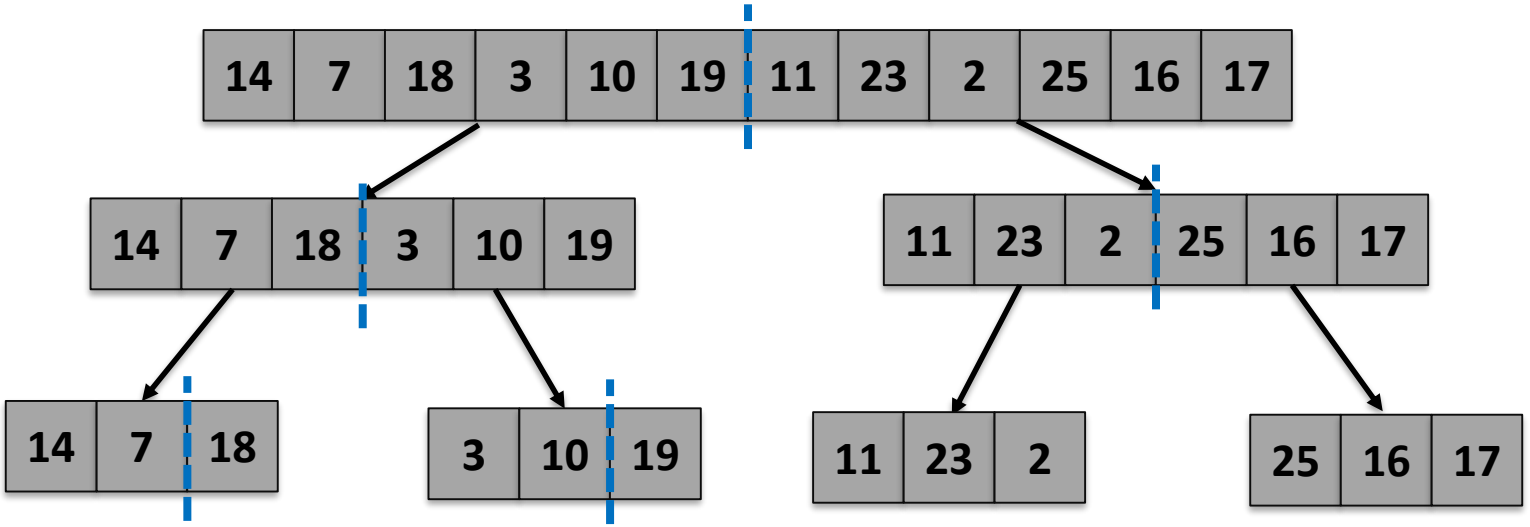
Example

Divide



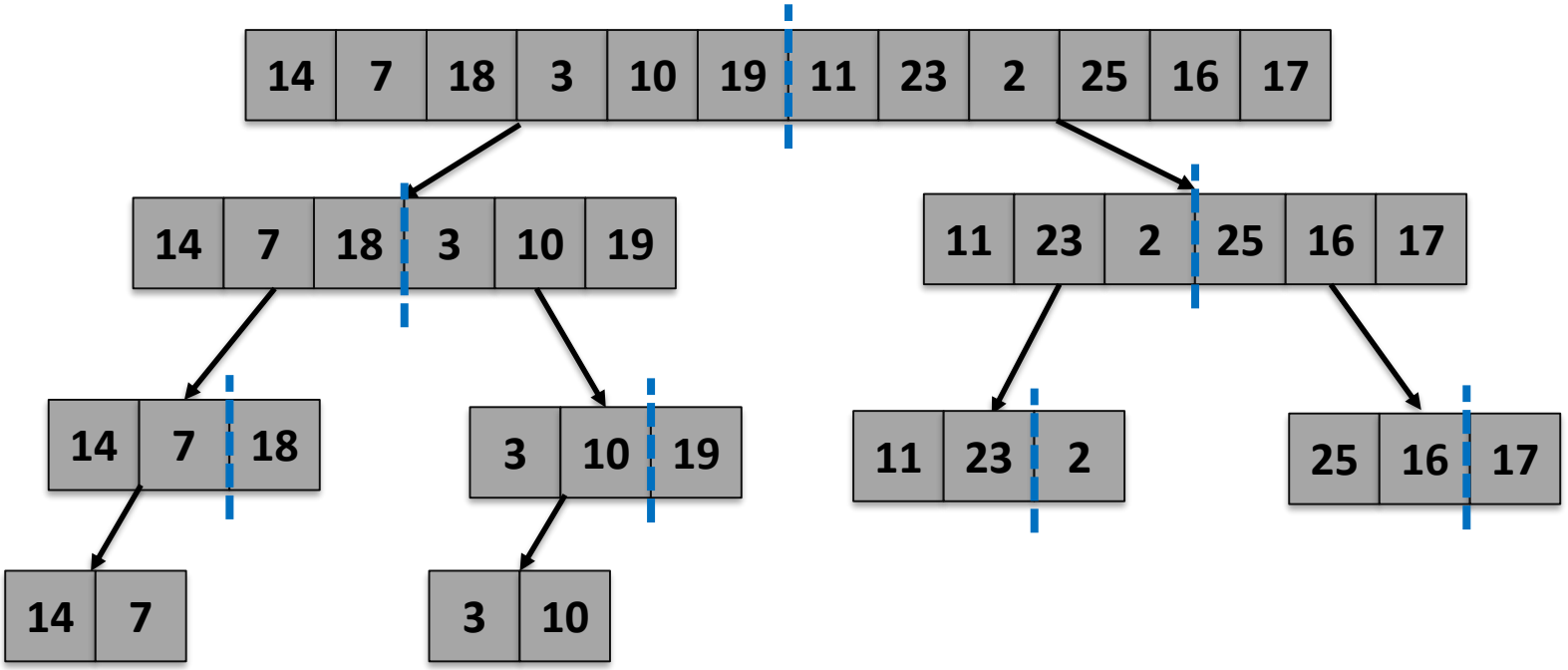
Example

Divide



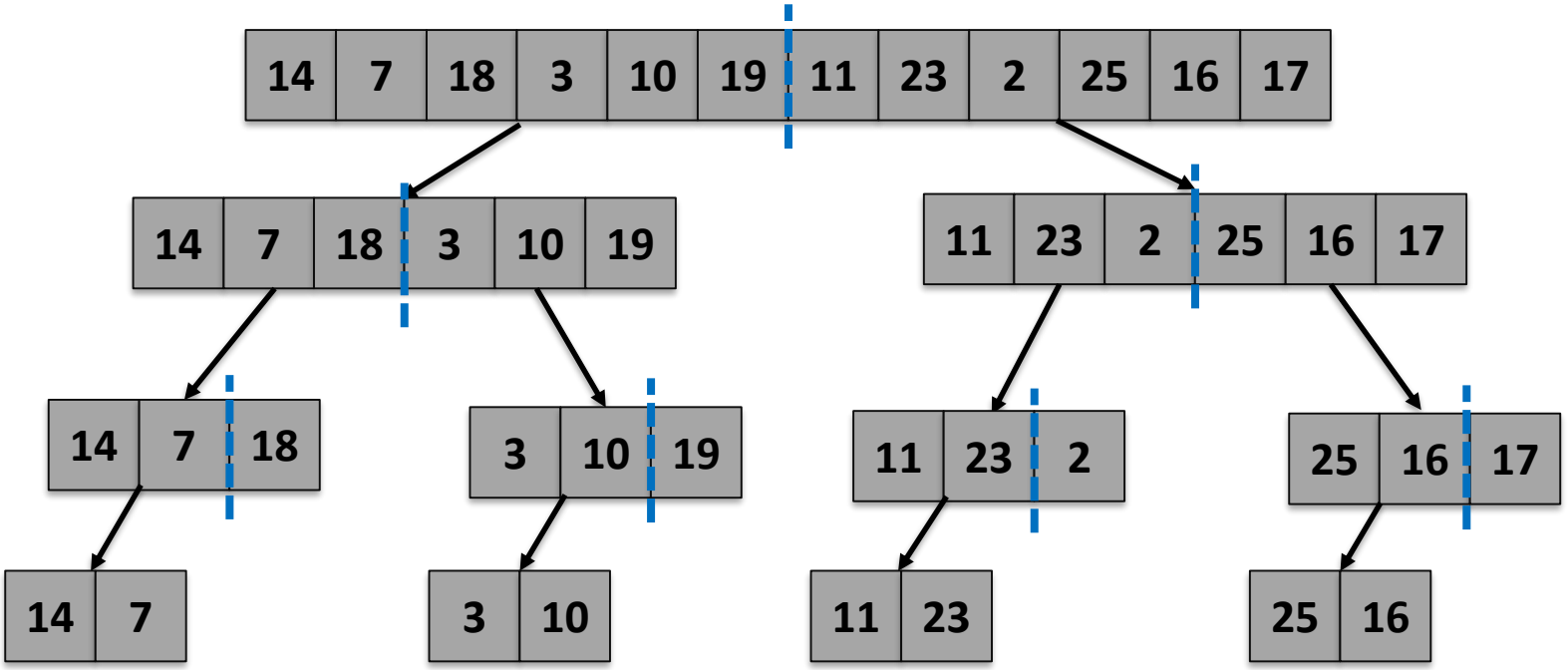
Example

Divide



Example

Divide

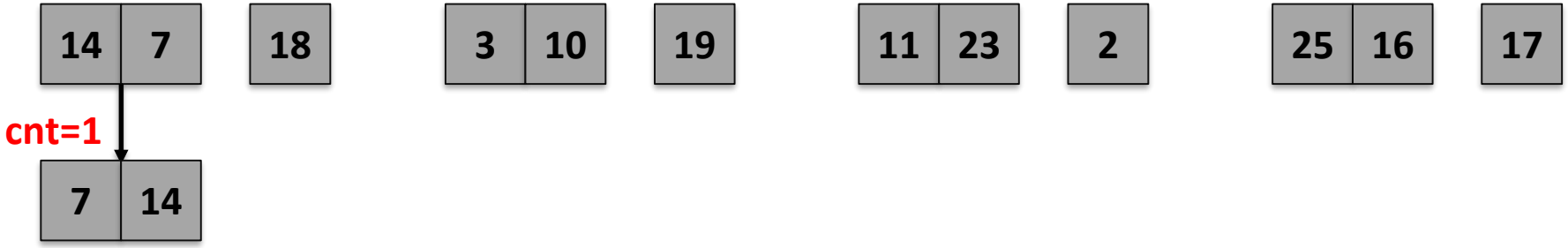


Example

Conquer

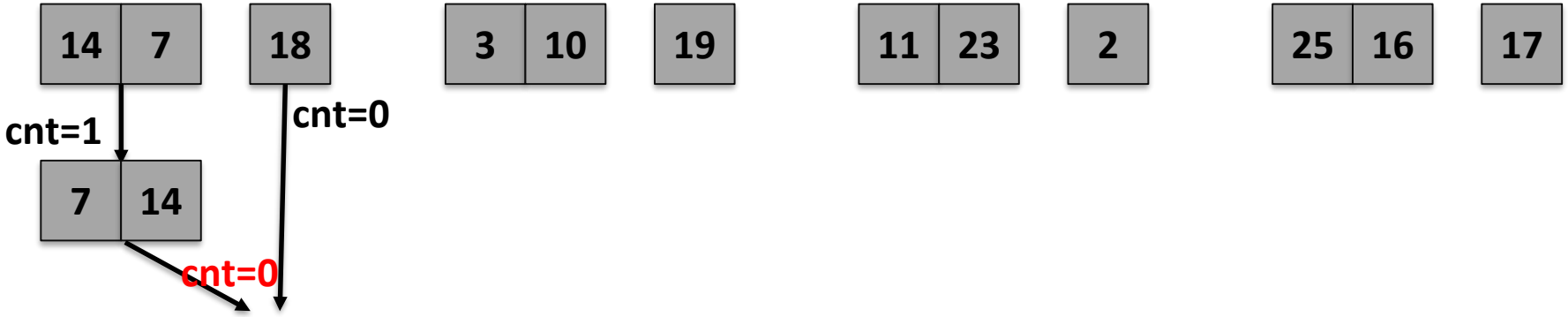
Example

Conquer



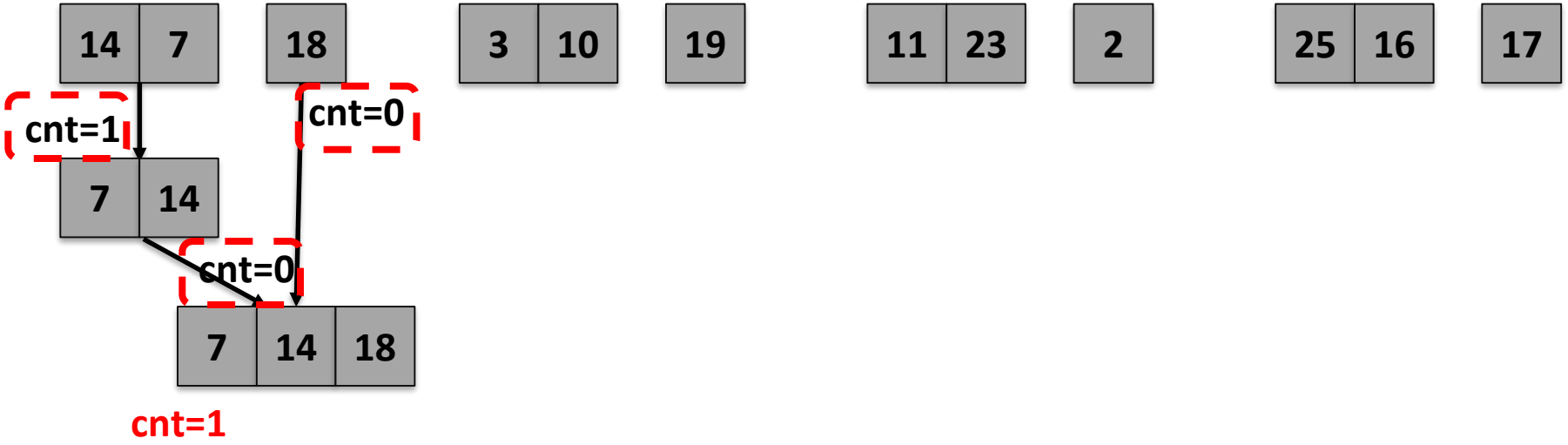
Example

Conquer



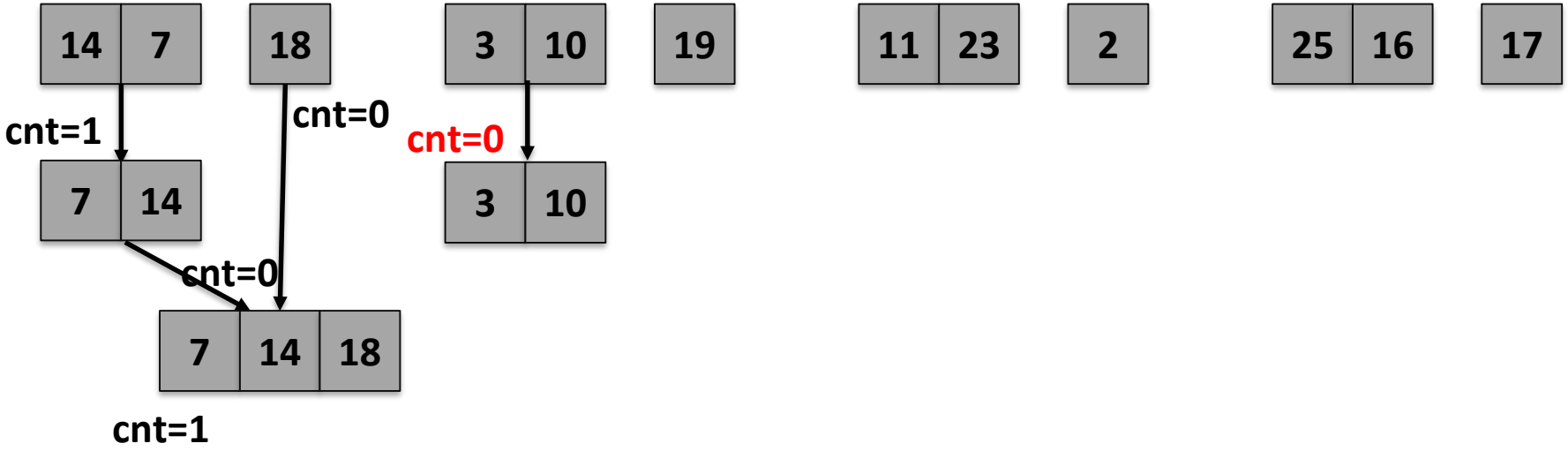
Example

Conquer



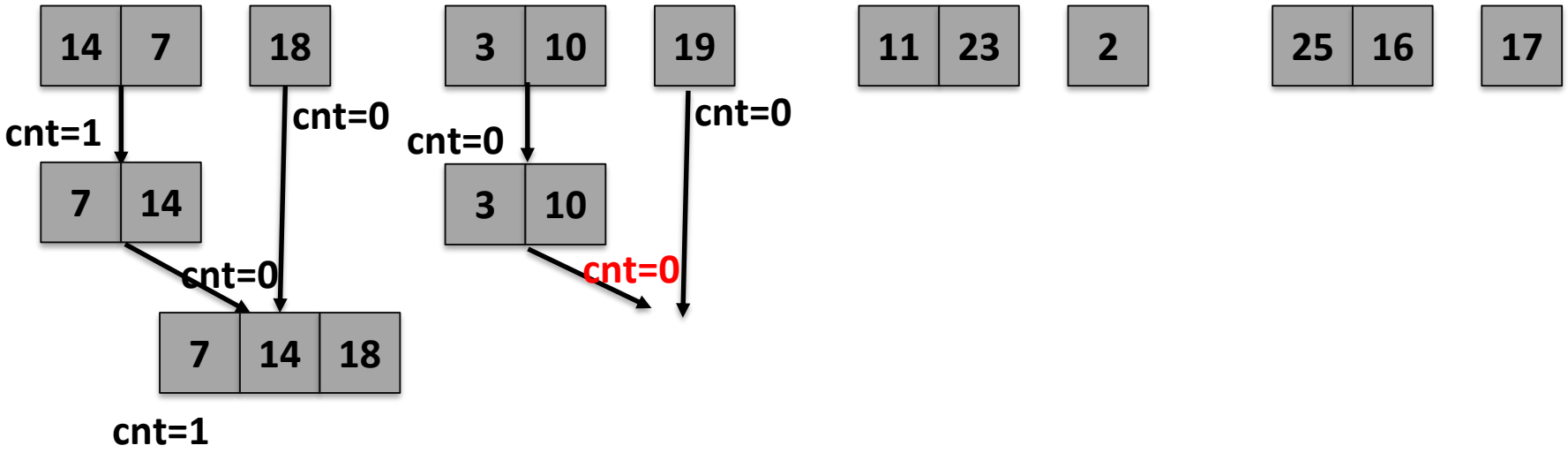
Example

Conquer



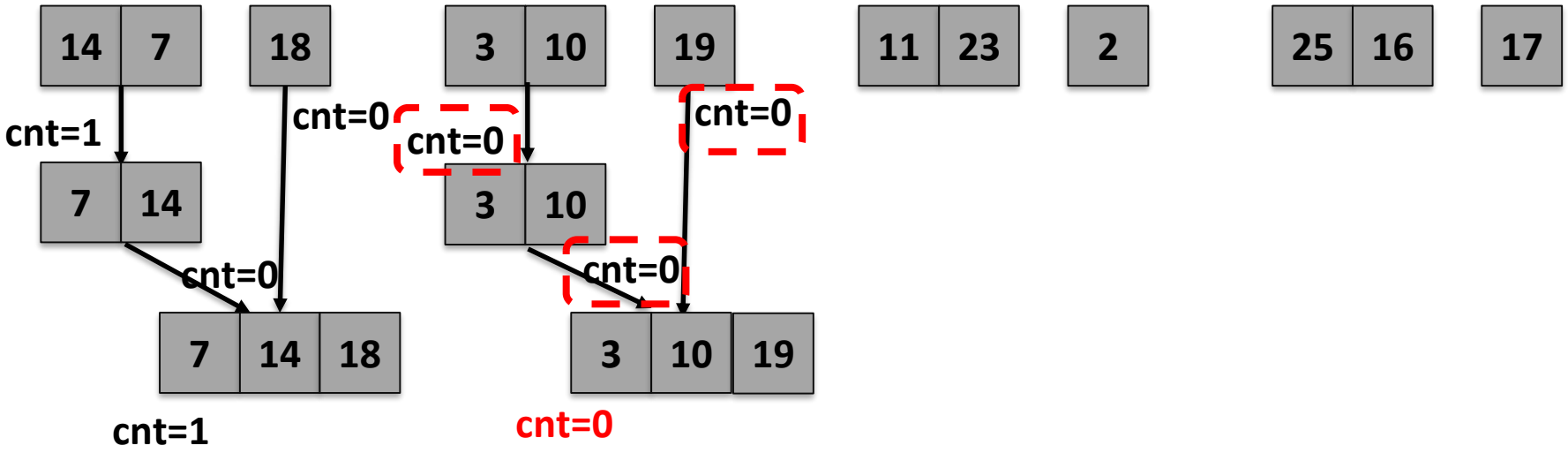
Example

Conquer



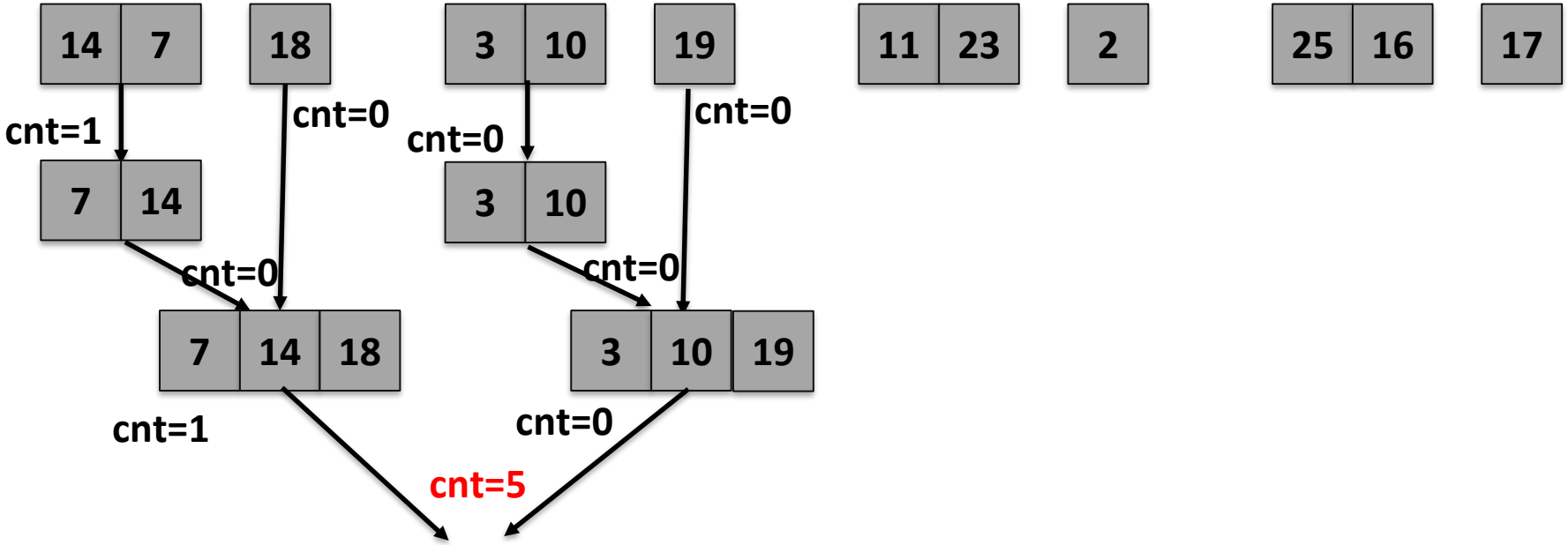
Example

Conquer



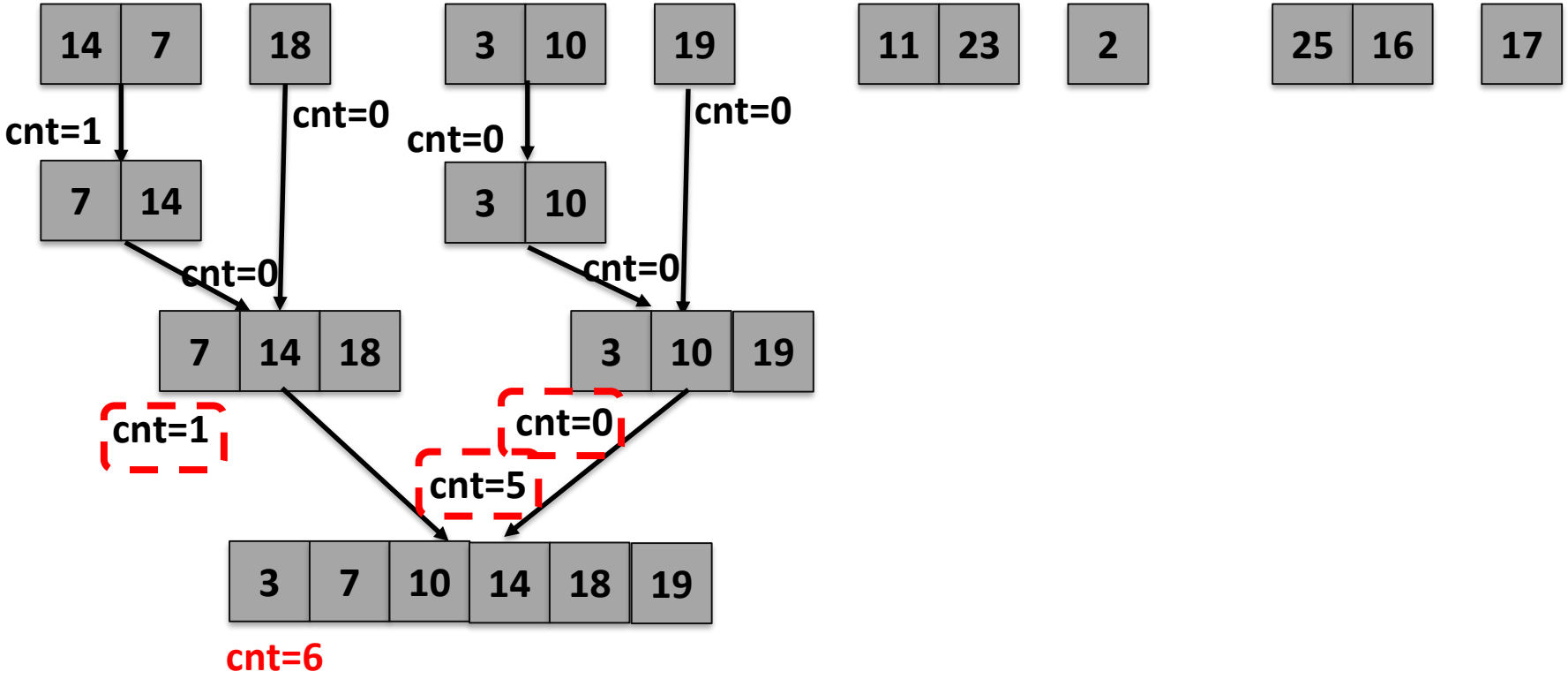
Example

Conquer



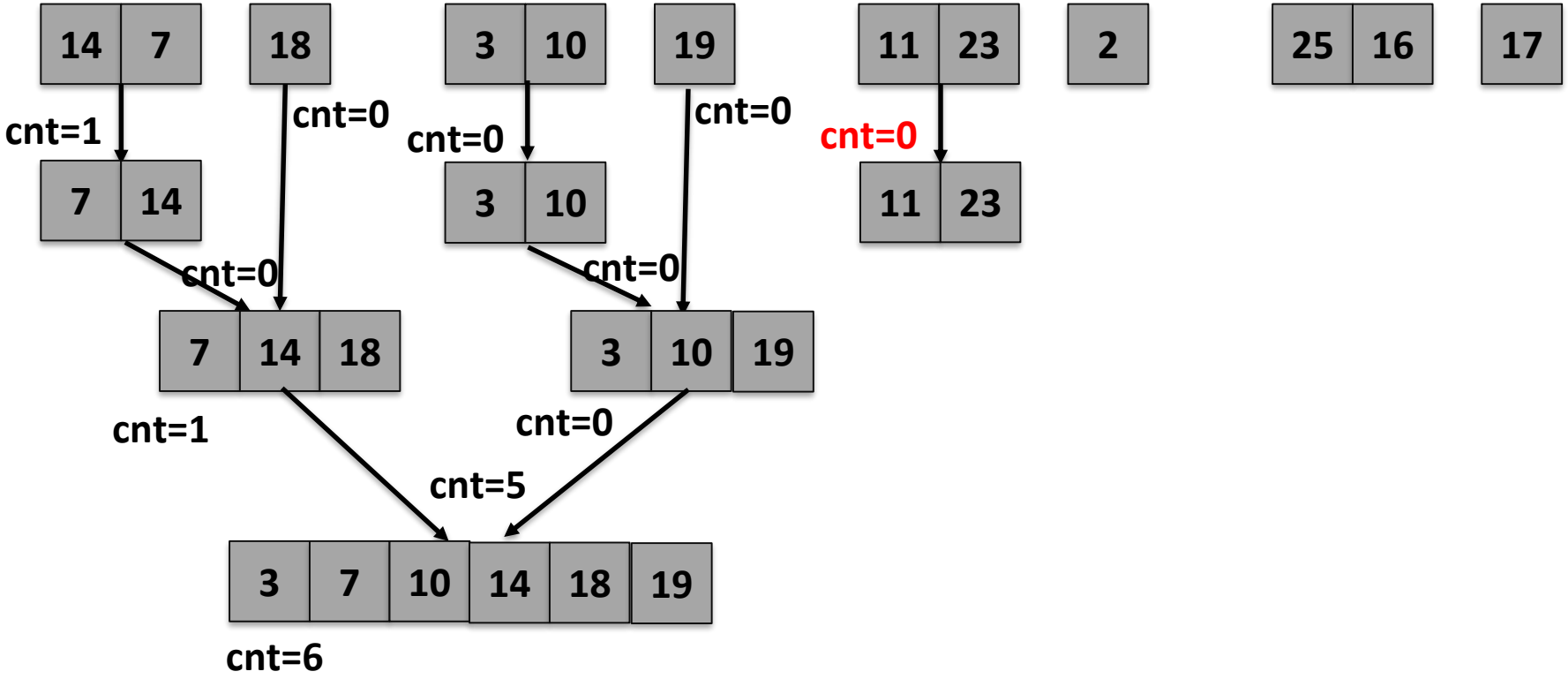
Example

Conquer



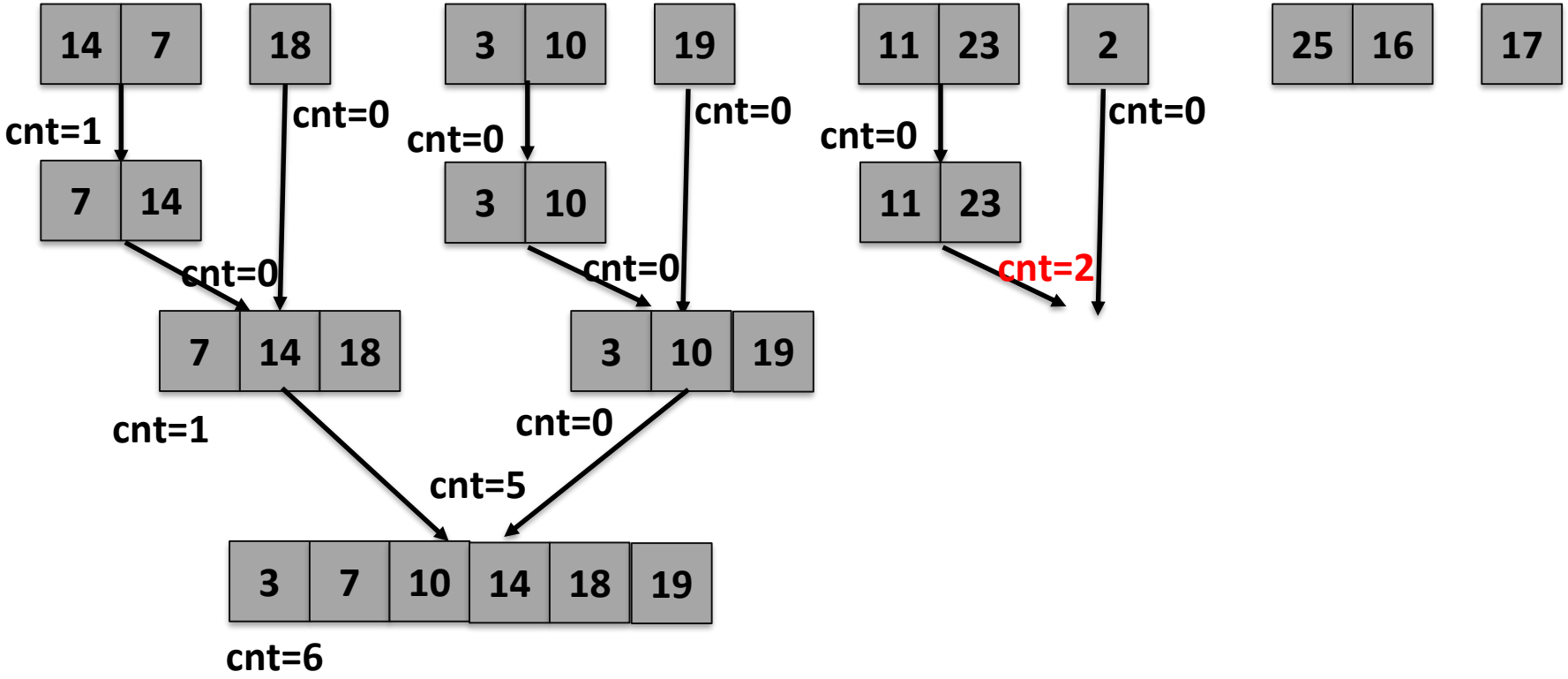
Example

Conquer



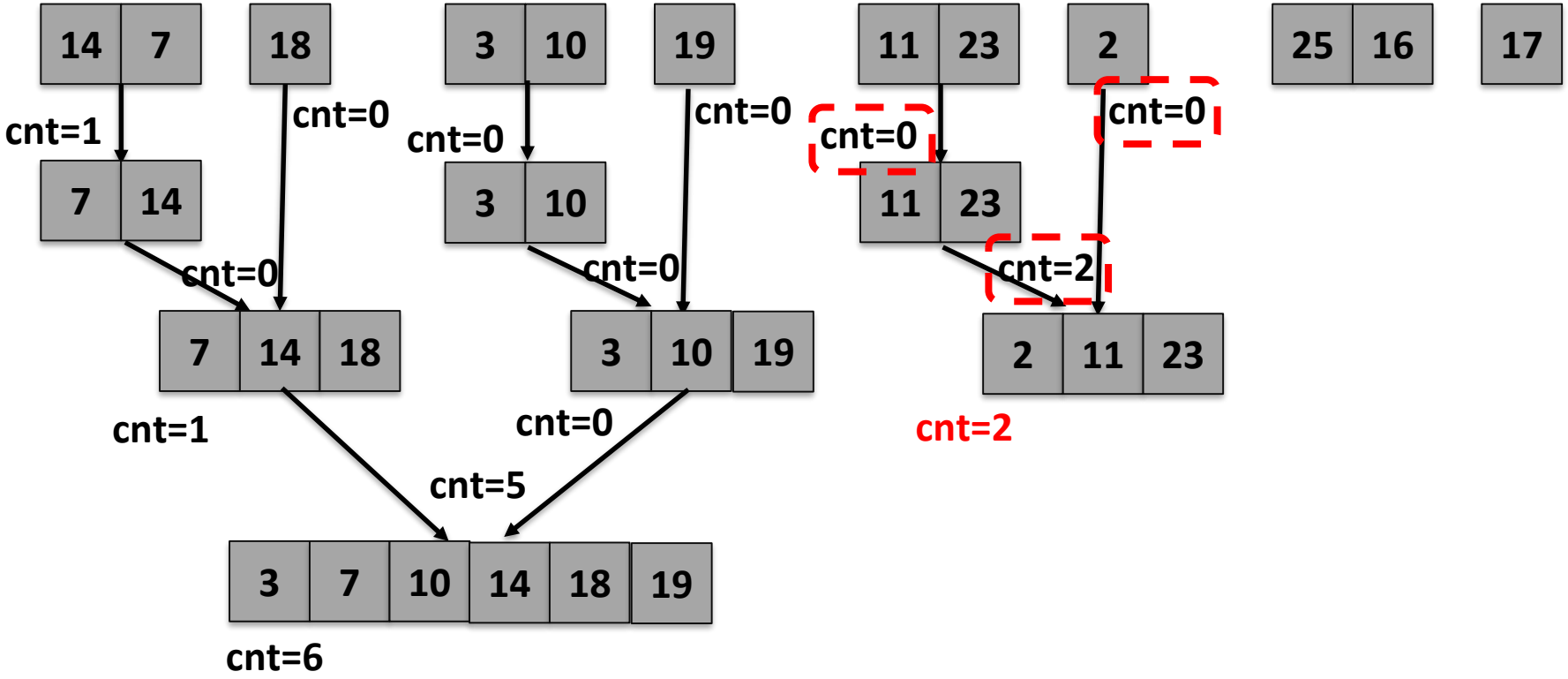
Example

Conquer



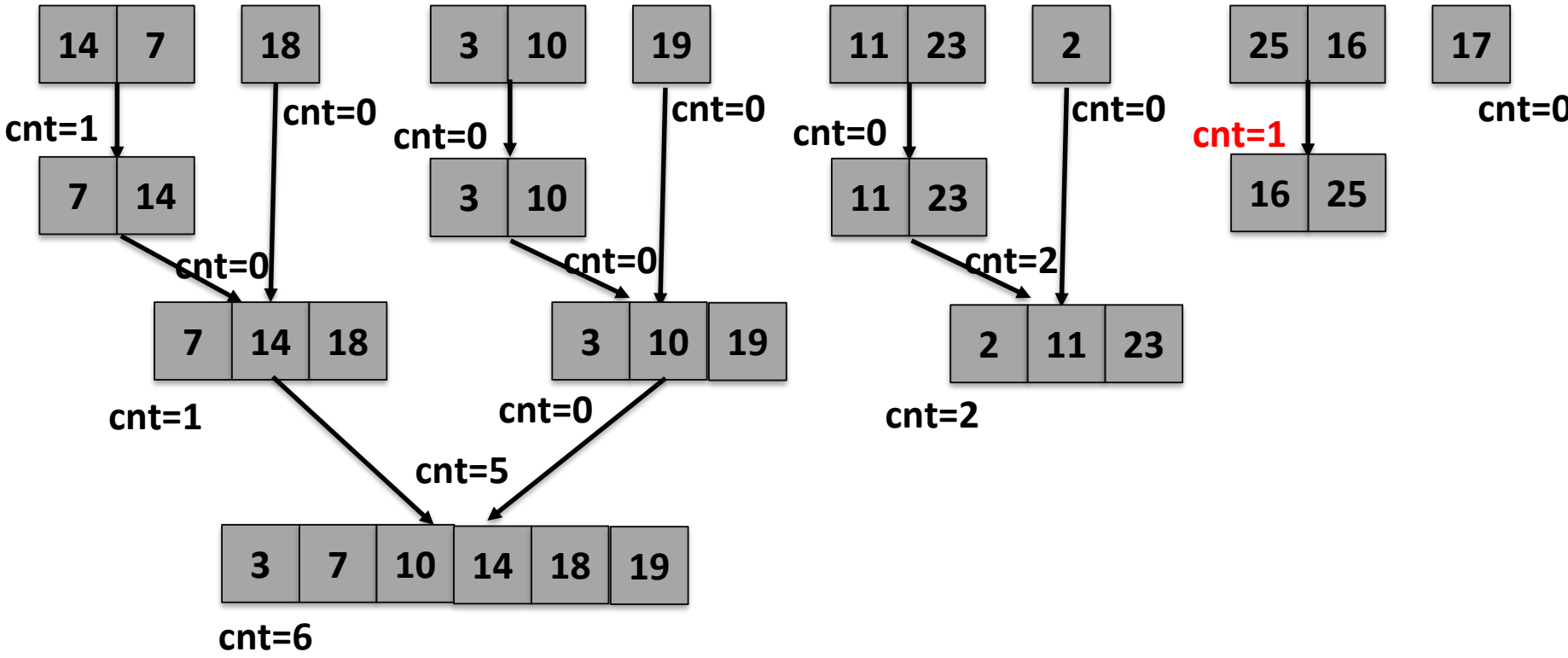
Example

Conquer



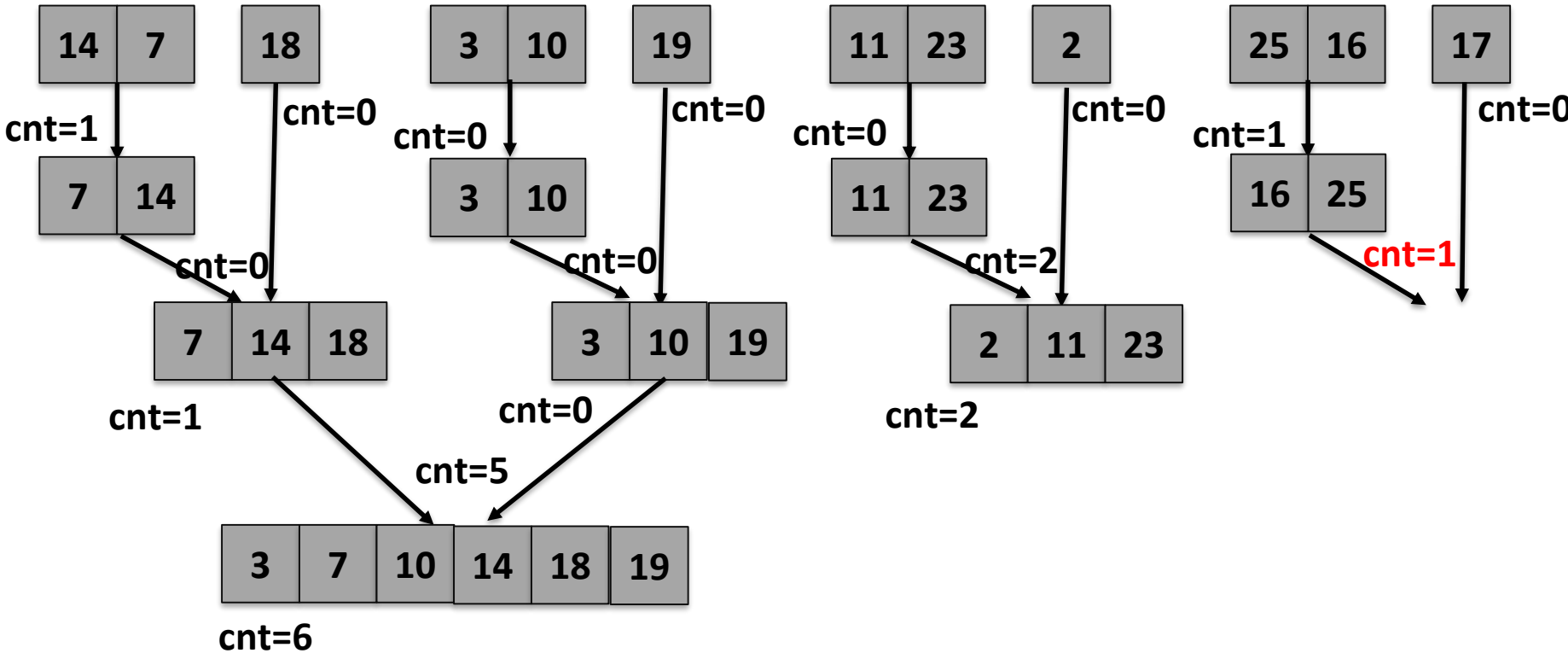
Example

Conquer



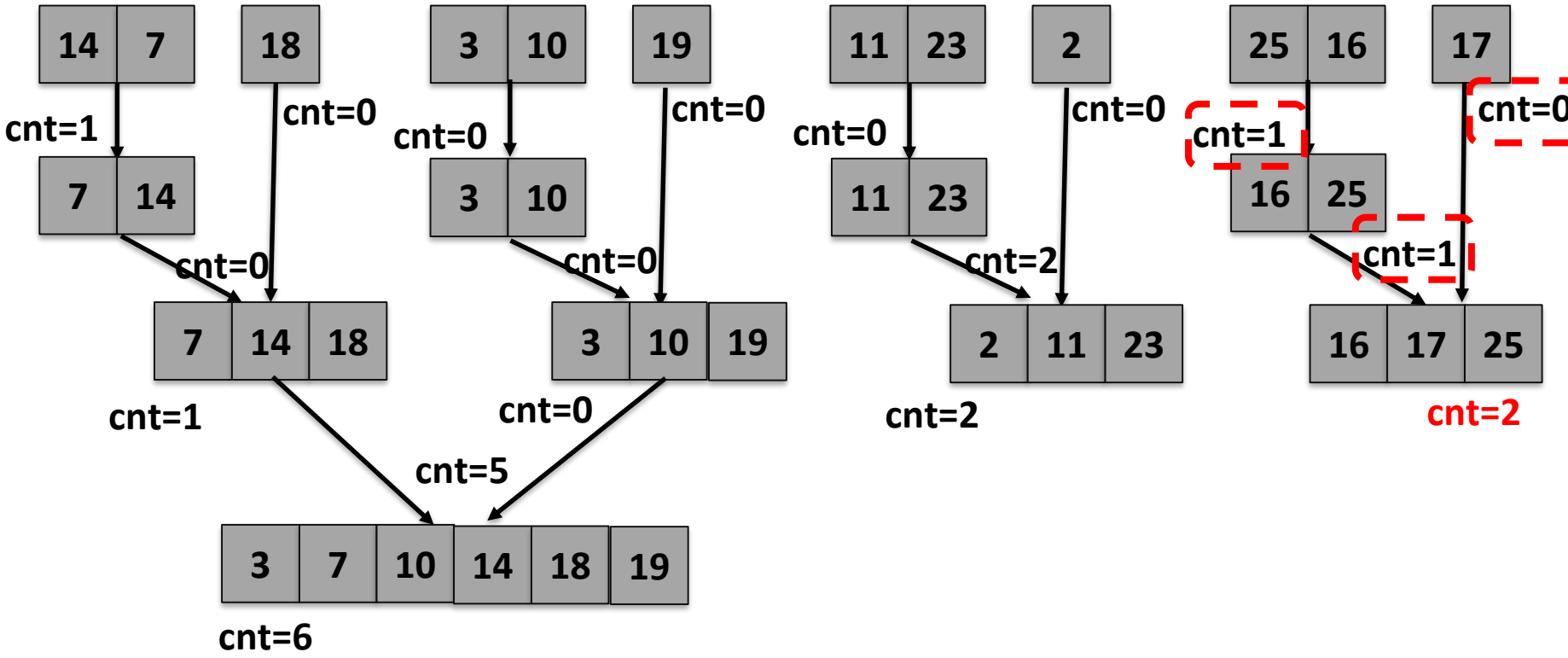
Example

Conquer



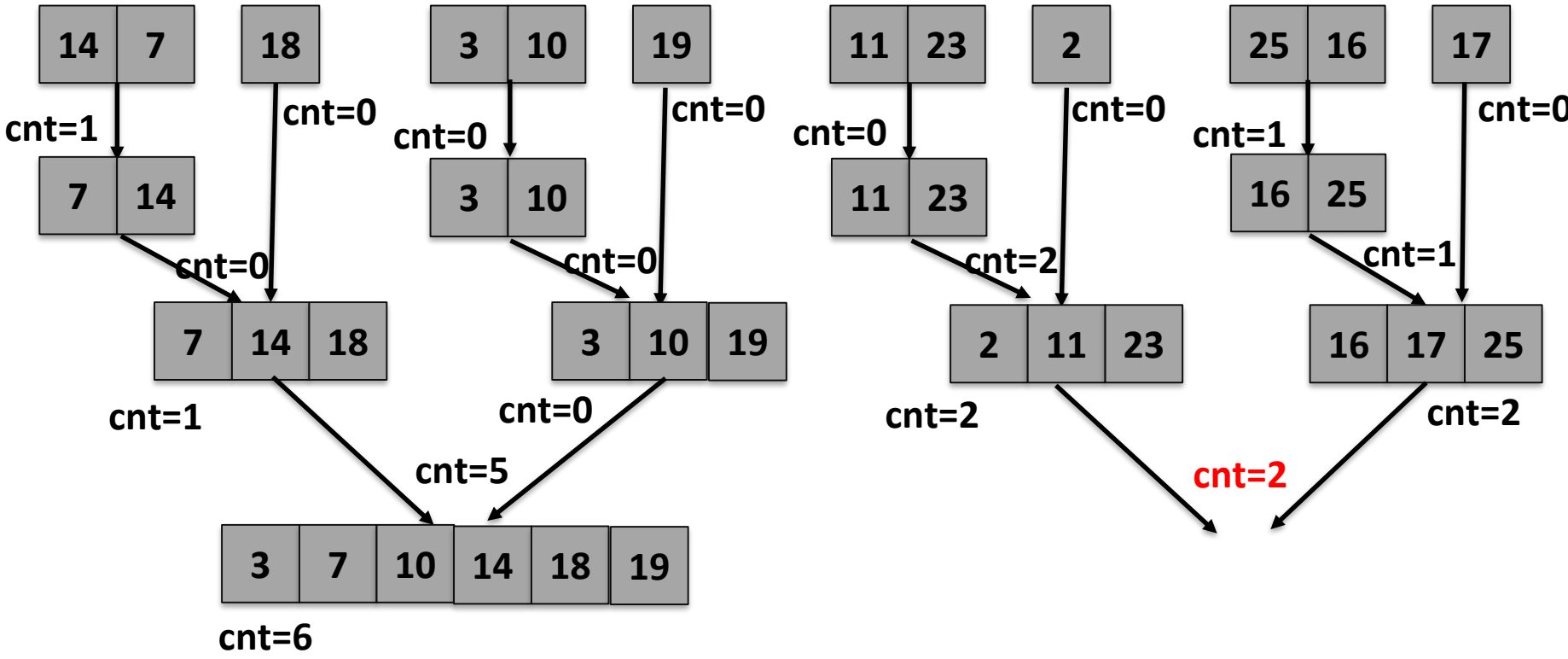
Example

Conquer



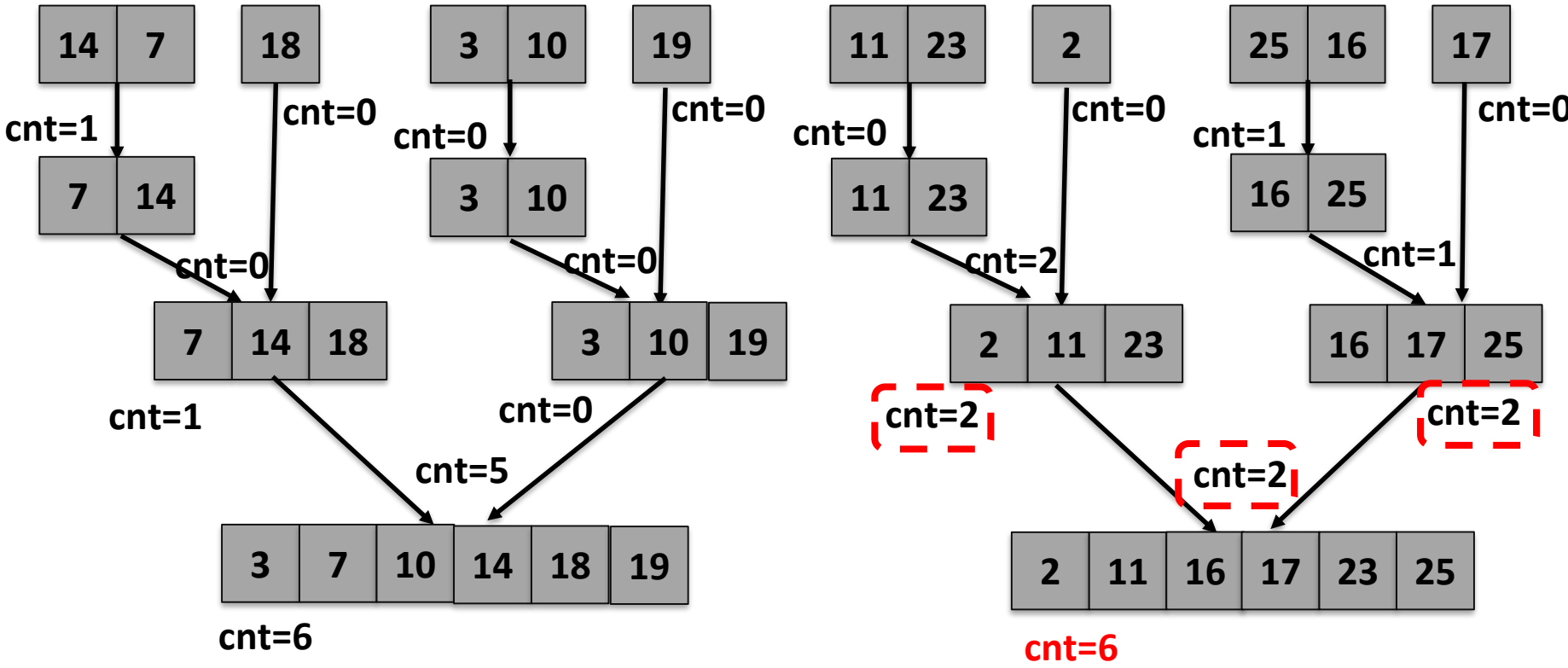
Example

Conquer



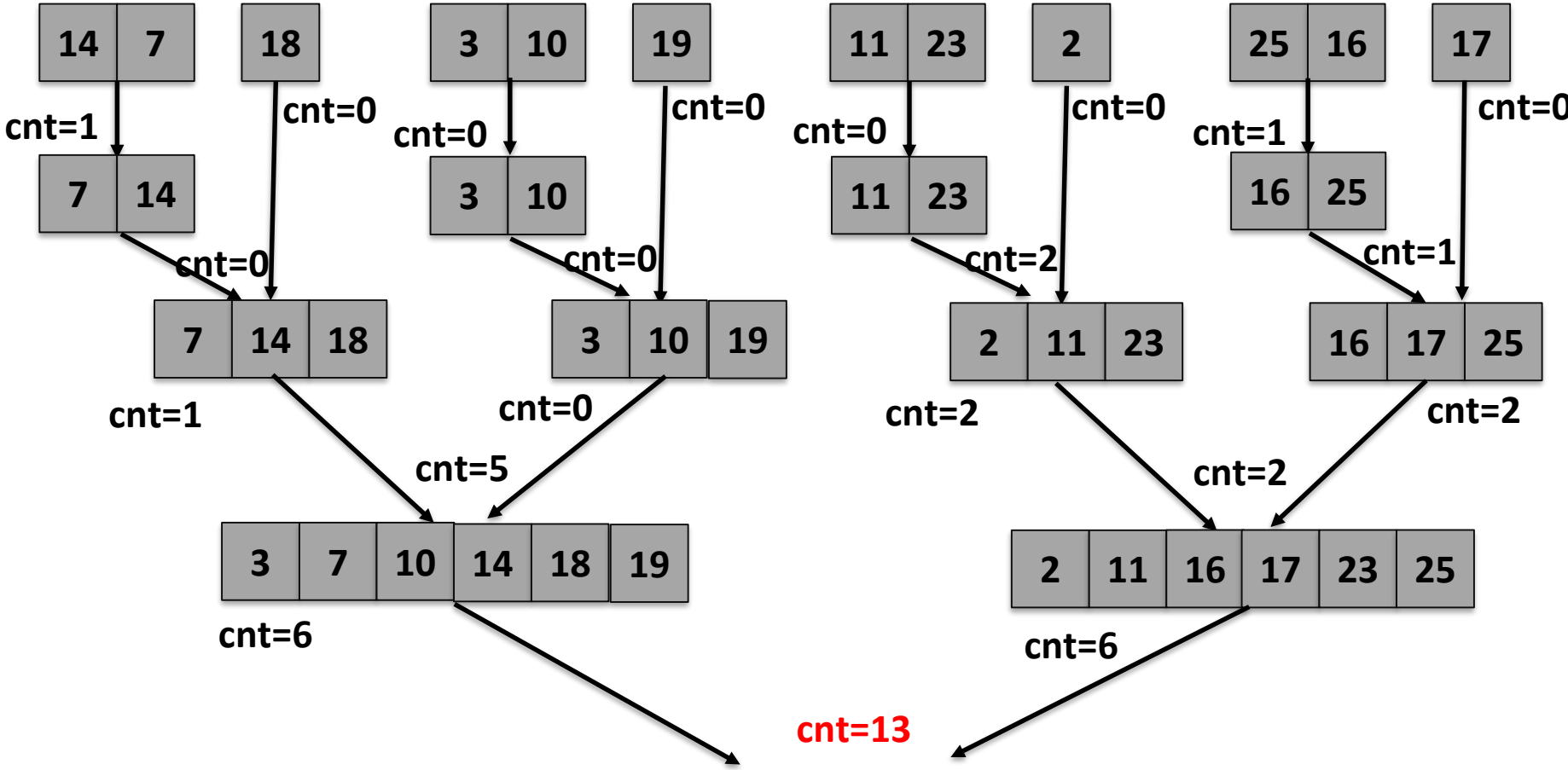
Example

Conquer



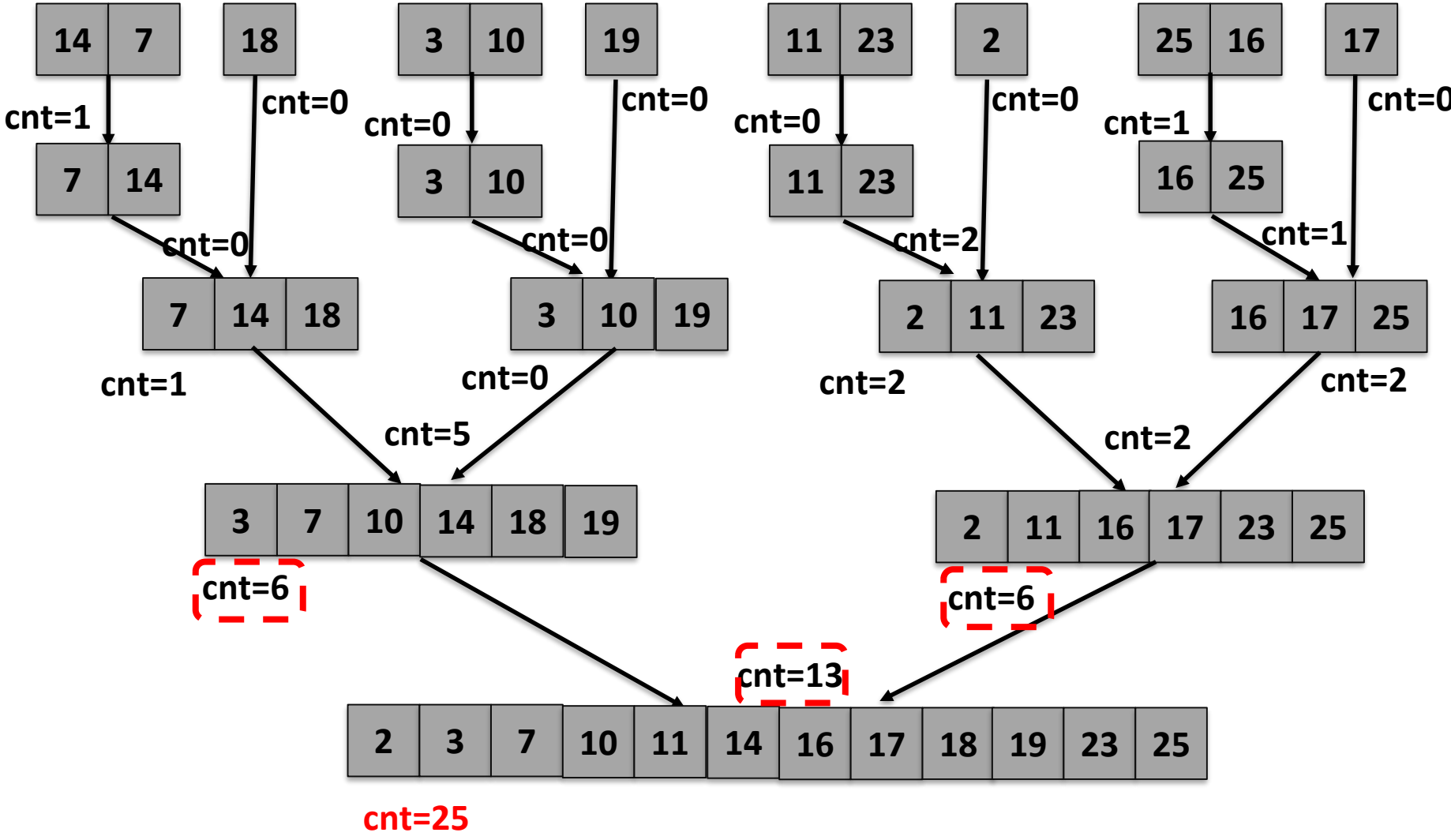
