AI VIET NAM
@aivietnam.edu.vn

Algorithm design and analysis

Divide and Conquer

Nguyễn Quốc Thái

CONTENT

Review Recursion

Fibonacci Number

- Recursion: a function makes one or more calls to itself during execution
- Powerful for performing repetitive tasks

Base case
Avoid infinite recursion

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

fibonacci(4)
```

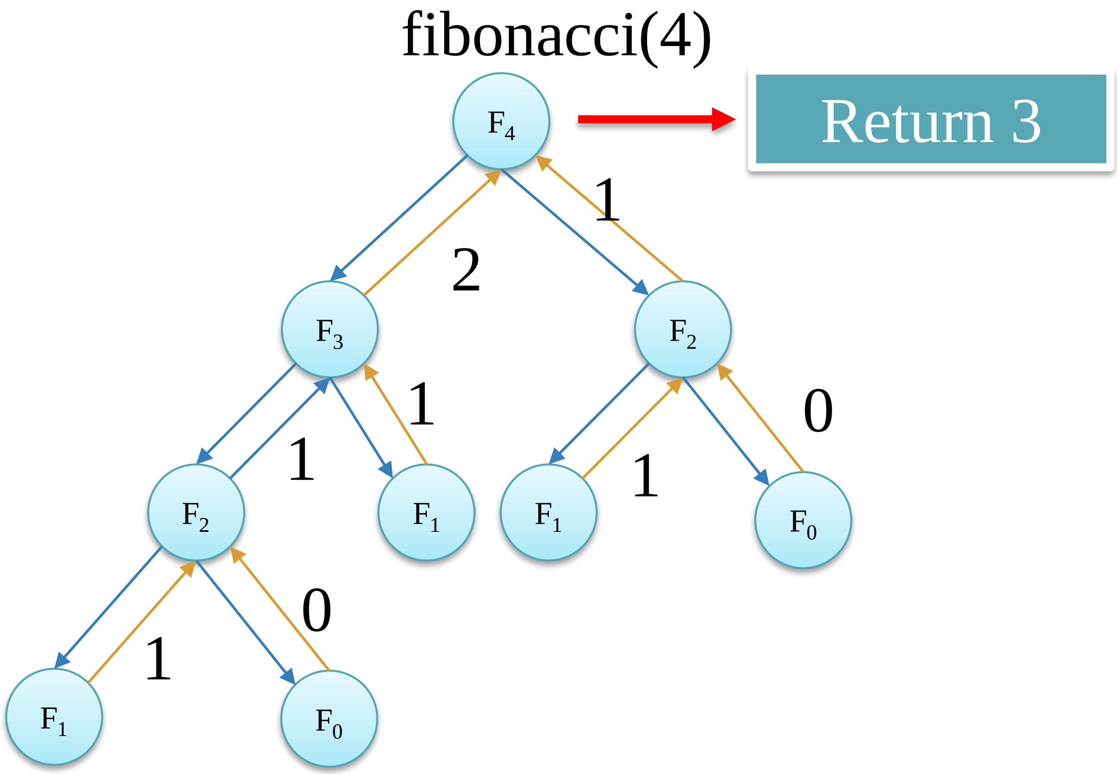
```
3
```

Call to itself

Smaller instance size

Review Recursion

Fibonacci Number




```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

fibonacci(4)
```

```
3
```

T(n) is $O(2^n)$

Sorting Problem

Sorting Problem

Input: a sequence of n number $<a_1, a_2, \ldots, a_n>$

Output: a permutation (reordering) $<a'_1, a'_2, \ldots, a'_n>$

such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

7	2	5	30	16	8	4	17

$\xrightarrow{\textbf{Sorting}}$

2	4	5	7	8	16	17	30

Sorting Problem

Sorting Problem

Selection Sort

```
def selection_sort(array):
    n = len(array)

    for step in range(n):
        min_idx = step

        for i in range(step + 1, n):

            if array[i] < array[min_idx]:
                min_idx = i

        array[step], array[min_idx] = array[min_idx], array[step]

    return array
```

Sorting Problem

Sorting Problem

Insertion Sort

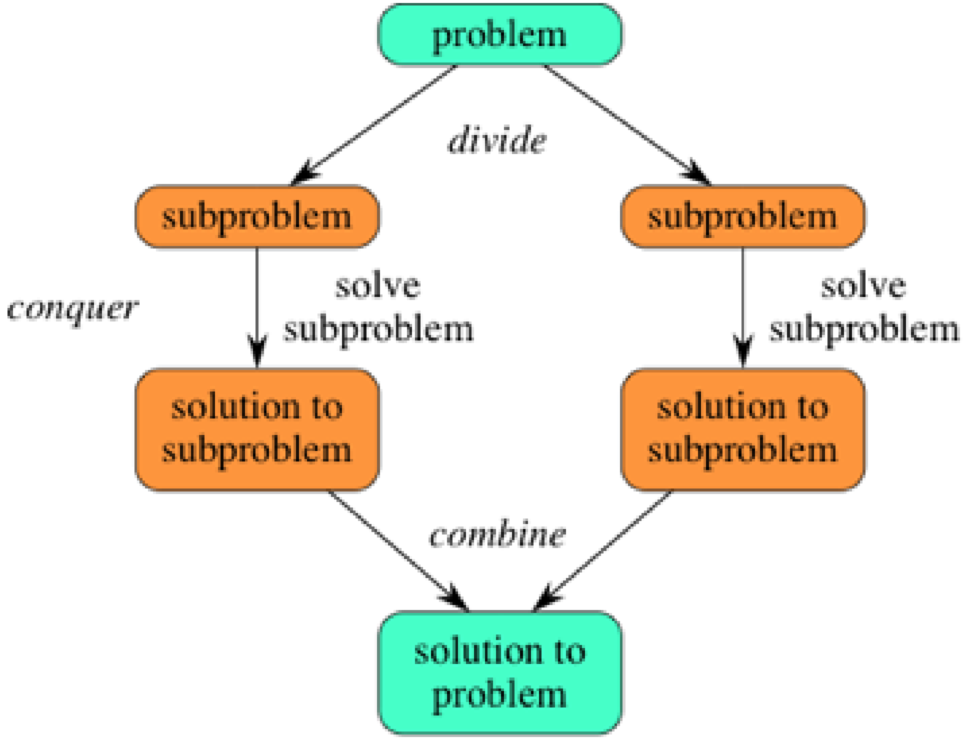
```
def insertion_sort(S):
    n = len(S)
    for step in range(1, n):
        key = S[step]
        i = step - 1
        while i >= 0 and key < S[i]:
            S[i + 1] = S[i]
            i = i - 1
        S[i + 1] = key
    return S
```

Divide-and-Conquer

- Use recursion in an algorithmic design
- Three steps:
  - **Divide** the problem into a number of subproblems: smaller instances of the same problem
  - **Conquer** the subproblems by solving recursively. If they are small enough, solve the subproblems as base cases.
  - **Combine** the solutions to the subproblems into the solution for the original problem

Divide-and-Conquer

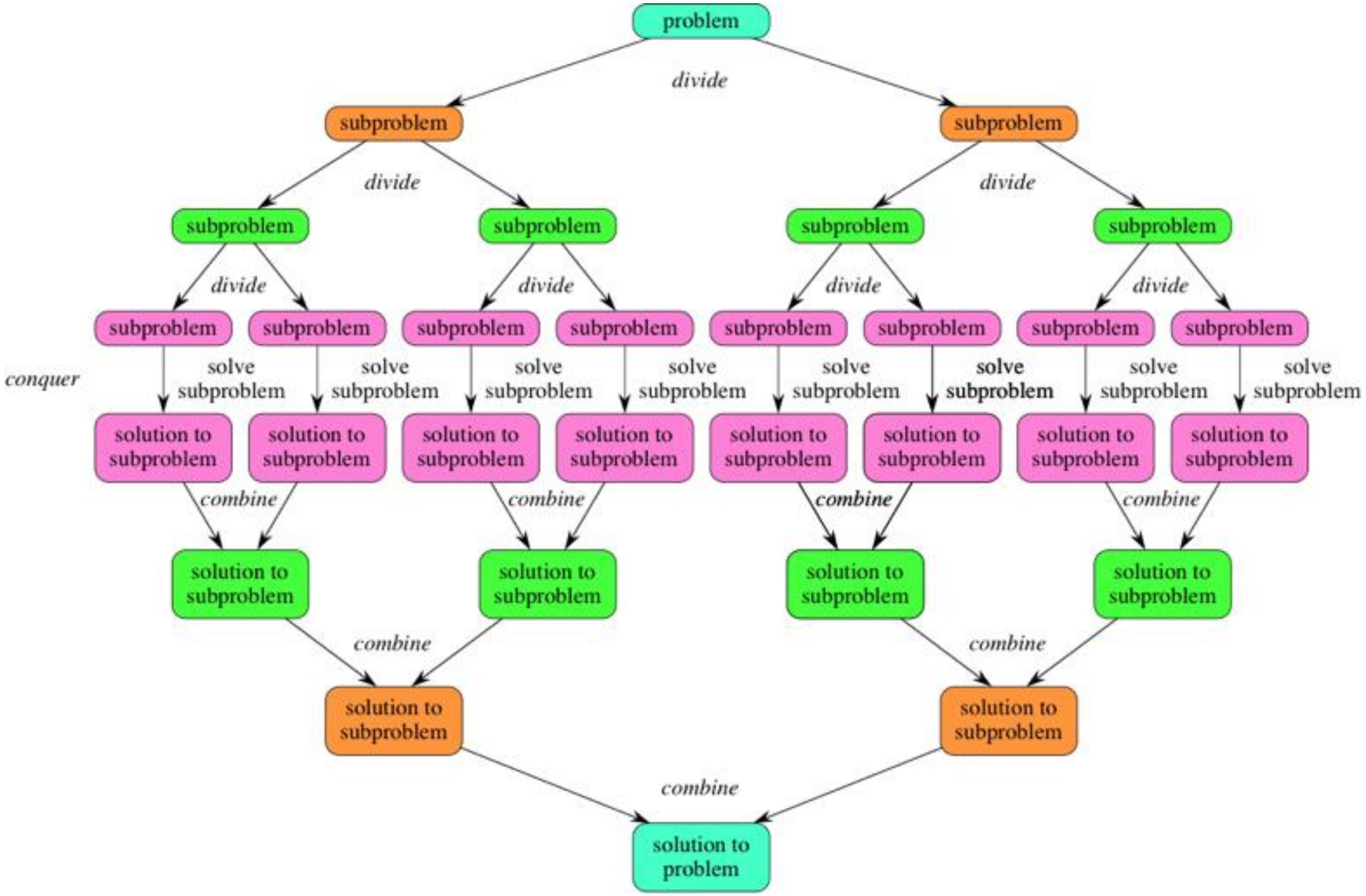
- Use recursion in an algorithmic design
- Three steps:
  - **Divide**
  - **Conquer**
  - **Combine**



Reference

Divide-and-Conquer

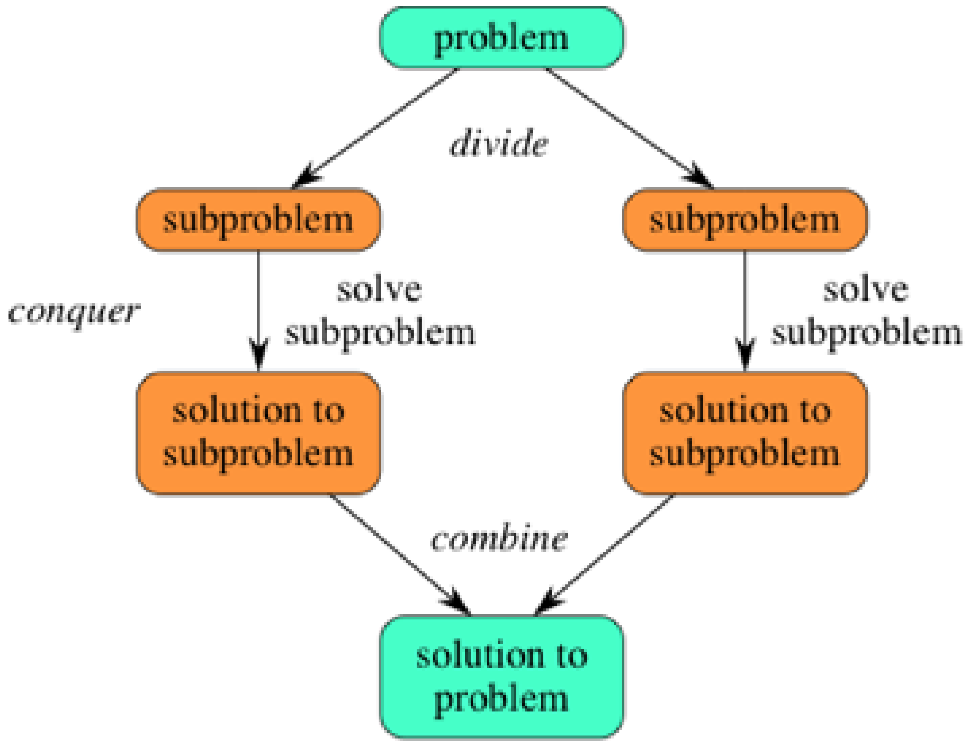
- Use recursion in an algorithmic design
- Three steps:
  - **Divide**
  - **Conquer**
  - **Combine**



[Reference]

Divide-and-Conquer

- Use recursion in an algorithmic design
- Three steps:
  - **Divide**
  - **Conquer**
  - **Combine**
- Example:
  - Merge Sort
  - Quick Sort



Divide-and-Conquer

Schema