Example

Conquer



Example

Conquer



Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Analysis of the D&C Algorithm

Proposition. The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.

Analysis of the D&C Algorithm

Proposition. The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.

Proof. The worst-case running time T(n) satisfies the recurrence:

Analysis of the D&C Algorithm

Proposition. The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.

Proof. The worst-case running time T(n) satisfies the recurrence:

$$T(n) = \begin{cases} O(1), & if\ n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n) & otherwise \end{cases}$$

Master Theorem

If $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ for some constant a > 0, b > 1 and d ≥ 0, then

$$T(n) = \begin{cases} O\left(n^d\right), & if\ d > \log_b a \\ O\left(n^d \log n\right), & if\ d = \log_b a \\ O(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_m x^m \end{aligned}$$

Compute the product $A(x)B(x)$

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_m x^m \end{aligned}$$

Compute the product $A(x)B(x)$

Example

$$\begin{aligned} A(x) &= 1 + 2x + 3x^2 \\ B(x) &= 3 + 2x + 2x^2 \\ A(x)B(x) &= 3 + 8x + 15x^2 + 10x^3 + 6x^4 \end{aligned}$$

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_m x^m \end{aligned}$$

Compute the product $A(x)B(x)$

Example

$$\begin{aligned} A(x) &= 1 + 2x + 3x^2 \\ B(x) &= 3 + 2x + 2x^2 \\ A(x)B(x) &= 3 + 8x + 15x^2 + 10x^3 + 6x^4 \end{aligned}$$

- Assume that the coefficients a_i and b_i are stored in arrays A[0...n] and B[0...m]

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_m x^m \end{aligned}$$

Compute the product $A(x)B(x)$

Example

$$\begin{aligned} A(x) &= 1 + 2x + 3x^2 \\ B(x) &= 3 + 2x + 2x^2 \\ A(x)B(x) &= 3 + 8x + 15x^2 + 10x^3 + 6x^4 \end{aligned}$$

- Assume that the coefficients a_i and b_i are stored in arrays A[0...n] and B[0...m]
- Cost: number of scalar multiplications and additions

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

Then

- $c_k = \sum_{0 \le i \le n, 0 \le j \le m, i+j=k} a_i b_j$, for all $0 \le k \le m + n$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

Then

- $c_k = \sum_{0 \le i \le n, 0 \le j \le m, i+j=k} a_i b_j$, for all $0 \le k \le m + n$

Definition

The vector $(c_0, c_1, \ldots, c_{m+n})$ is the convolution of the vectors $(a_0, a_1, \ldots, a_n)$ and $(b_0, b_1, \ldots, b_m)$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

Then

- $c_k = \sum_{0 \le i \le n, 0 \le j \le m, i+j=k} a_i b_j$, for all $0 \le k \le m + n$