```
DivideConquer(n)
    if n <= n0:                        # If n enough small (base case)
        return solve(A)
    else:
        # Divide into a subproblems, each instance: n/b
        subproblem in subproblems:
            DivideConquer(n/b)
        combine(all subproblems)
    return solution
```

Divide-and-Conquer

MERGE SORT

- **Divide:** divide the n-element sequence into two subsequences of n/2 elements each

  base case: sequence has length = 1, is already sorted
- **Conquer:** sort the two subsequences recursively using merge sort
- **Combine:** merge the two sorted subsequences to produce the sorted sequence

7	2	5	30	16	8	4	17

$\xrightarrow{\textbf{Sorting}}$

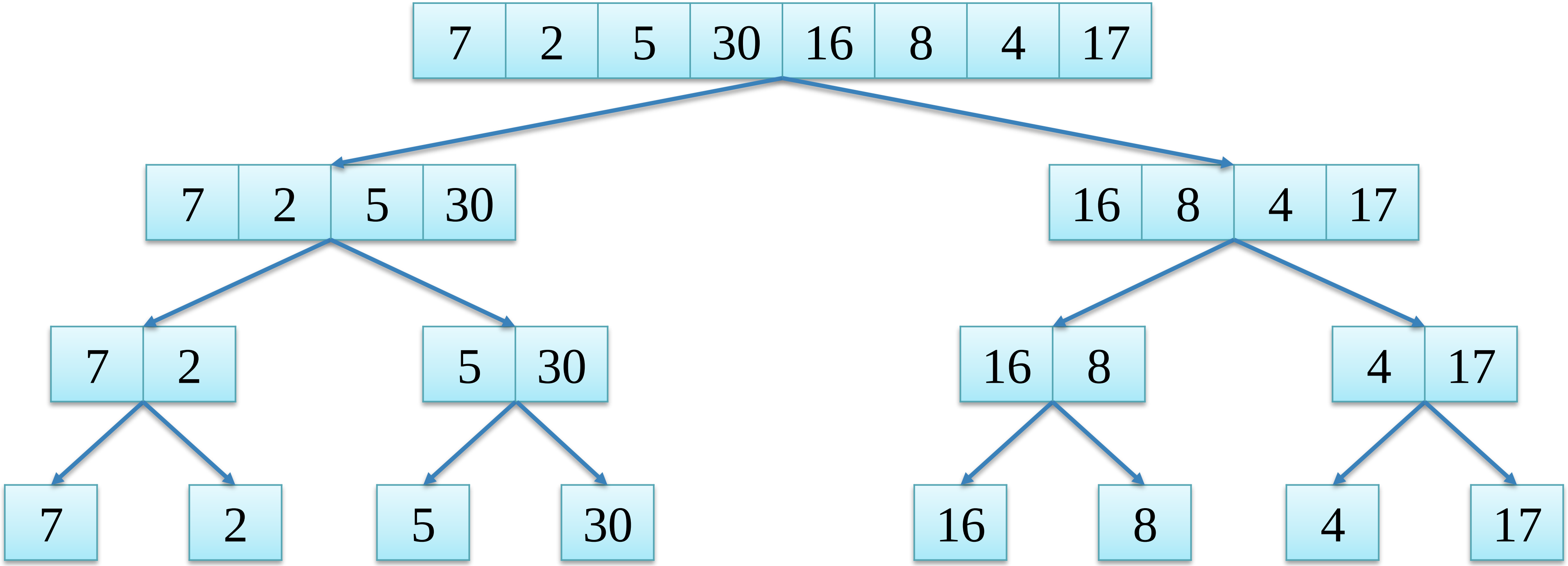
2	4	5	7	8	16	17	30

AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

MERGE SORT

- Divide – Conquer - Combine

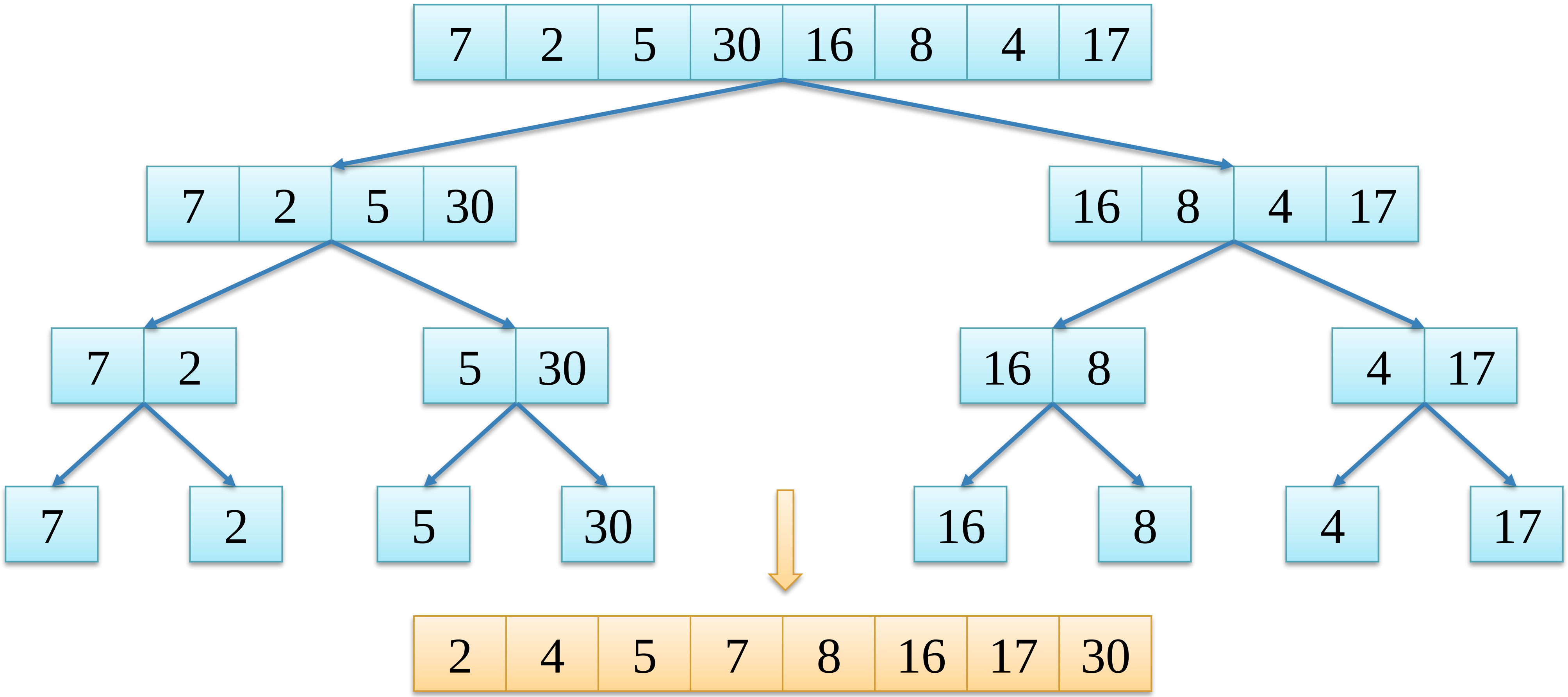


AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

MERGE SORT

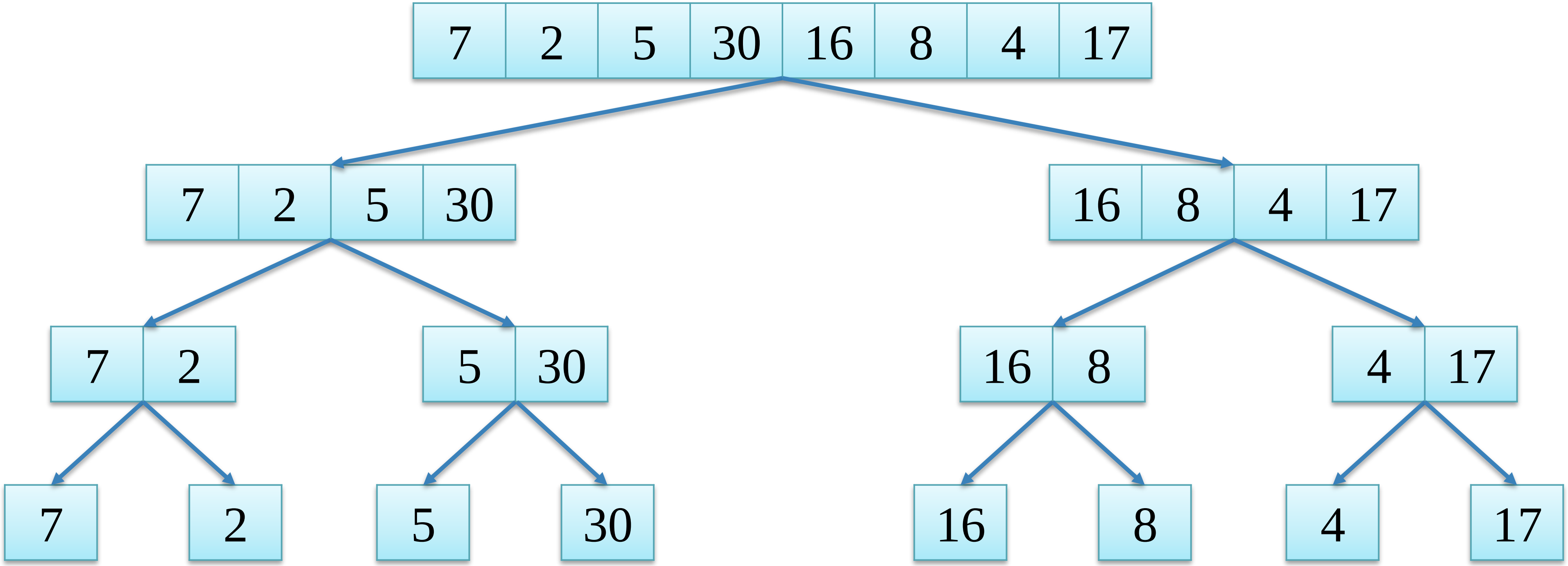
- Divide – Conquer - Combine



Divide-and-Conquer

MERGE SORT

- Divide – Conquer - Combine

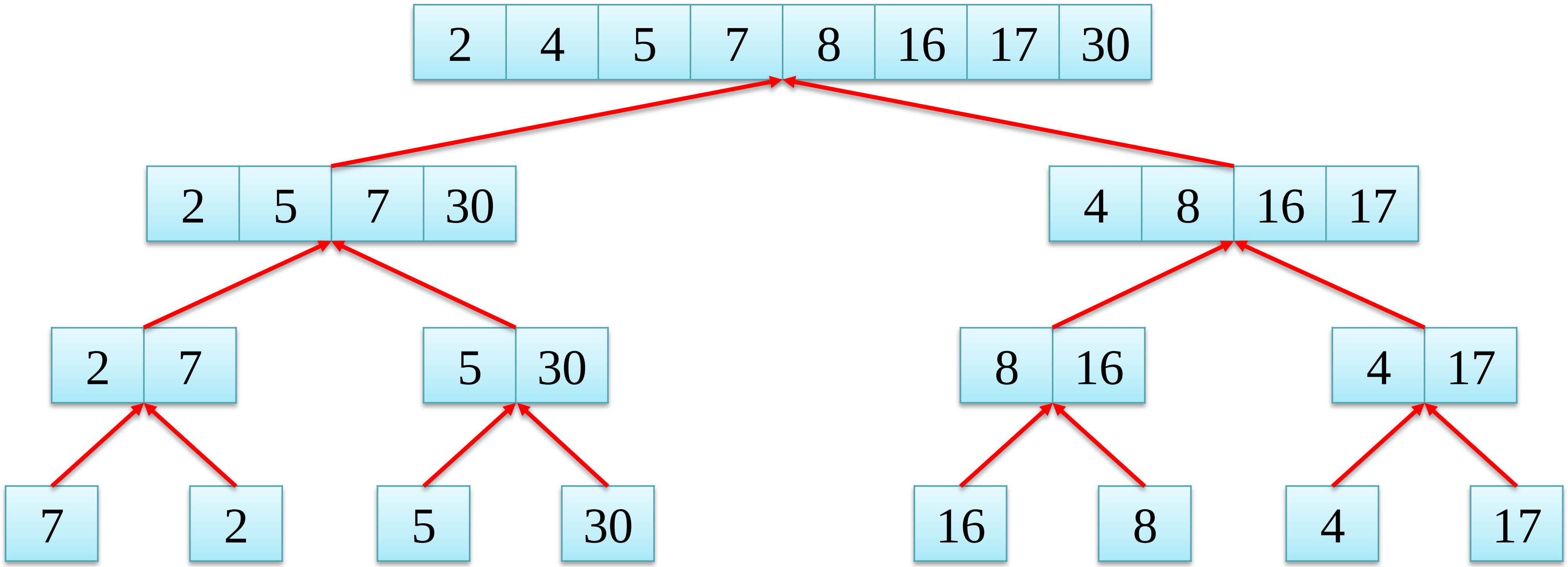


Merge is the key operation in merge sort

Divide-and-Conquer

MERGE SORT

- **Divide – Conquer - Combine**

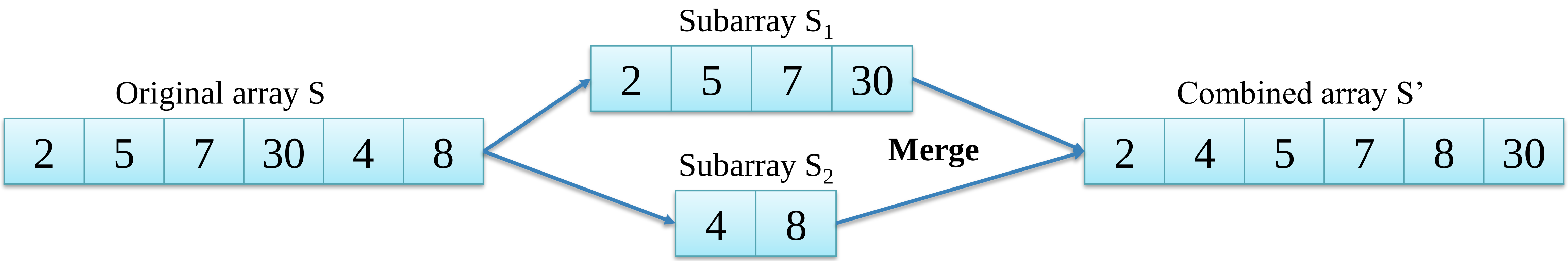


Merge is the key operation in merge sort

Divide-and-Conquer

MERGE SORT

- **Merge Function**



Divide-and-Conquer

MERGE SORT

➢ Merge Function

Subarray S_1

2	5	7	30

i=0

Subarray S_2

4	8

j=0

Compare

2	4

Combined array S’

2					

i+j=0

Subarray S_1

2	5	7	30

i=1

Subarray S_2

4	8

j=0

Compare

5	4

Combined array S’

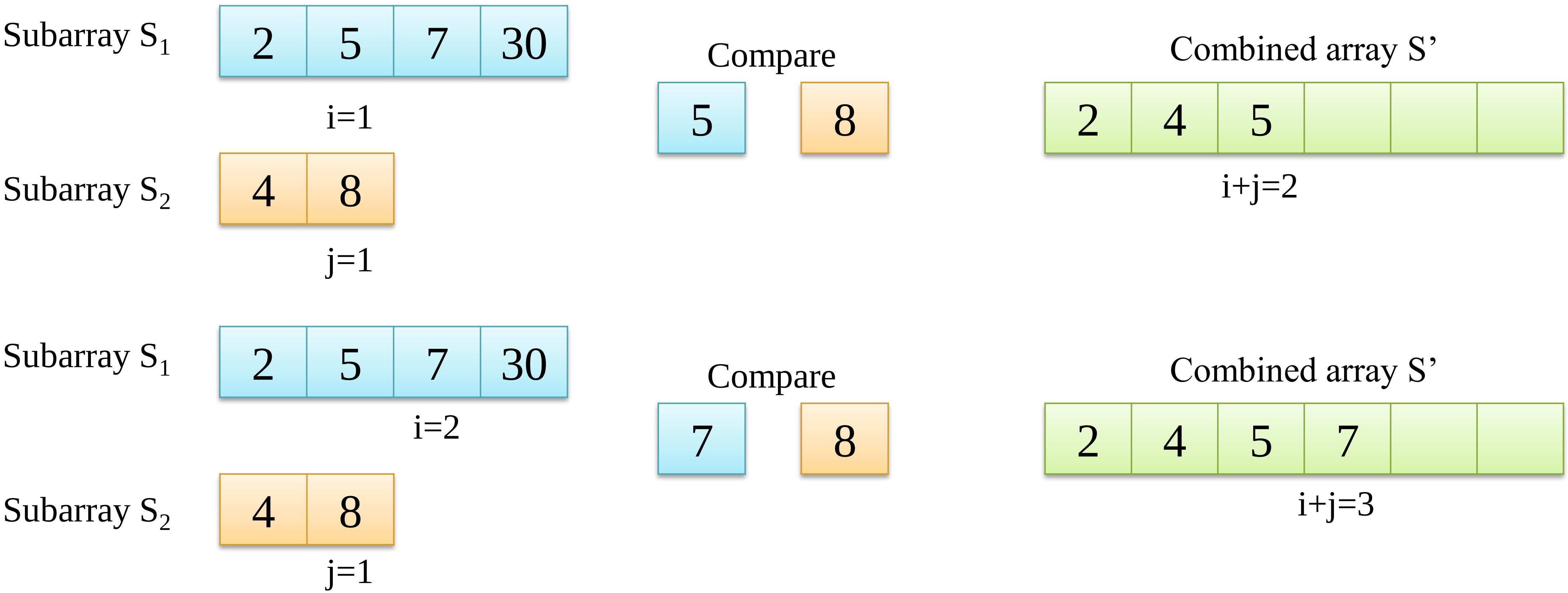
2	4				

i+j=1

Divide-and-Conquer

MERGE SORT

➢ **Merge Function**



Divide-and-Conquer

MERGE SORT

➢ Merge Function

Subarray S_1

2	5	7	30

i=3

Compare

30	8

Combined array S’

2	4	5	7	8	

i+j=4

Subarray S_2

4	8

j=1

Subarray S_1

2	5	7	30

i=3

Compare

30

Combined array S’

2	4	5	7	8	30

i+j=5

Subarray S_2

4	8

j=2

Divide-and-Conquer

MERGE SORT

➢ **Merge Function**

The number of compare operation?