Definition

The vector $(c_0, c_1, \ldots, c_{m+n})$ is the convolution of the vectors $(a_0, a_1, \ldots, a_n)$ and $(b_0, b_1, \ldots, b_m)$

- We need to calculate convolutions. This is a major problem in digital signal processing

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Direct (Brute Force) Approach

To ease analysis, assume n = m.

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \, , for\ all\ 0 \le k \le 2n$$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , } for\ all\ 0 \le k \le 2n$$

Direct approach:

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , } for\ all\ 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , } for\ all\ 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: O( )

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , for all } 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , for all } 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$
- Total number of additions: O( )

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , for all } 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$
- Total number of additions: $O(n^2)$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , for all } 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$
- Total number of additions: $O(n^2)$
- Complexity: O( )

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \text{ , for all } 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$
- Total number of additions: $O(n^2)$
- Complexity: $O(n^2)$

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) =$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) +$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) =$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) +$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}}$$
$$+$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} +$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^n$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^n$$

The original problem (of size n) is divided into 4 problems of input size n/2

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3x^7 - 6x^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3x^7 - 6x^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3x^7 - 6x^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$
$$A_0(x)B_0(x) = 2 + 9x + 10x^2$$
$$A_1(x)B_1(x) = 6 + 11x + 19x^2 + 3x^3 - 6x^4$$
$$A_0(x)B_1(x) = 4 + 16x + 27x^2 + 30x^3$$
$$A_1(x)B_0(x) = 3 + 7x + x^2 - 2x^3$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3x^7 - 6x^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$