```
def merge(S1, S2, S):
    i = j = 0
    while i + j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
            S[i+j] = S1[i]
            i += 1
        else:
            S[i+j] = S2[j]
            j += 1

S = [2, 5, 7, 30, 4, 8]
S1 = [2, 5, 7, 30]
S2 = [4, 8]
merge(S1, S2, S)
S
```

```
[2, 4, 5, 7, 8, 30]
```

Divide-and-Conquer

MERGE SORT

- **Merge Function**

$T(n)_{merge}$ is $O(n)$

```
def merge(S1, S2, S):
    i = j = 0
    while i + j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
            S[i+j] = S1[i]
            i += 1
        else:
            S[i+j] = S2[j]
            j += 1

S = [2, 5, 7, 30, 4, 8]
S1 = [2, 5, 7, 30]
S2 = [4, 8]
merge(S1, S2, S)
S
```

```
[2, 4, 5, 7, 8, 30]
```

Divide-and-Conquer

MERGE SORT

```
def merge(S1, S2, S):
    i = j = 0
    while i + j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
            S[i+j] = S1[i]
            i += 1
        else:
            S[i+j] = S2[j]
            j += 1

S = [2, 5, 7, 30, 4, 8]
S1 = [2, 5, 7, 30]
S2 = [4, 8]
merge(S1, S2, S)
S
```

```
[2, 4, 5, 7, 8, 30]
```

```
[6] def merge_sort(S):
        n = len(S)

        if n < 2:
            return

        mid = n//2
        S1 = S[0:mid]
        S2 = S[mid:n]

        merge_sort(S1)
        merge_sort(S2)

        merge(S1, S2, S)

    S = [7, 2, 5, 30, 16, 8, 4, 17]
    merge_sort(S)
    S
```

Base case

Divide

Conquer

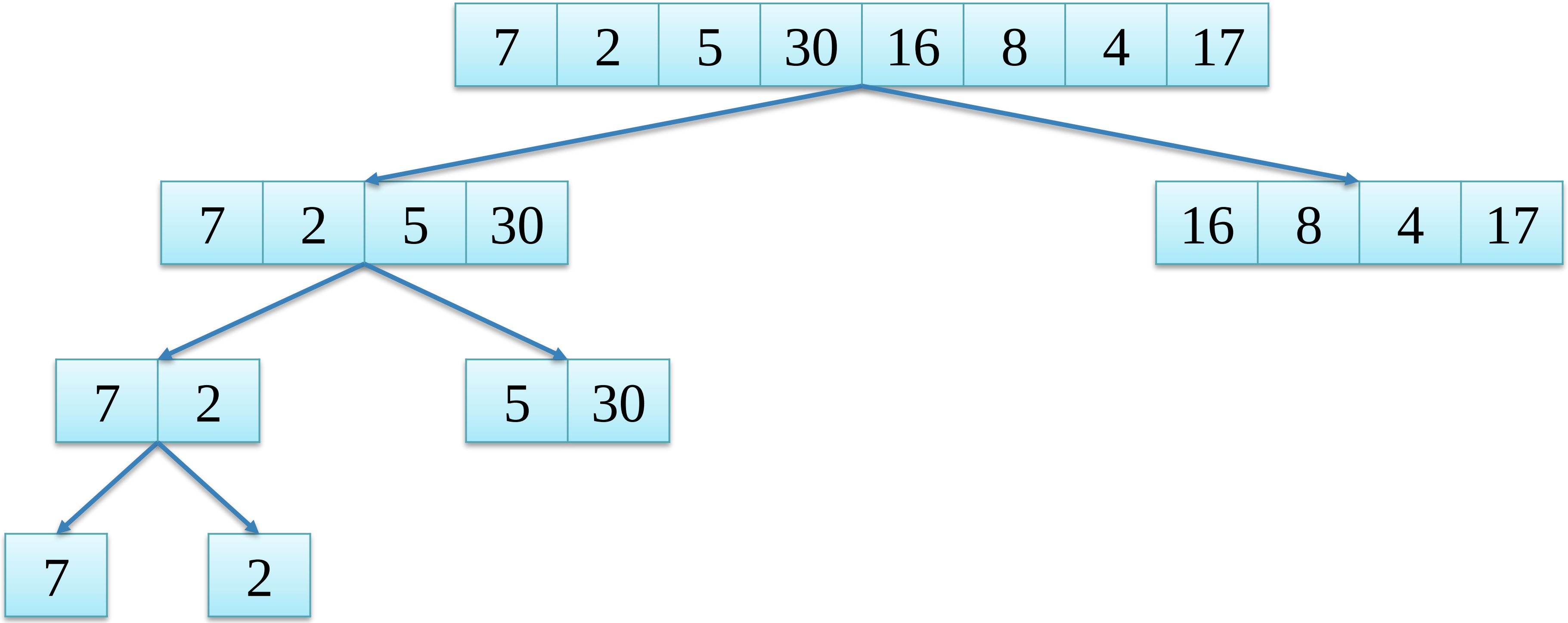
Combine

```
[2, 4, 5, 7, 8, 16, 17, 30]
```

Divide-and-Conquer

MERGE SORT

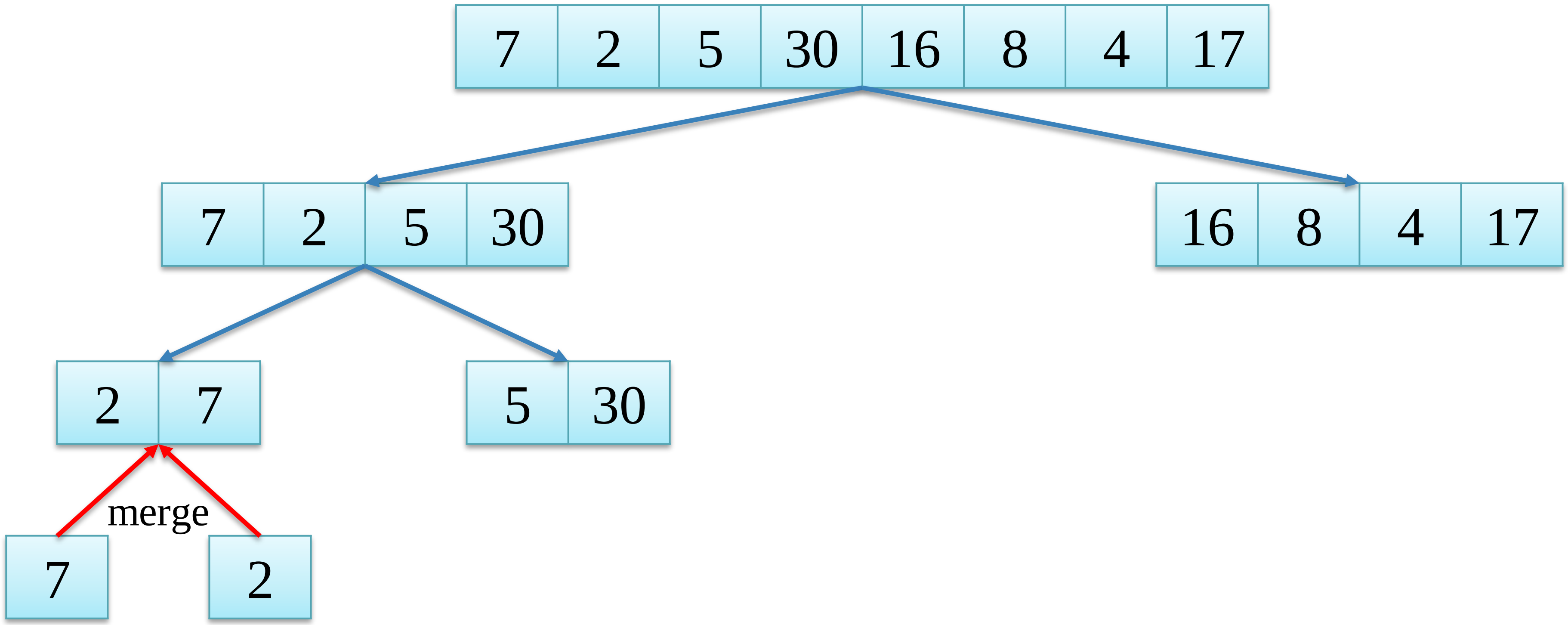
- Divide – Conquer - Combine



Divide-and-Conquer

MERGE SORT

- Divide – Conquer - Combine

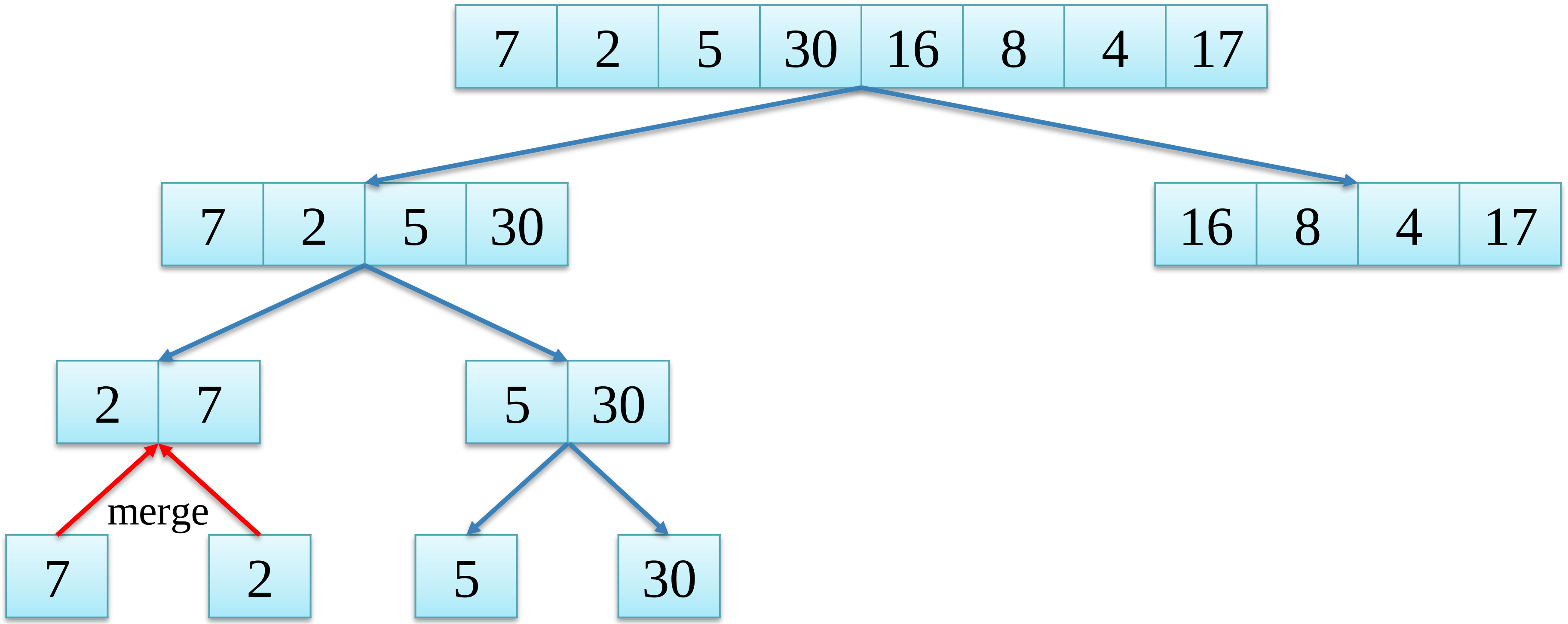


AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

MERGE SORT

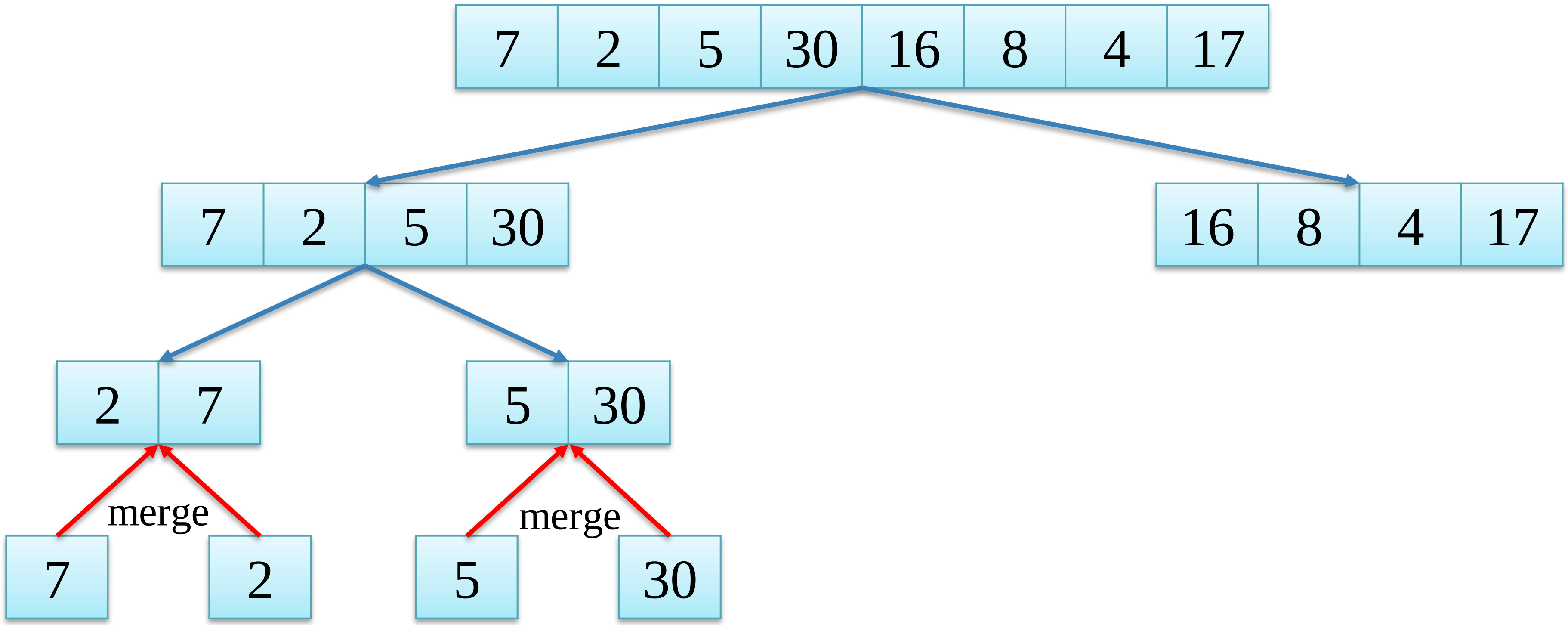
- **Divide – Conquer - Combine**



Divide-and-Conquer

MERGE SORT

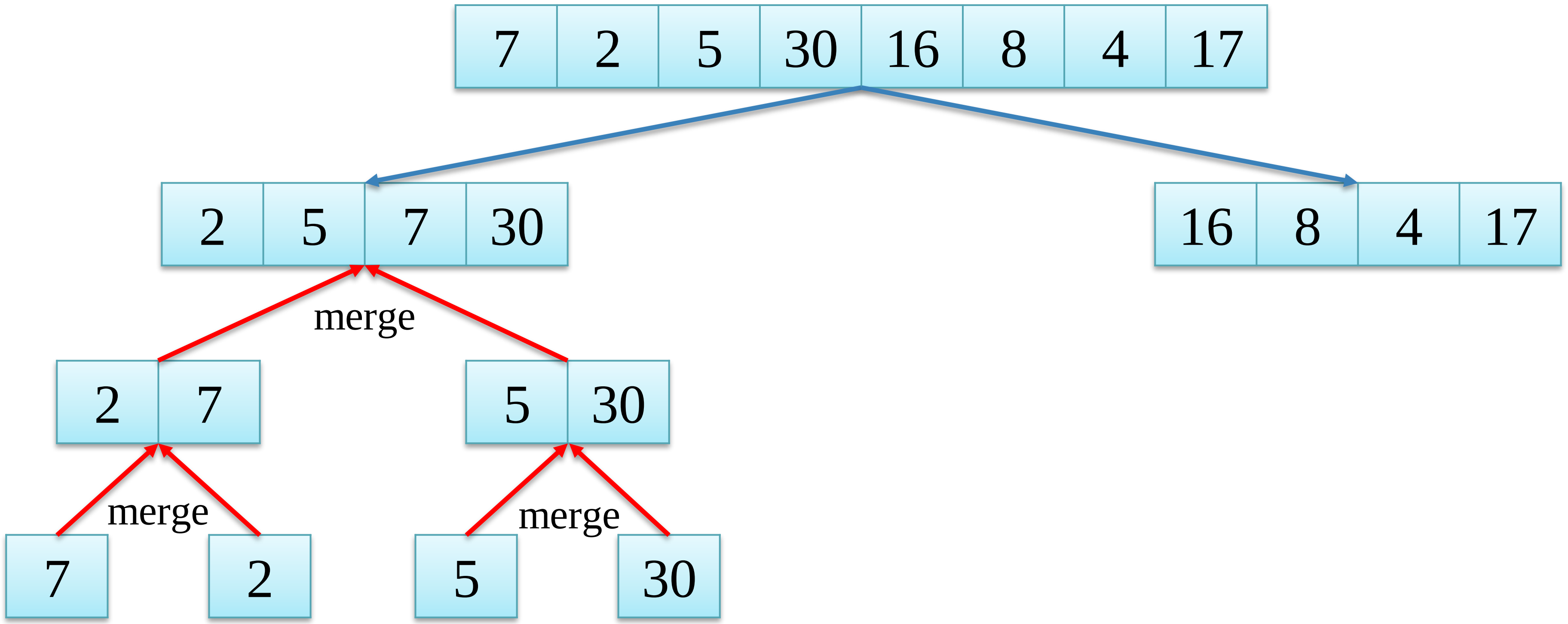
- **Divide – Conquer - Combine**



Divide-and-Conquer

MERGE SORT

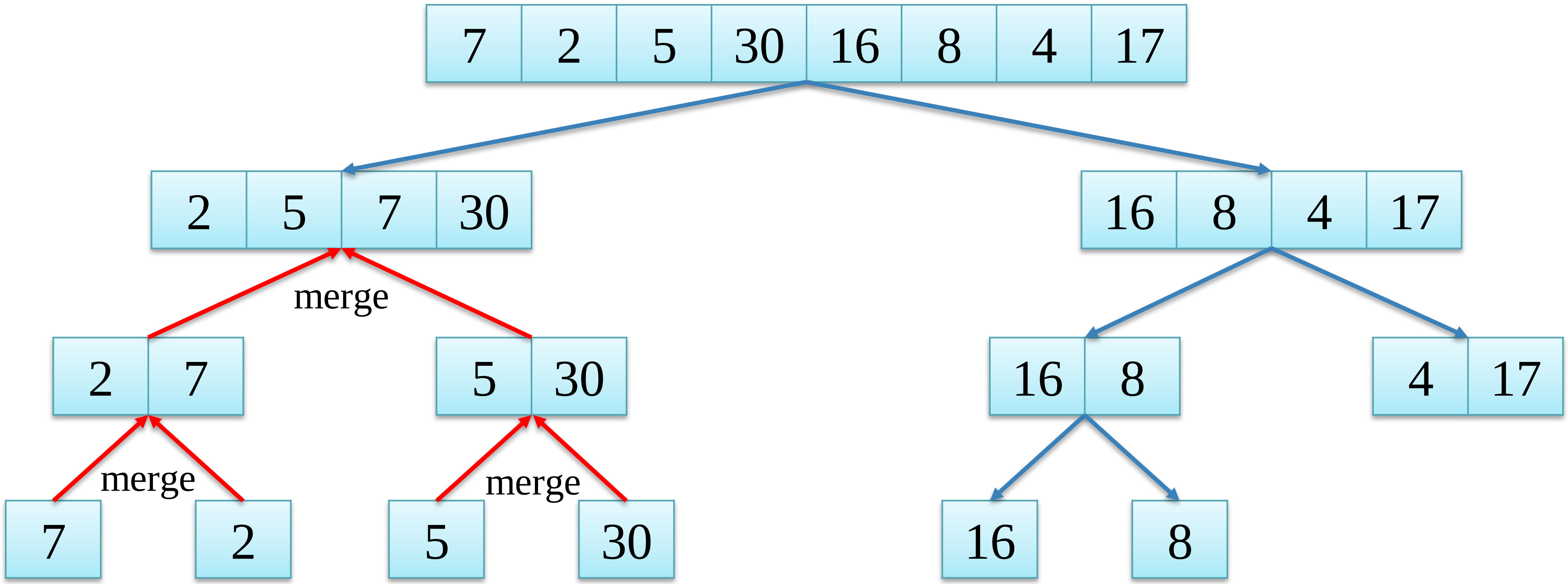
- **Divide – Conquer - Combine**



Divide-and-Conquer

MERGE SORT

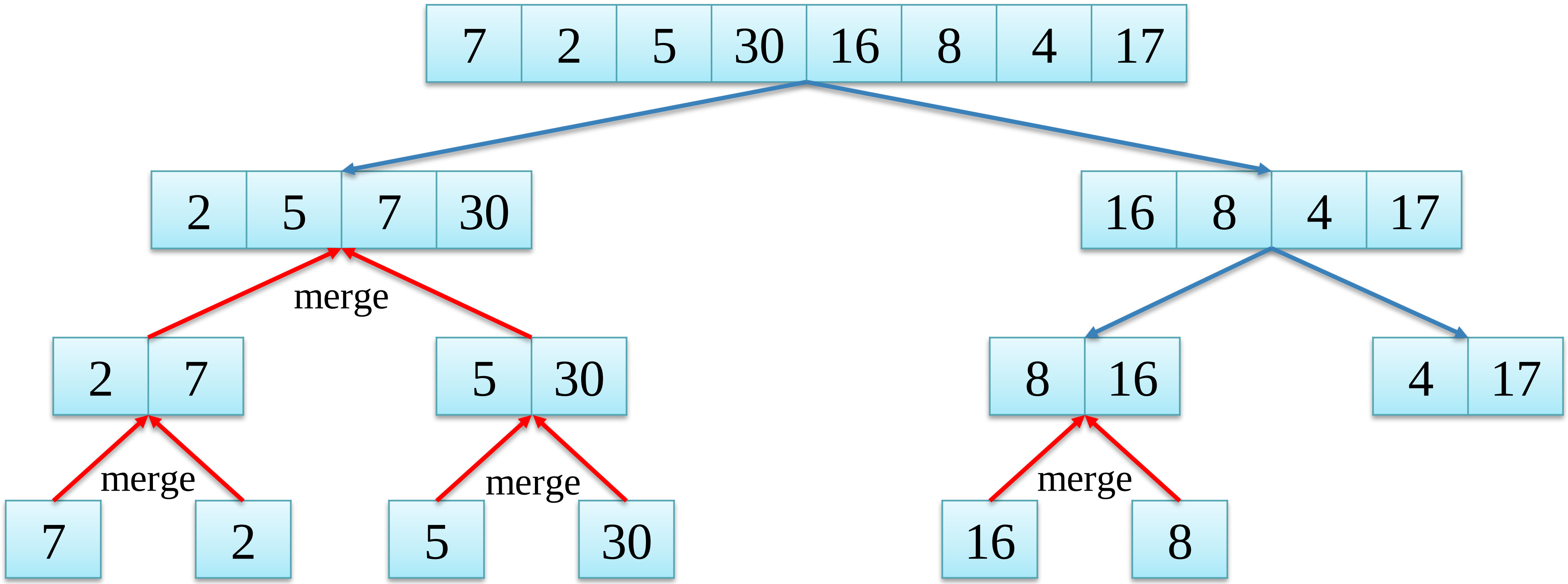
- Divide – Conquer - Combine



Divide-and-Conquer

MERGE SORT

- Divide – Conquer - Combine

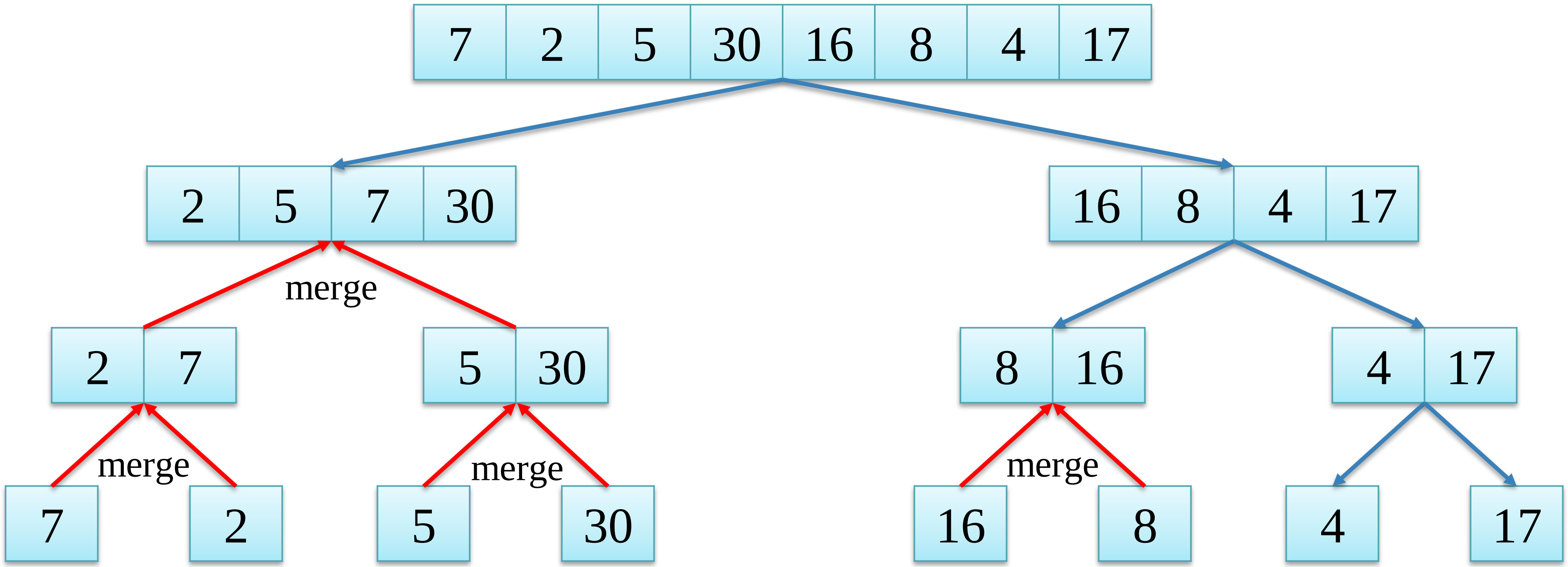


AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

MERGE SORT

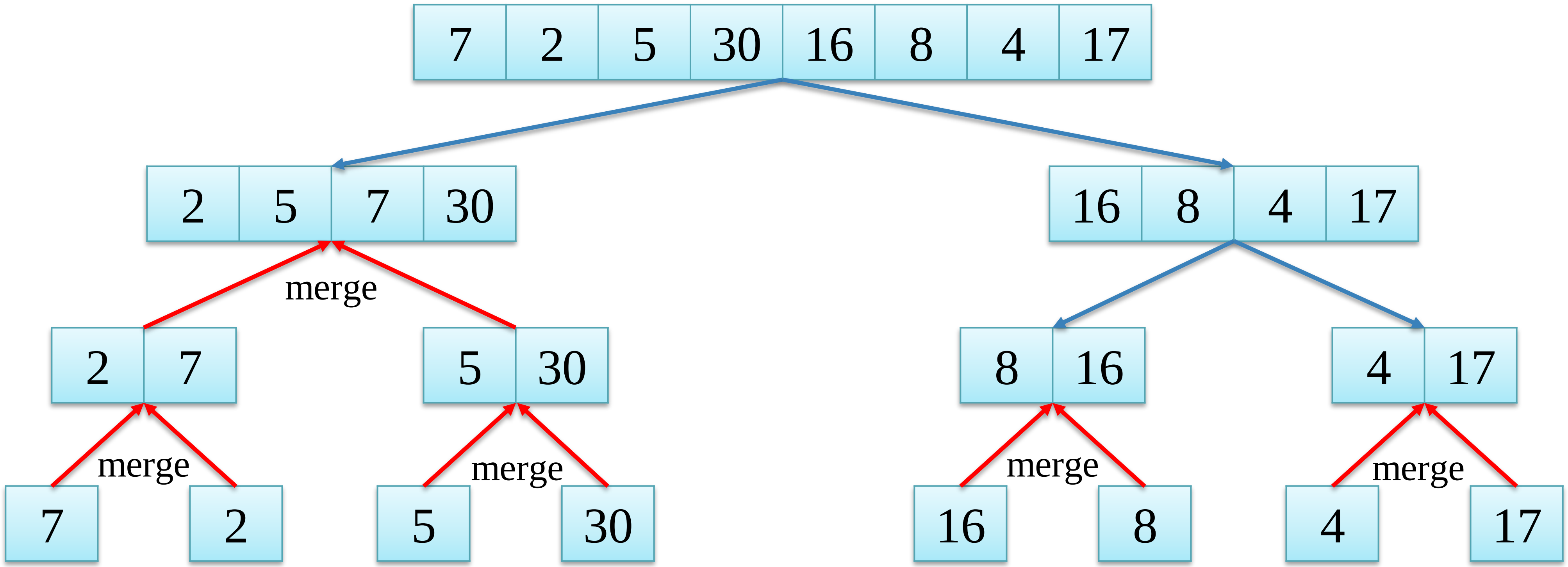
- Divide – Conquer - Combine



Divide-and-Conquer

MERGE SORT

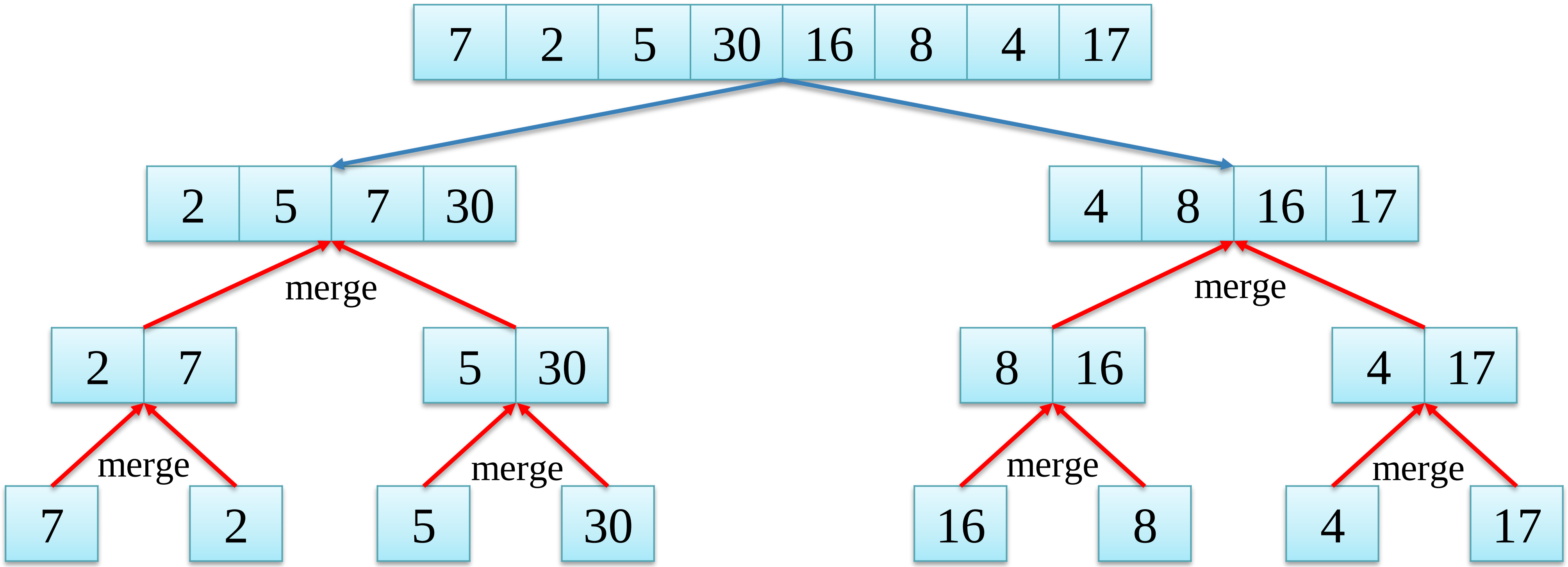
- Divide – Conquer - Combine



Divide-and-Conquer

MERGE SORT

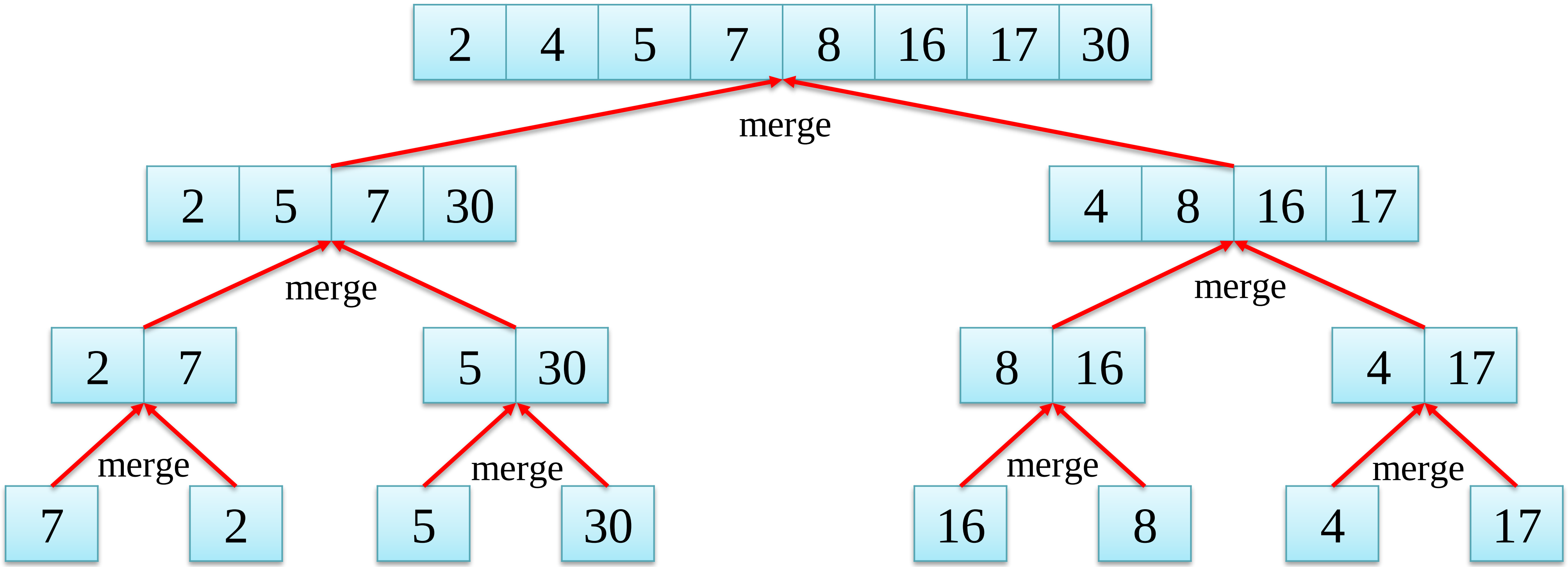
- **Divide – Conquer - Combine**



Divide-and-Conquer

MERGE SORT

- **Divide – Conquer - Combine**



Divide-and-Conquer

Analysis of Divide-and-Conquer

- Described by recursive equation
- Suppose T(n) is the running time on a problem of size n

$$T(n) = \begin{cases} O(1) & \text{if } n \le n_c \\ aT(n/b) + D(n) + C(n) & \text{if } n > n_c \end{cases}$$