$$A_0(x)B_0(x) = 2 + 9x + 10x^2$$
$$A_1(x)B_1(x) = 6 + 11x + 19x^2 + 3x^3 - 6x^4$$
$$A_0(x)B_1(x) = 4 + 16x + 27x^2 + 30x^3$$
$$A_1(x)B_0(x) = 3 + 7x + x^2 - 2x^3$$
$$A_0(x)B_1(x) + A_1(x)B_0(x) = 7 + 23x + 28x^2 + 28x^3$$

$$A_0(x)B_0(x) + \big(A_0(x)B_1(x) + A_1(x)B_0(x)\big)x^2 + A_1(x)B_1(x)x^4$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3x^7 - 6x^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$
$$A_0(x)B_0(x) = 2 + 9x + 10x^2$$
$$A_1(x)B_1(x) = 6 + 11x + 19x^2 + 3x^3 - 6x^4$$
$$A_0(x)B_1(x) = 4 + 16x + 27x^2 + 30x^3$$
$$A_1(x)B_0(x) = 3 + 7x + x^2 - 2x^3$$
$$A_0(x)B_1(x) + A_1(x)B_0(x) = 7 + 23x + 28x^2 + 28x^3$$

$$A_0(x)B_0(x) + \big(A_0(x)B_1(x) + A_1(x)B_0(x)\big)x^2 + A_1(x)B_1(x)x^4$$
$$= 2 + 9x + 17x^2 + 23x^3 + 34x^4 + 39x^5 + 19x^6 + 3x^7 - 6x^8$$

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute
  $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A_1(x)B_1(x)}$

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute
  $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A_1(x)B_1(x)}$
  by recursively calling the algorithm 4 times

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute
  $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A_1(x)B_1(x)}$
  by recursively calling the algorithm 4 times

Combine

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute
  $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A_1(x)B_1(x)}$
  by recursively calling the algorithm 4 times

Combine

- Add the following four polynomials

$$A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^{n}$$

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute
  $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A_1(x)B_1(x)}$
  by recursively calling the algorithm 4 times

Combine

- Add the following four polynomials

$$A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^{n}$$

- Takes O( ) operations

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute
  $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A_1(x)B_1(x)}$
  by recursively calling the algorithm 4 times

Combine

- Add the following four polynomials

$$A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^{n}$$

- Takes O(n) operations

The First Divide-and-Conquer Algorithm

PolyMulti1$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;

The First Divide-and-Conquer Algorithm

PolyMulti1$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;
$U(x) \leftarrow$PolyMulti1$(A_0(x), B_0(x))$;//$T(n/2)$
$V(x) \leftarrow$PolyMulti1$(A_0(x), B_1(x))$;//$T(n/2)$
$W(x) \leftarrow$PolyMulti1$(A_1(x), B_0(x))$;//$T(n/2)$
$Z(x) \leftarrow$PolyMulti1$(A_1(x), B_1(x))$;//$T(n/2)$

The First Divide-and-Conquer Algorithm

PolyMulti1$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;
$U(x) \leftarrow$PolyMulti1$(A_0(x), B_0(x))$;//$T(n/2)$
$V(x) \leftarrow$PolyMulti1$(A_0(x), B_1(x))$;//$T(n/2)$
$W(x) \leftarrow$PolyMulti1$(A_1(x), B_0(x))$;//$T(n/2)$
$Z(x) \leftarrow$PolyMulti1$(A_1(x), B_1(x))$;//$T(n/2)$
return $(U(x) + [V(x) + W(x)]x^{\frac{n}{2}} + Z(x)x^n)$;//$O(n)$

The First Divide-and-Conquer Algorithm

PolyMulti1$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;
$U(x) \leftarrow$PolyMulti1$(A_0(x), B_0(x))$;//$T(n/2)$
$V(x) \leftarrow$PolyMulti1$(A_0(x), B_1(x))$;//$T(n/2)$
$W(x) \leftarrow$PolyMulti1$(A_1(x), B_0(x))$;//$T(n/2)$
$Z(x) \leftarrow$PolyMulti1$(A_1(x), B_1(x))$;//$T(n/2)$
return $(U(x) + [V(x) + W(x)]x^{\frac{n}{2}} + Z(x)x^n)$;//$O(n)$

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Analysis of Running Time

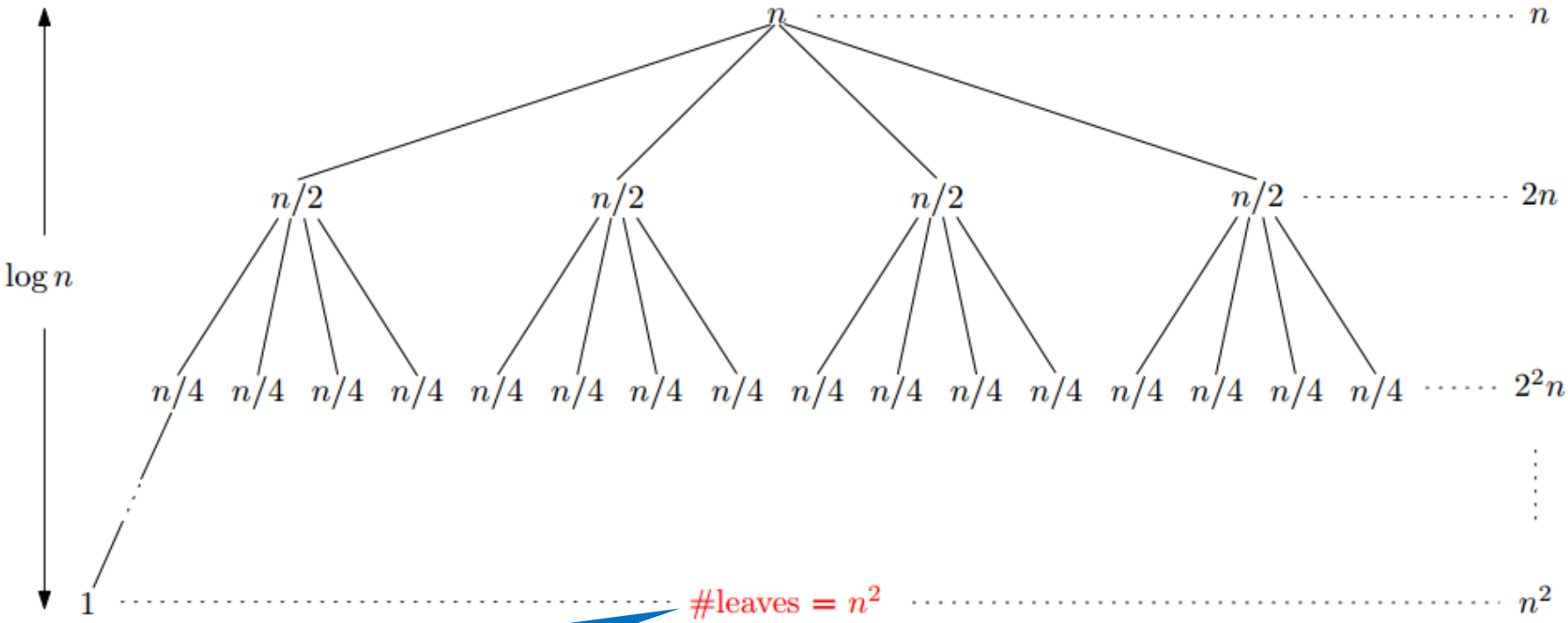
Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Analysis of Running Time

Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

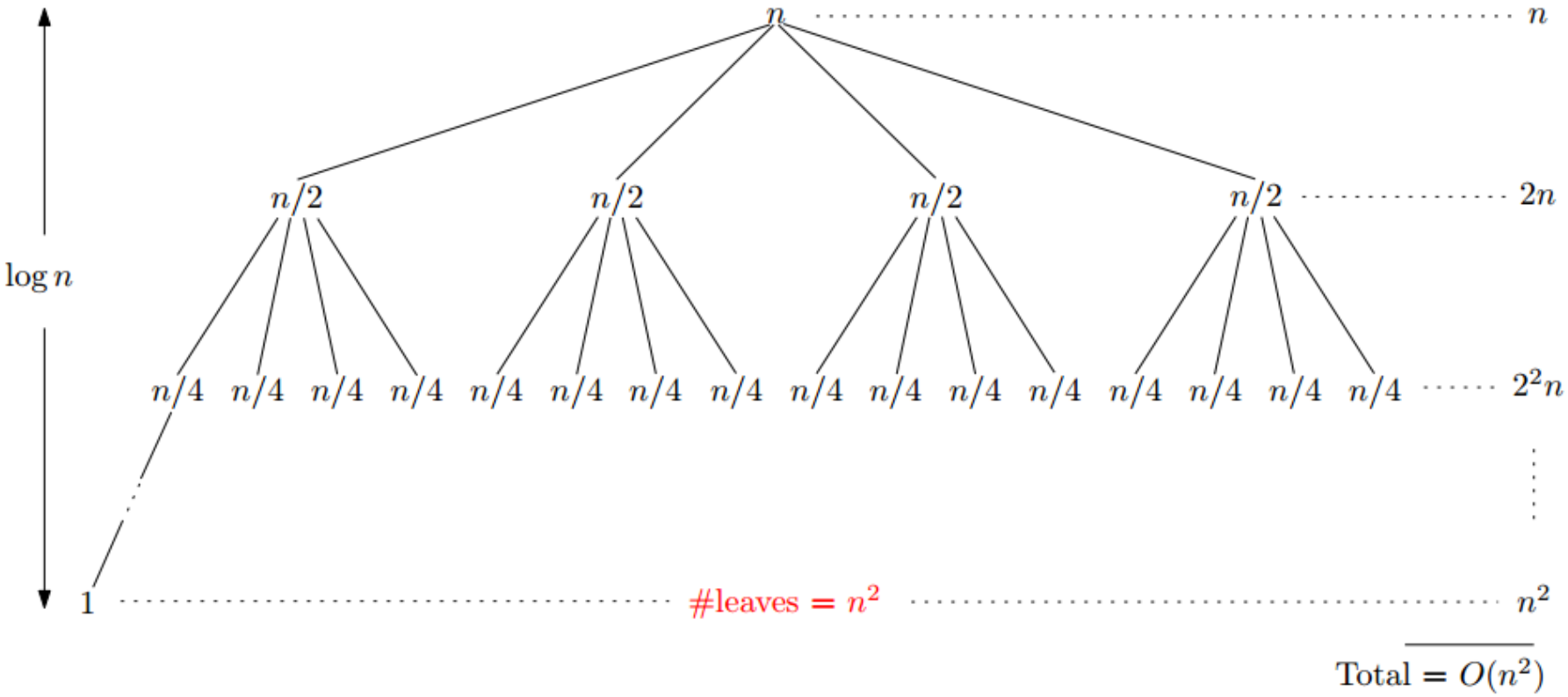


$$4^{\log n} = n^{\log 4} = n^2$$

Analysis of Running Time

Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

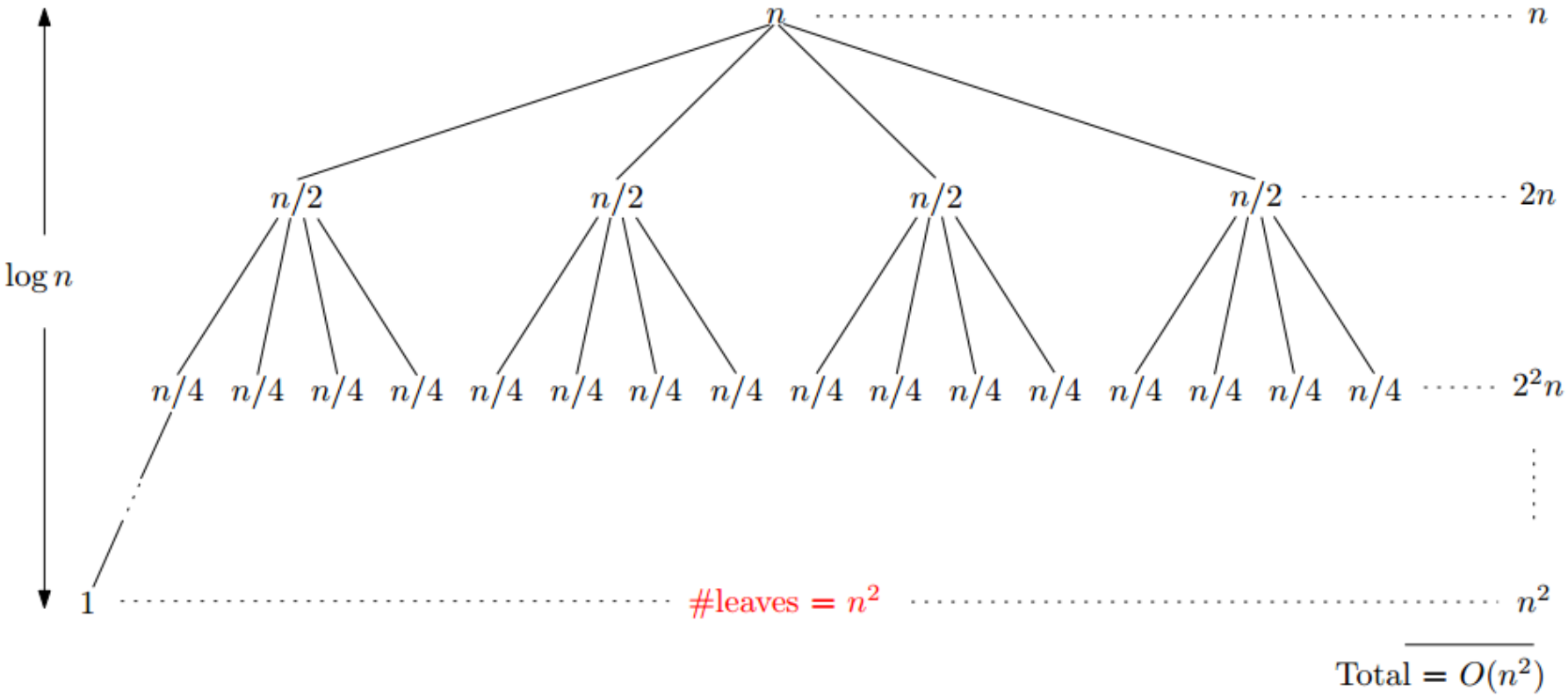


Same order as the brute force approach!