Where:

- a: number of subproblems
- n/b: size of each subproblem
- D(n): cost of divide operation
- C(n): cost of combination operation

AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

MERGE SORT

- **Analysis of MERGE SORT**
  - **Divide:** D(n) = O(1)
  - **Conquer:** a=2, b=2 => 2T(n/2)
  - **Combine:** C(n) = O(n)

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

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

Compute T(n)

Divide-and-Conquer

MERGE SORT

- **Analysis of MERGE SORT**
  - **Divide:** D(n) = O(1)
  - **Conquer:** a=2, b=2 => 2T(n/2)
  - **Combine:** C(n) = O(n)

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

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

Compute T(n)

Recursion-tree Method

Master Theorem

Divide-and-Conquer

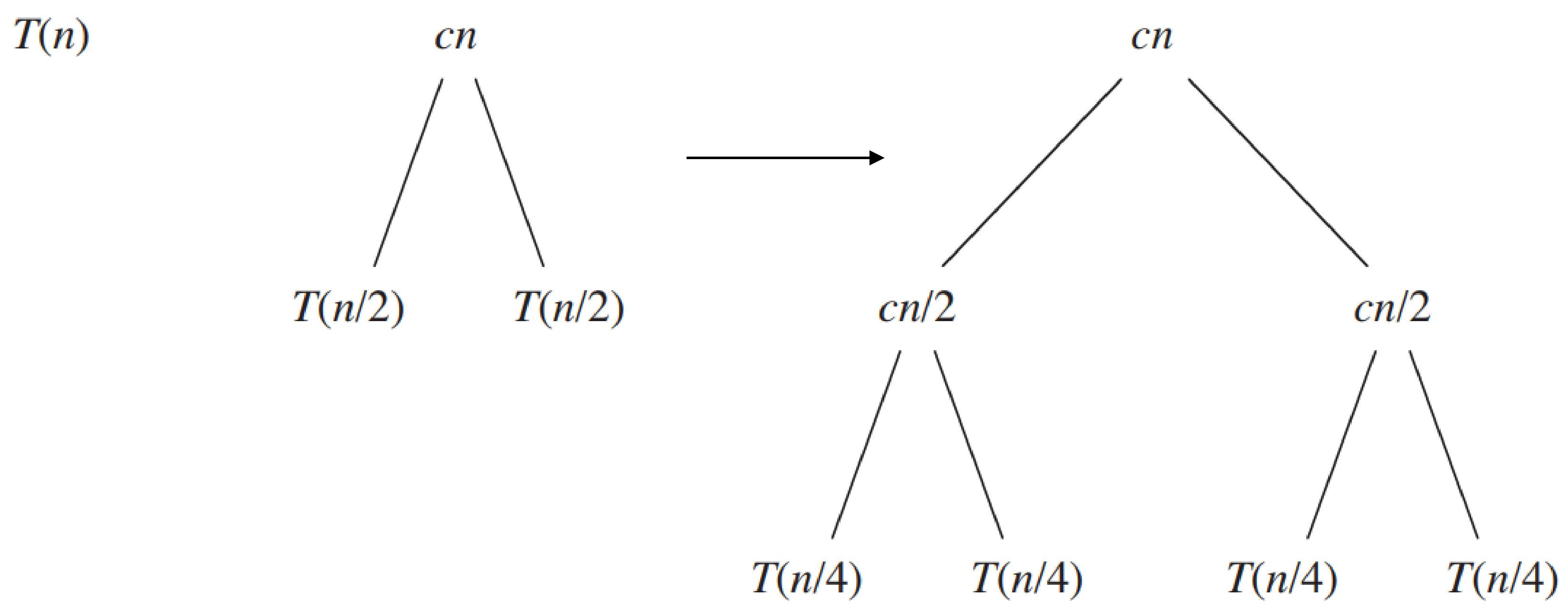
The Recursion-tree Method

- Idea:
  - Each node represents the cost of a single subproblem
  - Sum up the costs with each level to get level cost
  - Sum up all the level costs to get total cost
- Particularly suitable for divide-and-conquer recurrence
- Best used to generate a good guess, tolerating “sloppiness”
- If trying carefully to draw the recursion-tree and compute cost, then used as direct proof

Divide-and-Conquer

Recursion-tree for MERGE SORT

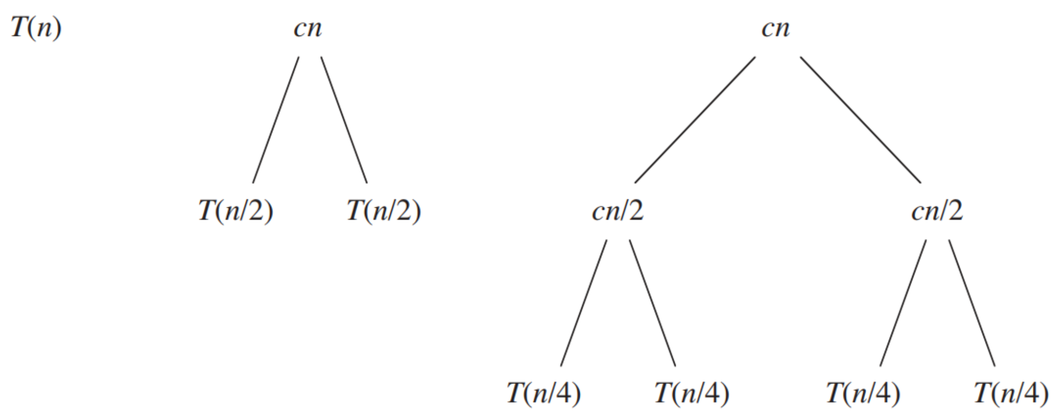
$$T(n) = 2T(n/2) + cn$$

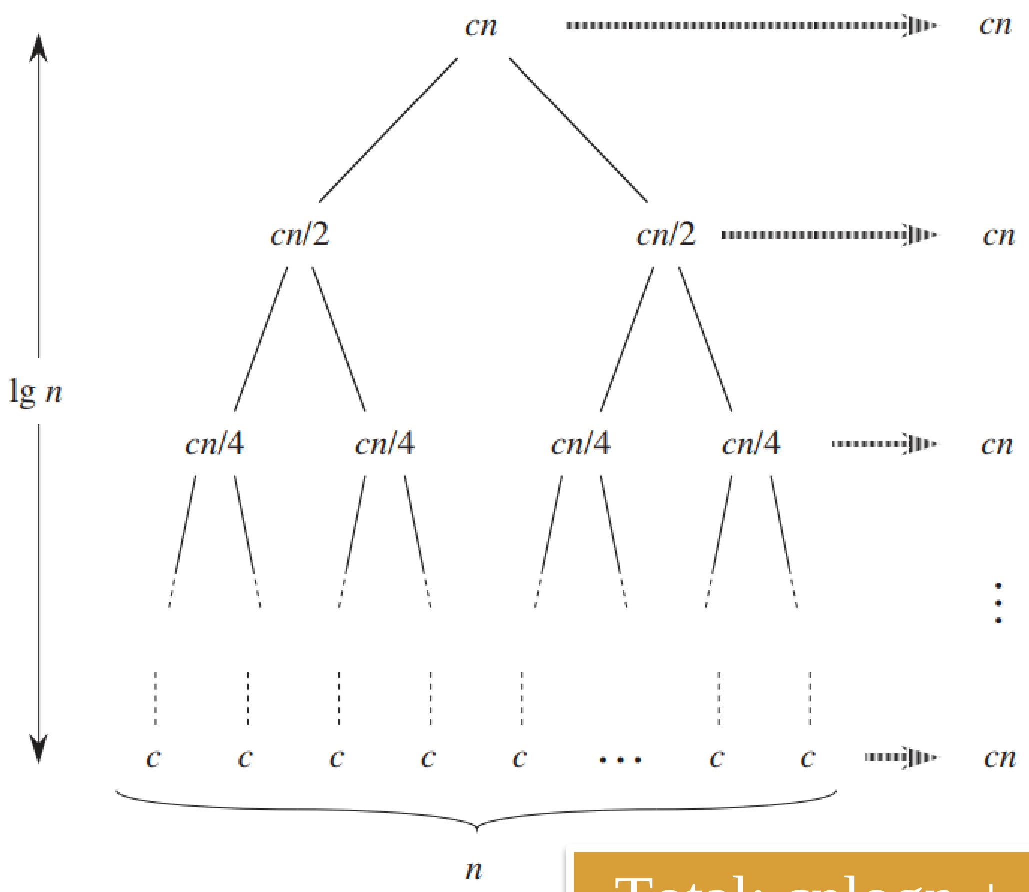


Divide-and-Conquer

Recursion-tree for MERGE SORT

$$T(n) = 2T(n/2) + cn$$



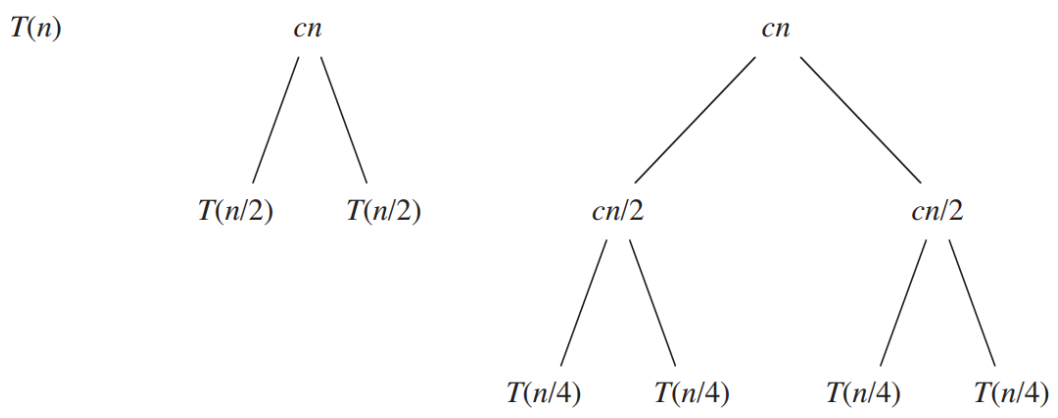


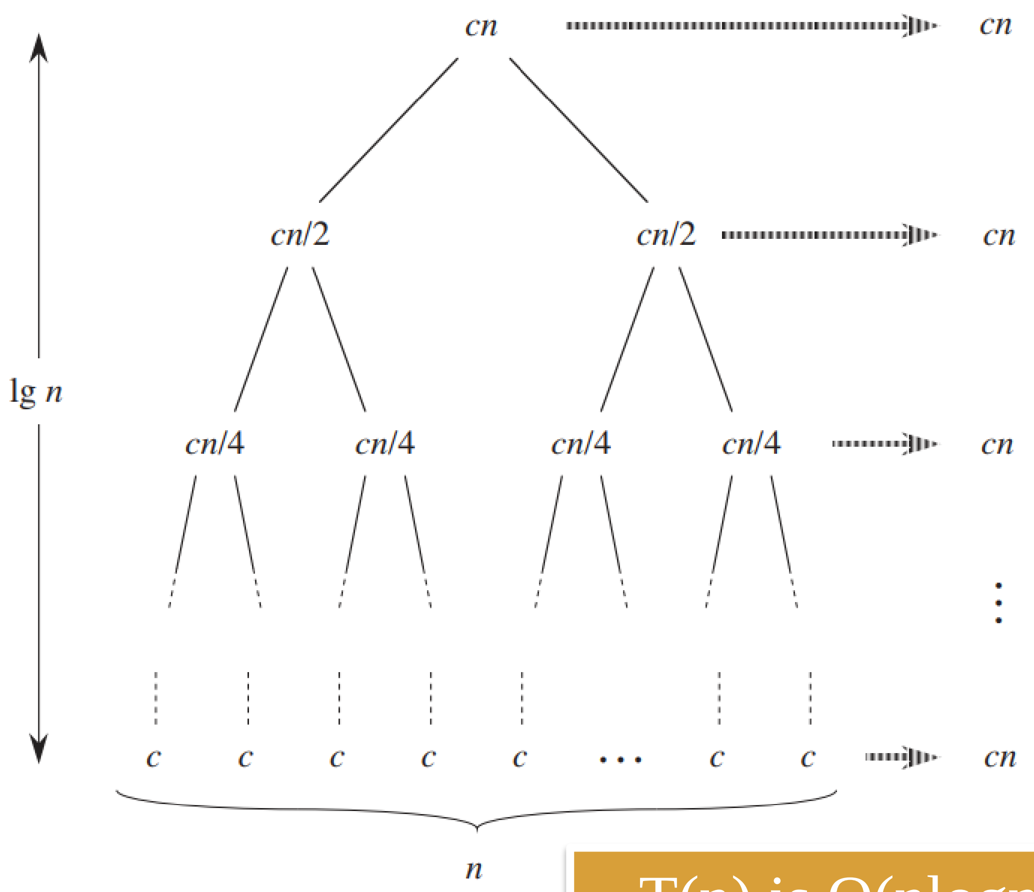
Total: cnlogn + cn

Divide-and-Conquer

Recursion-tree for MERGE SORT

$$T(n) = 2T(n/2) + cn$$

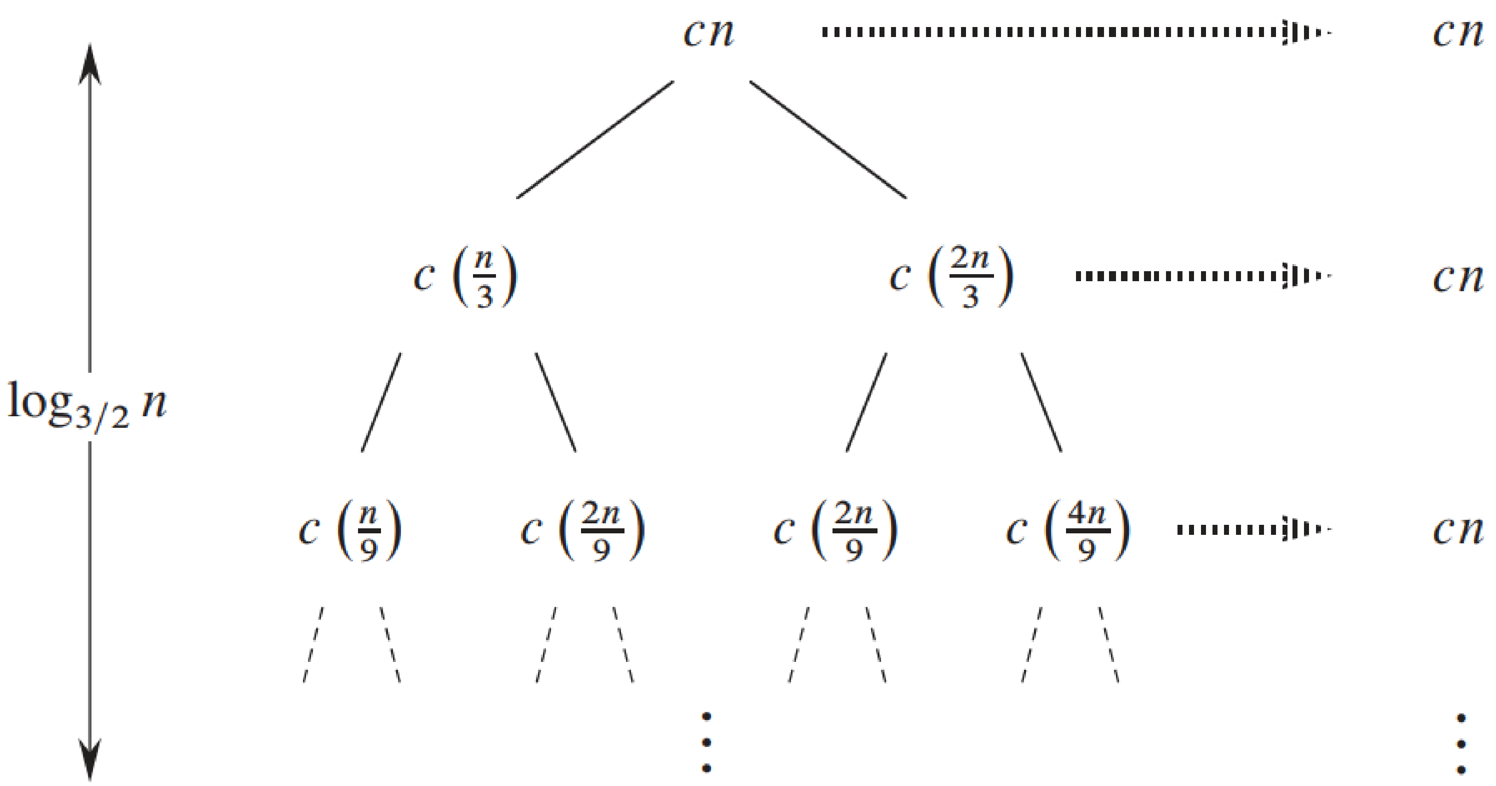




T(n) is O(nlogn)

Divide-and-Conquer

Recursion-tree for T(n) = T(n/3) + T(2n/3) + O(n)



T(n) is O(nlogn)

Divide-and-Conquer

Master Method/Theorem

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

$a \geq 1$, $b > 1$ are positive integers, f(n) is non-negative function

- Three case:

Case 1: $n^{\log_b(a)-\varepsilon} > f(n)$; then $T(n)$ is $O\left(n^{\log_b(a)}\right)$

Case 2: $n^{\log_b(a)} = f(n)$; then $T(n)$ is $O\left(n^{\log_b(a)} * \log(n)\right)$

Case 3: $n^{\log_b(a)+\varepsilon} < f(n)$; then $T(n)$ is $O(f(n))$

Divide-and-Conquer

Master Theorem for MERGE SORT

T(n) = 2T(n/2) + n

=> a=2, b=2, f(n)=n

=> $n^{\log_b(a)} = n^{\log_2(2)} = n = f(n)$

- By Case 2: $n^{\log_b(a)} = f(n);$ then $T(n)$ is $O\left(n^{\log_b(a)} * \log(n)\right)$

=> T(n) is O(nlogn)

Divide-and-Conquer

Master Theorem

T(n) = 9T(n/3) + n

=> a=9, b=3, f(n)=n

=> $n^{\log_b(a)} = n^{\log_3(9)} = n^2$

=> $f(n) = n = n^{\log_3(9) - \varepsilon}$ for $\varepsilon = 1$

- By Case 1: $n^{\log_b(a) - \varepsilon} > f(n)$; then $T(n)$ is $O\left(n^{\log_b(a)}\right)$

=> T(n) is O(n^2)

AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

MERGE SORT