Analysis of Running Time

Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



Same order as the brute force approach! No improvement!

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0B_0 \\ Z &= A_1B_1 \end{aligned}$$

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0B_0 \\ Z &= A_1B_1 \end{aligned}$$

- *U and Z are what we originally wanted*

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0B_0 \\ Z &= A_1B_1 \end{aligned}$$

- *U and Z are what we originally wanted*
- $A_0B_1 + A_1B_0 =$

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0B_0 \\ Z &= A_1B_1 \end{aligned}$$

- *U and Z are what we originally wanted*
- $A_0B_1 + A_1B_0 = Y - U - Z$

The improved Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$

$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;

The improved Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2$(A_0(x) + A_1(x), B_0(x) + B_1(x))$;//$T(n/2)$
$U(x) \leftarrow$PolyMulti2$(A_0(x), B_0(x))$;//$T(n/2)$
$Z(x) \leftarrow$PolyMulti2$(A_1(x), B_1(x))$;//$T(n/2)$

The Second Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2$(A_0(x) + A_1(x), B_0(x) + B_1(x))$;//$T(n/2)$
$U(x) \leftarrow$PolyMulti2$(A_0(x), B_0(x))$;//$T(n/2)$
$Z(x) \leftarrow$PolyMulti2$(A_1(x), B_1(x))$;//$T(n/2)$
return $(U(x) + [Y(x) - U(x) - Z(x)]x^{\frac{n}{2}} + Z(x)x^{2\frac{n}{2}})$;//$O(n)$

The improved Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2$(A_0(x) + A_1(x), B_0(x) + B_1(x))$;//$T(n/2)$
$U(x) \leftarrow$PolyMulti2$(A_0(x), B_0(x))$;//$T(n/2)$
$Z(x) \leftarrow$PolyMulti2$(A_1(x), B_1(x))$;//$T(n/2)$
return $(U(x) + [Y(x) - U(x) - Z(x)]x^{\frac{n}{2}} + Z(x)x^{2\frac{n}{2}})$;//$O(n)$

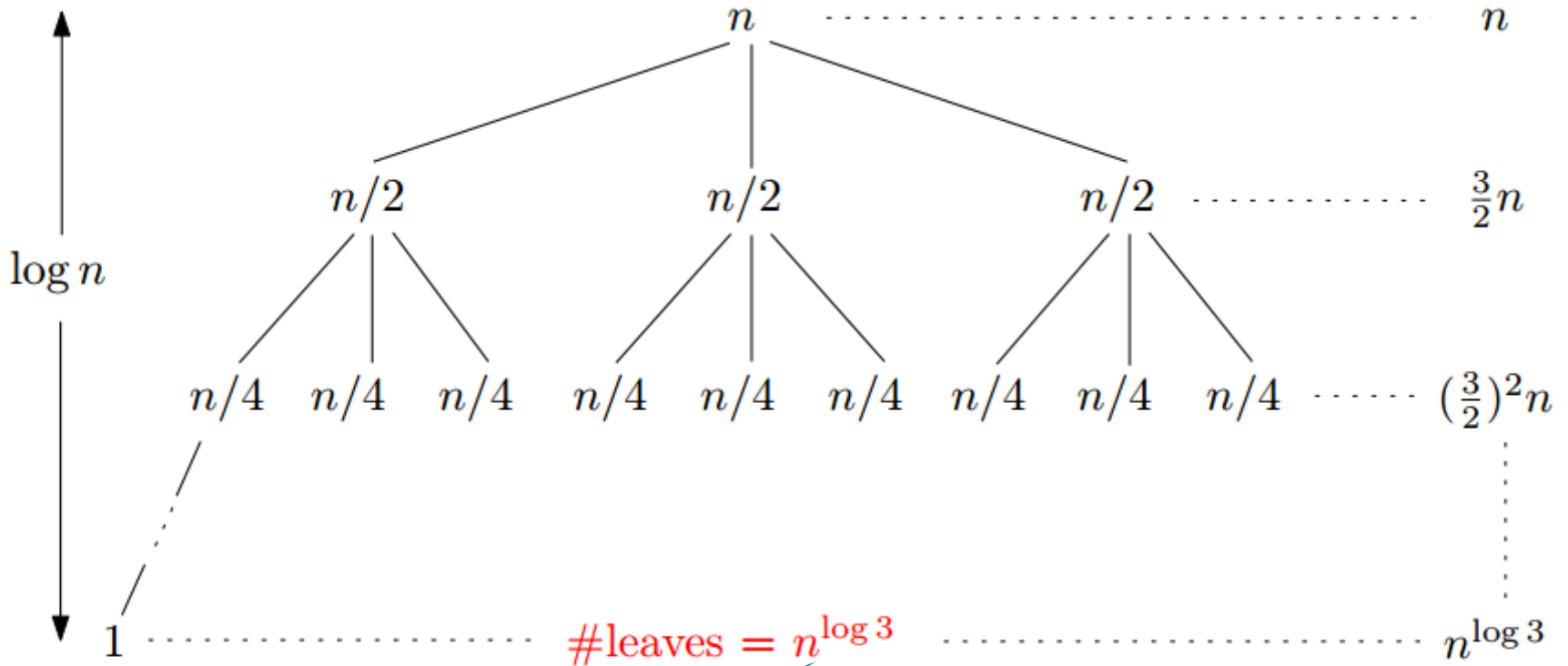
$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Running Time of the Improved Algorithm

$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Running Time of the Improved Algorithm

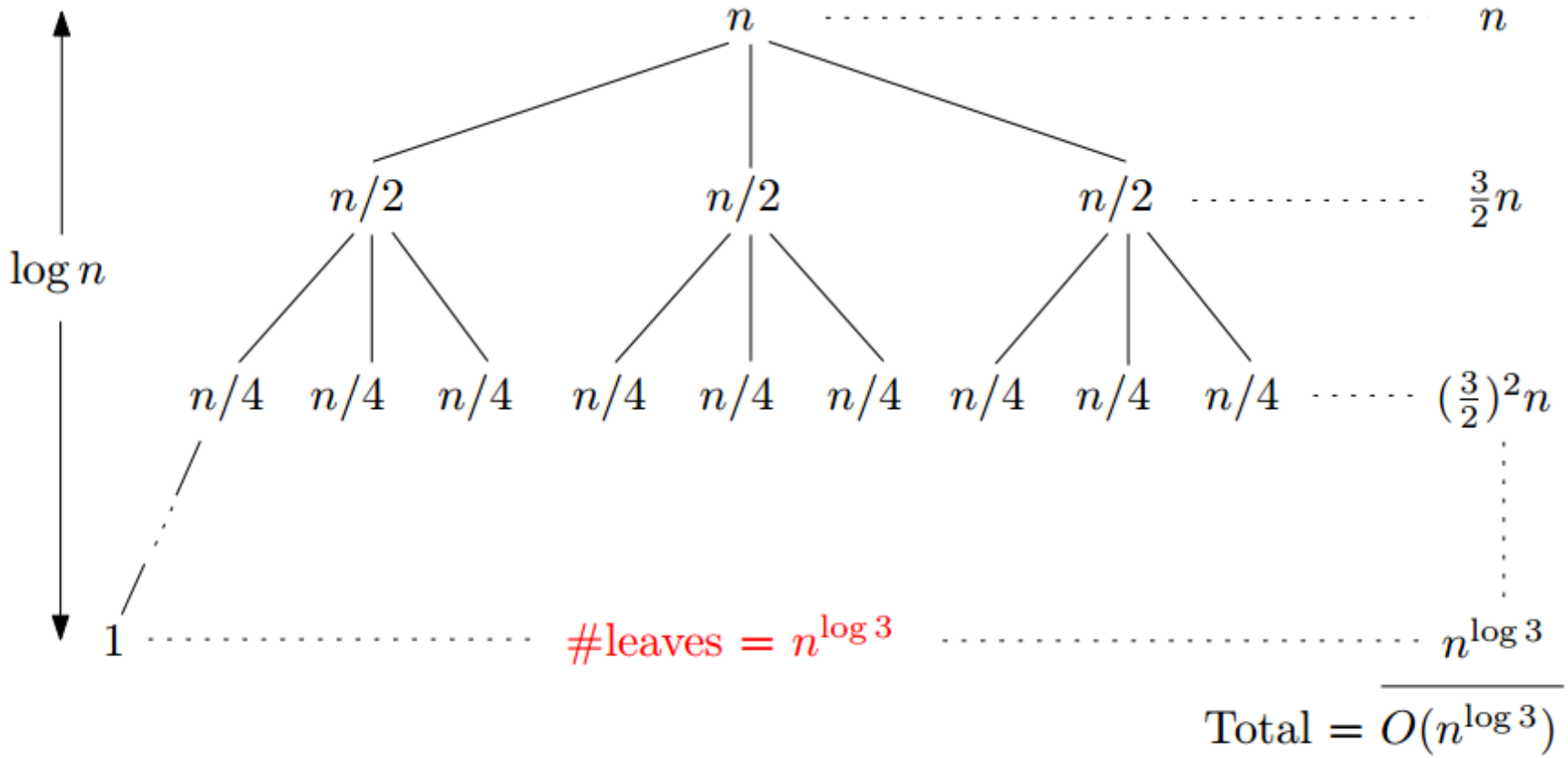
$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



$3^{logn} = n^{log3}$

Running Time of the Improved Algorithm

$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



The second method is much better!

Outline

- Review to Divide-and-Conquer Paradigm
- Counting Inversions Problem
  - Problem definition
  - A brute force algorithm
  - A divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force
- There is actually an O(n log n) solution to the polynomial multiplication problem

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force
- There is actually an O(n log n) solution to the polynomial multiplication problem
  - It involves using the Fast Fourier Transform algorithm as a subroutine
  - The FFT is another classic divide-and-conquer algorithm(check Chapt 30 in CLRS if interested)

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force
- There is actually an O(n log n) solution to the polynomial multiplication problem
  - It involves using the Fast Fourier Transform algorithm as a subroutine
  - The FFT is another classic divide-and-conquer algorithm(check Chapt 30 in CLRS if interested)
- The idea of using 3 multiplications instead of 4 is used in large-integer multiplications
  - A similar idea is the basis of the classic Strassen matrix multiplication algorithm (CLRS 4.2)