```python
def merge(S1, S2, S):
    i = j = 0
    while i + j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
            S[i+j] = S1[i]
            i += 1
        else:
            S[i+j] = S2[j]
            j += 1

S = [2, 5, 7, 30, 4, 8]
S1 = [2, 5, 7, 30]
S2 = [4, 8]
merge(S1, S2, S)
S
```

```
[2, 4, 5, 7, 8, 30]
```

```python
[6] def merge_sort(S):
        n = len(S)

        if n < 2:
            return

        mid = n//2
        S1 = S[0:mid]
        S2 = S[mid:n]

        merge_sort(S1)
        merge_sort(S2)

        merge(S1, S2, S)

    S = [7, 2, 5, 30, 16, 8, 4, 17]
    merge_sort(S)
    S
```

```
[2, 4, 5, 7, 8, 16, 17, 30]
```

Base case

Divide: O(1)

Conquer: 2T(n/2)

Combine: O(n)

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

Divide-and-Conquer

QUICK SORT

- **Divide:** A sequence *S* is divided into subarrays by selection *x*: a pivot element (a specific element from *S*)

    L: elements less than pivot

    R: elements greater than pivot

- **Conquer:** Sort the two subarrays L and R by recursive calls to quicksort
- **Combine:** Subarrays are already sorted => no work is needed to combine

7	2	5	30	16	8	4	17

Sorting →

2	4	5	7	8	16	17	30

AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

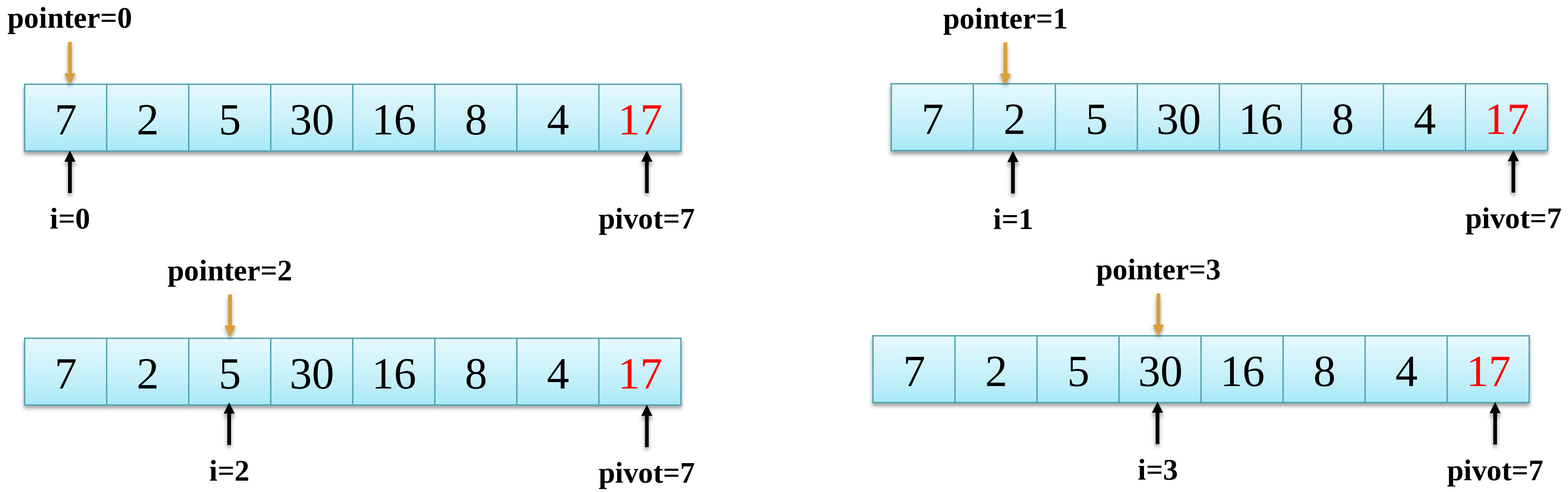
QUICK SORT

➢ **Divide:**

Select the Pivot element: the last element in S

7	2	5	30	16	8	4	17

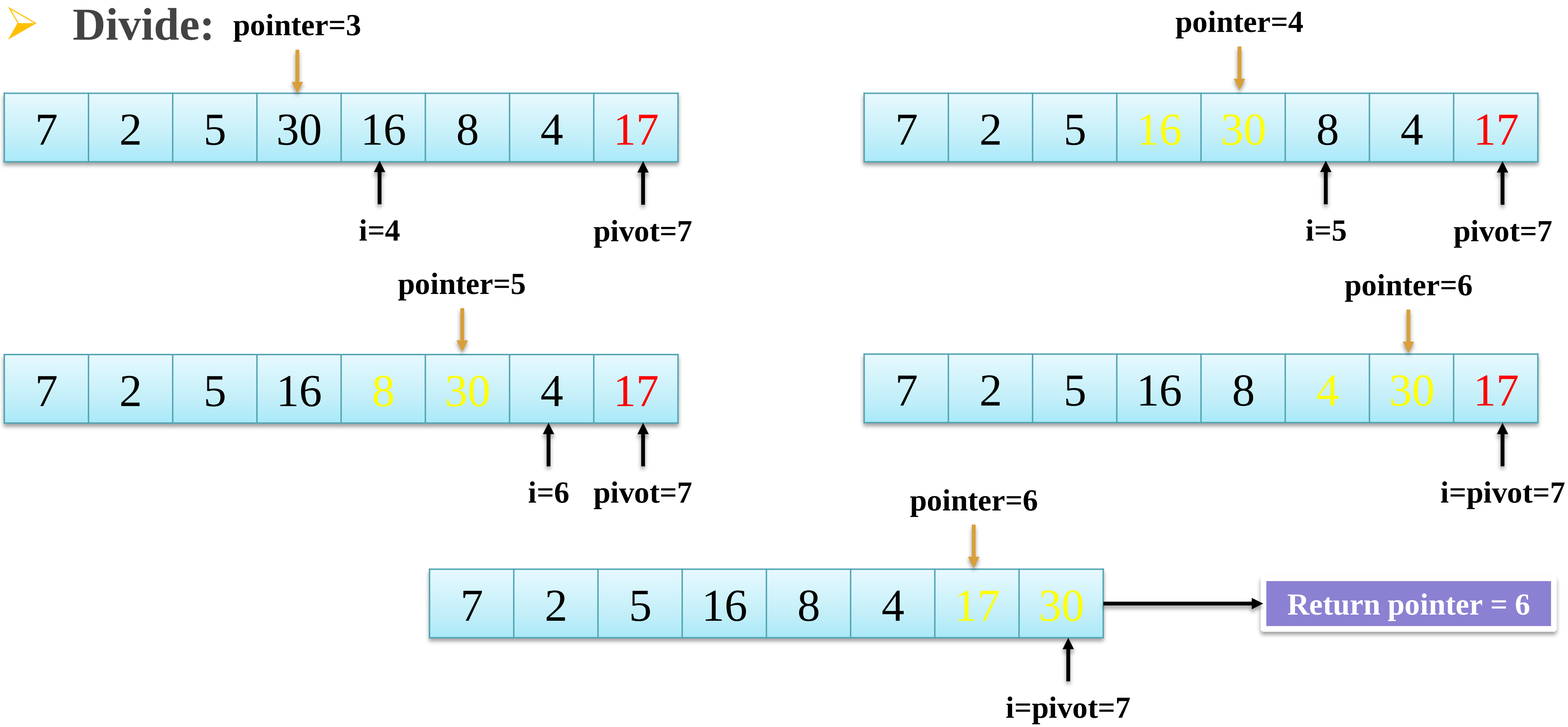
Rearrange the sequence



Divide-and-Conquer

QUICK SORT

- **Divide:**



Divide-and-Conquer

QUICK SORT

- **Divide:**

Select the Pivot element: the last element in S

Rearrange the sequence

7	2	5	16	8	4	17	30

Return pointer = 6

- **Conquer:**

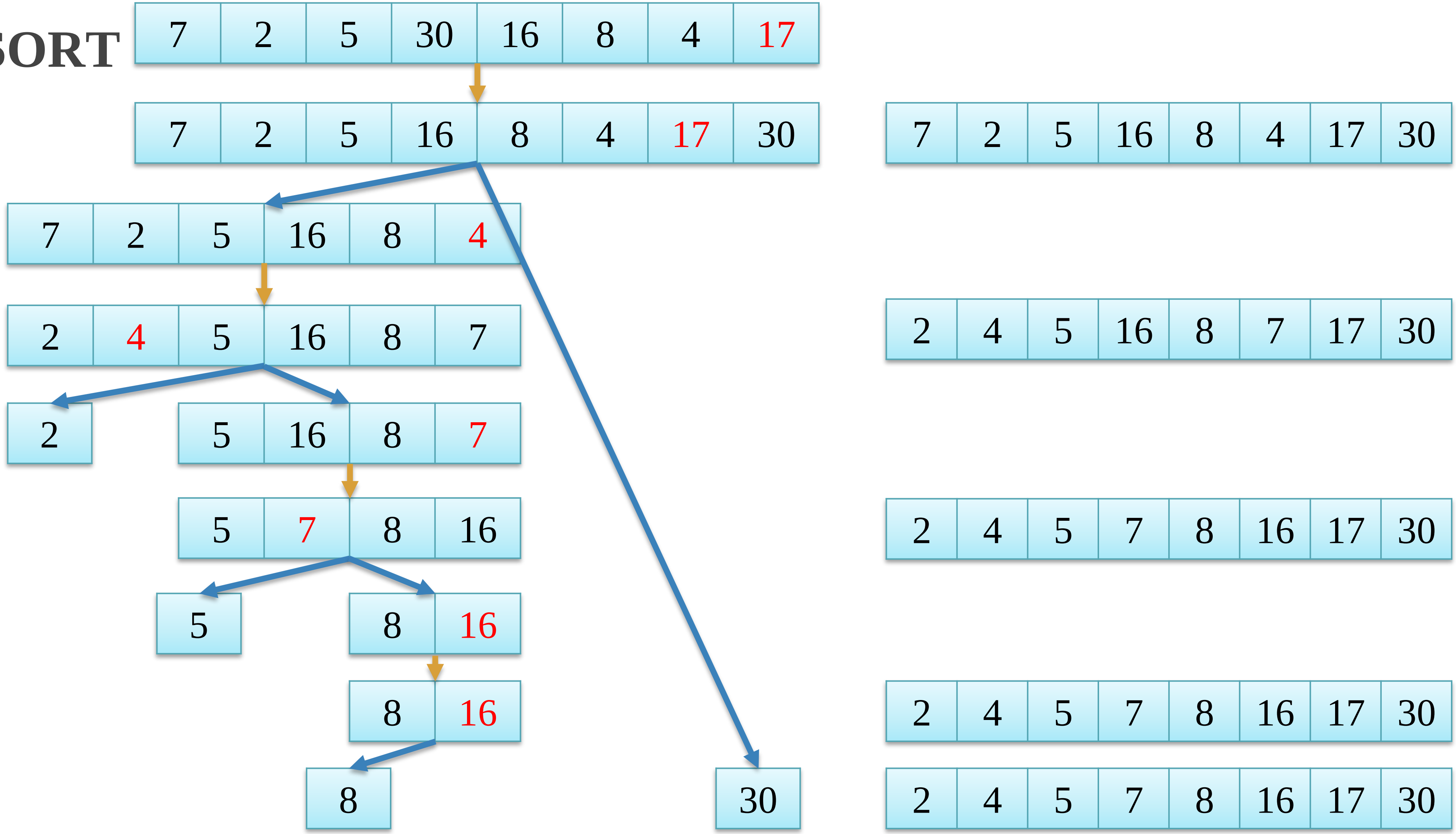
7	2	5	16	8	4	17	30

7	2	5	16	8	4

30

Divide-and-Conquer

QUICK SORT



Divide-and-Conquer

QUICK SORT

```python
def partition(low, high, S):
    pivot, pointer = S[high], low

    for i in range(low, high):
        if S[i] <= pivot:
            S[i], S[pointer] = S[pointer], S[i]
            pointer += 1

    S[pointer], S[high] = S[high], S[pointer]

    return pointer
```

$T(n)_{divide}$ is ???

```python
def quicksort(low, high, S):
    if len(S) == 1:
        return S

    if low < high:
        p = partition(low, high, S)

        quicksort(low, p-1, S)
        quicksort(p+1, high, S)


S = [7, 2, 5, 30, 16, 8, 4, 17]
low = 0
high = len(S)-1
quicksort(low, high, S)
S
```

Base case

Divide

Conquer

AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

QUICK SORT

➢ Analysis of QUICK SORT

- **Divide:** O(n)
- **Conquer:** a=2, => $T(n/b_1) + T(n/b_2)$
- **Combine:**

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(n/b_1) + T(n/b_2) + O(n) & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n/b_1) + T(n/b_2) + cn & \text{if } n > 1 \end{cases}$$

Compute b_1, b_2

Divide-and-Conquer

QUICK SORT

- **Analysis of QUICK SORT**
- **Worst-case partitioning**
  - Partition at the last element, the largest element

    => "bad" split: sequence into two subsequences of size 0 and n-1
  - **Divide:** O(n)
  - **Conquer:** a=2, => T(n-1)

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + cn & \text{if } n > 1 \end{cases}$$

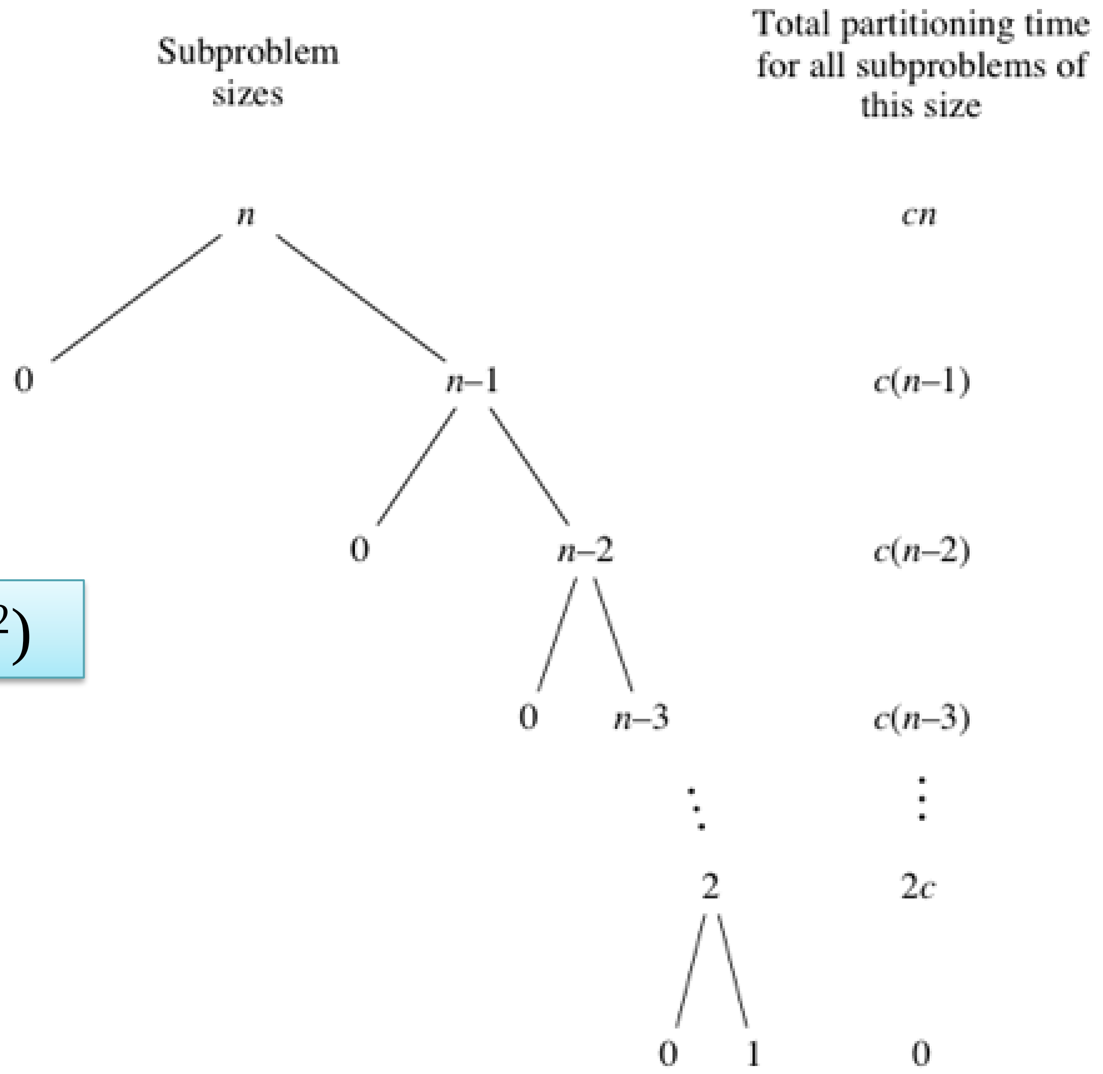
Divide-and-Conquer

QUICK SORT

- **Analysis of QUICK SORT**
- **Worst-case partitioning**

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + cn & \text{if } n > 1 \end{cases}$$

$T(n)$ is $O(n^2)$



Divide-and-Conquer

QUICK SORT

- **Analysis of QUICK SORT**
- **Best-case partitioning**
  - Partition at the average element

    => “good” split: sequence into two subsequences of size (n-1)/2
  - **Divide:** O(n)
  - **Conquer:** a=2, => 2T((n-1)/2)

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T((n-1)/2) + cn & \text{if } n > 1 \end{cases}$$

AI VIET NAM
@aivietnam.edu.vn

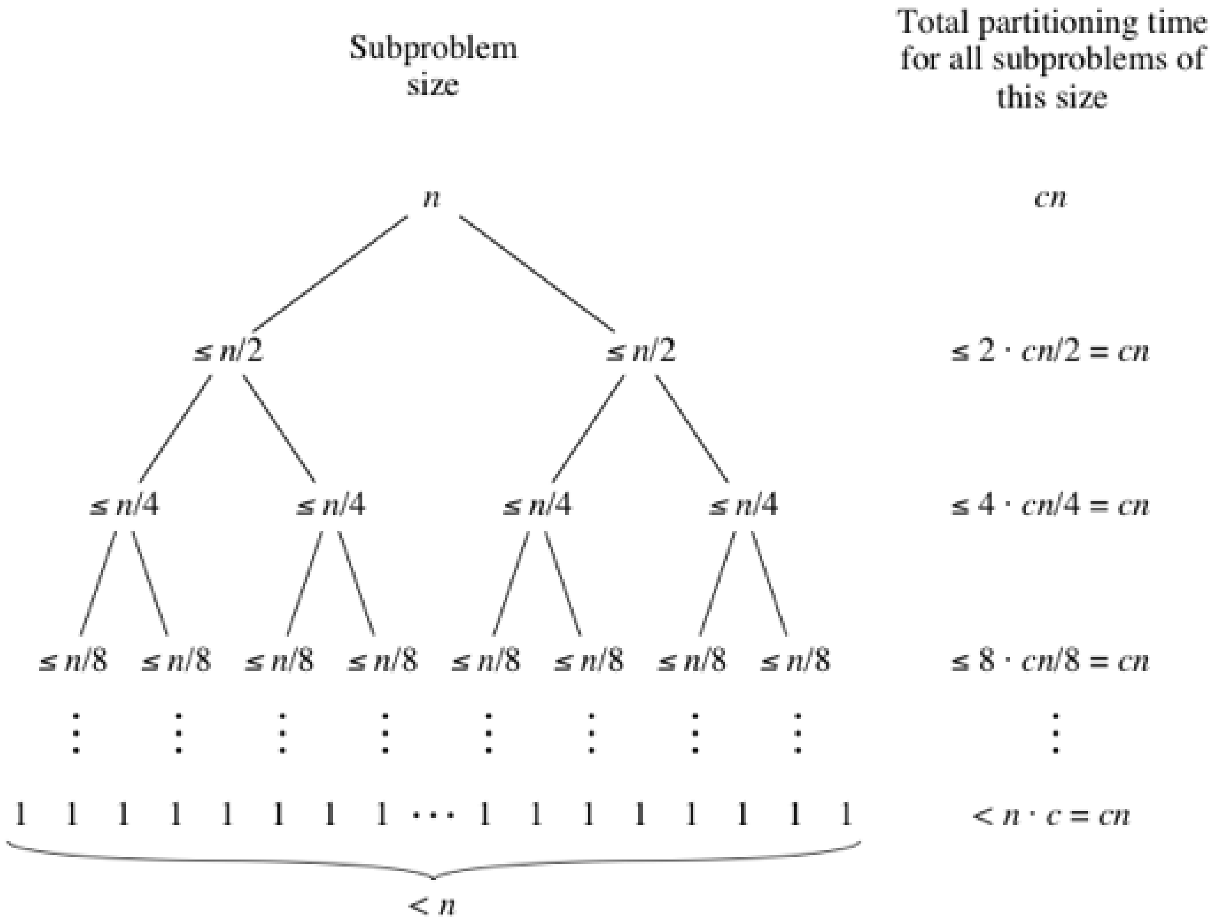
Divide-and-Conquer

QUICK SORT

- **Analysis of QUICK SORT**
- **Best-case partitioning**

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T((n-1)/2) + cn & \text{if } n > 1 \end{cases}$$

$\boxed{T(n) \text{ is } O(n\log n)}$



AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

QUICK SORT

- **Analysis of QUICK SORT**
- **Average-case partitioning**
  - Partition at the any element

    => Example: partitioning algorithm always produces a 9-to-1 proportional split
  - **Divide:** O(n)
  - **Conquer:** a=2, => T(n/10) + T(9n/10)

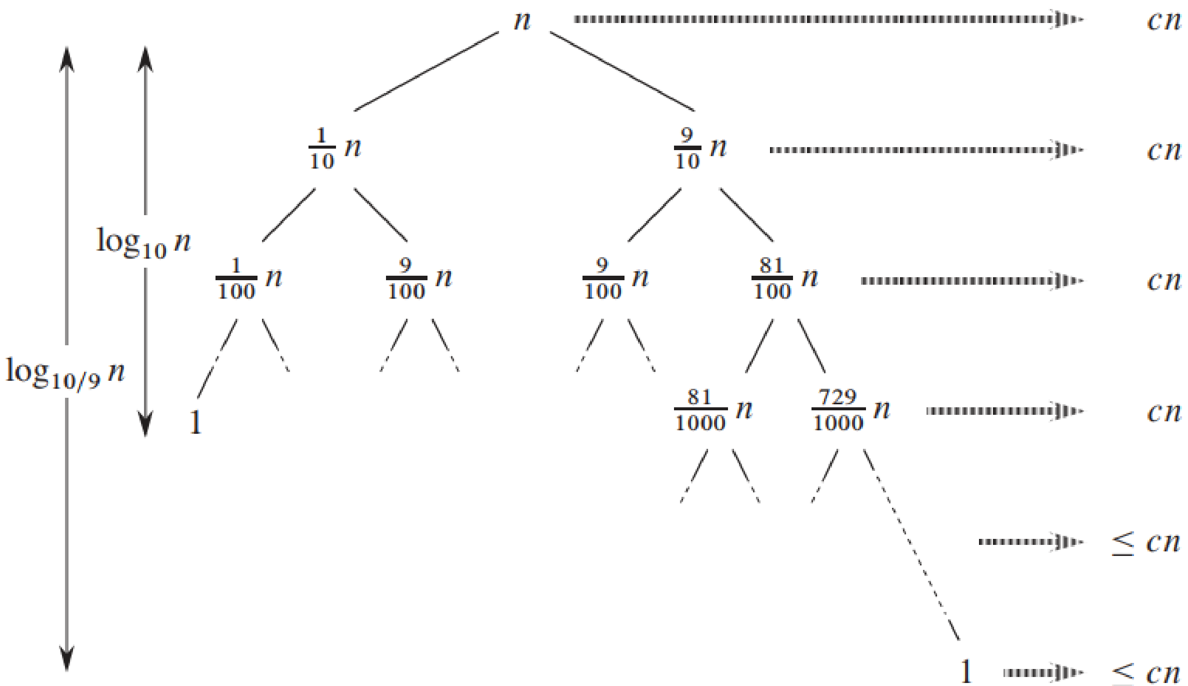
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n/10) + T(9n/10) + cn & \text{if } n > 1 \end{cases}$$

Divide-and-Conquer

QUICK SORT

- **Analysis of QUICK SORT**
- **Average-case partitioning**

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n/10) + T(9n/10) + cn & \text{if } n > 1 \end{cases}$$



T(n) is O(nlogn)

Divide-and-Conquer

QUICK SORT

```python
def partition(low, high, S):
    pivot, pointer = S[high], low

    for i in range(low, high):
        if S[i] <= pivot:
            S[i], S[pointer] = S[pointer], S[i]
            pointer += 1

    S[pointer], S[high] = S[high], S[pointer]

    return pointer
```

```python
def quicksort(low, high, S):
    if len(S) == 1:
        return S

    if low < high:
        p = partition(low, high, S)

        quicksort(low, p-1, S)
        quicksort(p+1, high, S)


S = [7, 2, 5, 30, 16, 8, 4, 17]
low = 0
high = len(S)-1
quicksort(low, high, S)
S
```

Base case

Divide: O(n)

Conquer:
$T(n/b_1)+T(n/b_2)$

Worst case: T(n) is $O(n^2)$
Best case: T(n) is O(nlogn)
Average case: T(n) is O(nlogn)

AI VIET NAM
@aivietnam.edu.vn

Divide-and-Conquer

QUICK SORT

- **Divide:** A sequence *S* is divided into subarrays by selection *x*: a pivot element (a specific element from *S*)

    L: elements less than pivot. R: elements greater than pivot

- **Conquer**
- **Pivot Selection**
    - The last element

2	4	5	7	8	16

    - The first element

2	4	5	7	8	16

    - Random

2	4	5	7	8	16

    - The median-of-three

2	4	5	7	8	16

Divide-and-Conquer

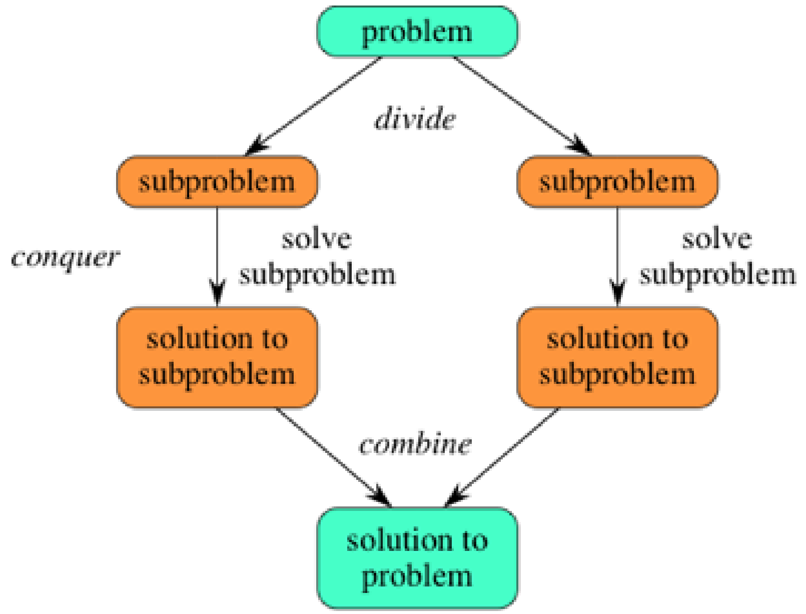
SUMMARY

- Three steps:
  - **Divide – Conquer – Combine**
- Analysis:

T(n) is the running time on a problem of size n

$$T(n) = \begin{cases} O(1) & \text{if } n \le n_c \\ aT(n/b) + D(n) + C(n) & \text{if } n > n_c \end{cases}$$

Use: Recursion-tree or Master Method



- Example
  - Merge Sort – O(nlogn)
  - Quick Sort – Worst: $O(n^2)$ – Best: O(nlogn) – Average: O(nlogn)

SUMMARY

ALGORITHM ANALYSIS

Algorithm Analysis

COMPUTATIONAL COMPLEXITY

Steps to calculate computational complexity

Python code → (1) → **Characterize Function** → (2) → **Asymptotic Notation**

```
[6]  S = [1, 2, 3]
     n = len(S)
     for i in range(n):
         for j in range(n):
             total = S[i] + S[j]
             print(total)
         print('- - - -')
```

$$T(n) = an^2 + bn + c$$

$$O(n^2)$$

$$T(n) = (c_3+c_4+c_5)n^2 + (c_2+c_3+c_6)n + (c_0+c_1+c_2)$$
$$\leq (c_0+c_1+2c_2+2c_3+c_4+c_5+c_6)n^2$$
$$= c'n^2$$

For $c' = c_0+c_1+2c_2+2c_3+c_4+c_5+c_6$, $n_0=1$

Algorithm Analysis

COMPUTATIONAL COMPLEXITY

- Running time complexity: The number of primitive operations that are performed
- A function *f(n)*: characterizes the number of primitive operations that are performed as a function of the input size *n*
- Most important functions:

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1 (c)	logn	n	nlogn	n^2	n^3	a^n

AI VIET NAM
@aivietnam.edu.vn

Algorithm Analysis

COMPUTATIONAL COMPLEXITY

- Asymptotic Analysis

  $f(n)$ is $O(g(n))$: $f(n) \le cg(n)$, for $n \ge n_0$

  $f(n)$ is $\Omega(g(n))$: $f(n) \ge cg(n)$, for $n \ge n_0$

  $f(n)$ is $\Theta(g(n))$: $c_1 g(n) \le f(n) \le c_2 g(n)$, for $n \ge n_0$

- Example:

  $n^{1/\log n}$ is $O(1)$

  $7\log n + 1$ is $O(\log n)$

  $4^{\log n} + 5n$ is $O(n^2)$

  $3n^2 - 2n\log n$ is $\Omega(n^2)$

  $3n\log n + 2^{\log n} + 5\log n$ is $\Theta(n\log n)$

Algorithm Analysis

COMPUTATIONAL COMPLEXITY

Example:

Bubble Sort

```
def bubble_sort(s):
    n = len(s)
    for step in range(n):
        for i in range(0, n-step-1):
            if s[i] > s[i+1]:
                temp = s[i]
                s[i] = s[i+1]
                s[i+1] = temp
```

Algorithm Analysis

COMPUTATIONAL COMPLEXITY

Example:

Bubble Sort

(Optimized)

```
def optimized_bubble_sort(s):
    n = len(s)
    for step in range(n):
        swapped = false
        for i in range(0, n-step-1):
            if s[i] > s[i+1]:
                temp = s[i]
                s[i] = s[i+1]
                s[i+1] = temp
                swapped = true
        if not swapped:
            break
```

Algorithm Analysis

COMPUTATIONAL COMPLEXITY

Example:

Bubble Sort

(Optimized)

```
def bubble_sort(s):
    n = len(s)
    for step in range(n):
        for i in range(0, n-step-1):
            if s[i] > s[i+1]:
                temp = s[i]
                s[i] = s[i+1]
                s[i+1] = temp
```

```
def optimized_bubble_sort(s):
    n = len(s)
    for step in range(n):
        swapped = false
        for i in range(0, n-step-1):
            if s[i] > s[i+1]:
                temp = s[i]
                s[i] = s[i+1]
                s[i+1] = temp
                swapped = true
        if not swapped:
            break
```

Algorithm Analysis

COMPUTATIONAL COMPLEXITY

Example:

Bubble Sort

(Optimized)

```
def bubble_sort(s):
    n = len(s)
    for step in range(n):
        for i in range(0, n-step-1):
            if s[i] > s[i+1]:
                temp = s[i]
                s[i] = s[i+1]
                s[i+1] = temp
```

$O(n^2)$

```
def optimized_bubble_sort(s):
    n = len(s)
    for step in range(n):
        swapped = false
        for i in range(0, n-step-1):
            if s[i] > s[i+1]:
                temp = s[i]
                s[i] = s[i+1]
                s[i+1] = temp
                swapped = true
        if not swapped:
            break
```

Best case: $O(n)$
Average case: $O(n^2)$
Worst case: $O(n^2)$

SUMMARY

ALGORITHM DESIGN

Algorithm Design

ALGORITHM

- **Brute Force:** based on problem statement and definitions
- **Recursion:** function makes one or more calls to itself during execution
- **Divide-and-Conquer:** divide (problem => subproblems), conquer: recursively, combine (subproblems => original problem)
- **Two pointer (Technique):** The idea here is to iterate two different parts of the array simultaneously to get the answer faster

AI VIET NAM
@aivietnam.edu.vn

Algorithm Design

SEARCHING PROBLEM

Input: a sorted sequence of n number <a_1, a_2,…, a_n>, key

Output: index of key in the sequence if exist, -1 if not exist

	Searching Algorithm	Time Complexity		
		Best Case	Average Case	Worst Case
Brute Force	Linear Search	O(1)	O(n)	O(n)
Recursion	Binary Search	O(1)	O(logn)	O(logn)

AI VIET NAM
@aivietnam.edu.vn

Algorithm Design

SORTING PROBLEM

Input: a sequence of n number $<a_1, a_2, \ldots, a_n>$

Output: a permutation (reordering) $<a'_1, a'_2, \ldots, a'_n>$; such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

	Sorting Algorithm	Time Complexity		
		Best Case	Average Case	Worst Case
Brute Force	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Brute Force	Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Brute Force	Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Brute Force	Optimized Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
DC	Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
DC	Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$

EXAM

- Zoom: 08:00 P.M 25/06/2022
- Time: 90 mins
- Submission file (colab-ipynb)
- Contents:
  - Q1: Algorithm Analysis (Asymptotic Analysis)
  - Q2: Algorithm Analysis (Step by step)
  - Q3: Algorithm Analysis (Compute T(n) using recursion tree and master method)
  - Q4: Algorithm Design (Without code)
  - Q5: Algorithm Design (With code)
  - Q6: Algorithm Design (Improve code)

Reference

(1) Introduction to Algorithms, 3rd Edition; Thomas H.Cormen et al; 2009

(2) Data Structures & Algorithms; Michael T.Goodrich et al; 2013

(3) Algorithms, 4th; Robert Sedgewick et al; 2011

Thanks!

Any questions?