Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Algorithms

What are algorithms?

Why do we need them?

Algorithm

- An algorithm is a ***<u>well-defined computational procedure</u>*** that takes some input values and produces output values ***<u>for a computation problem</u>***.
  - ***<u>An algorithm</u>*** is a sequence of well-defined ***<u>computational steps</u>*** that transform the input into the output.

- Ex. Computing the area of a circle.

1. $r \leftarrow$ radius (read the radius).
2. $area \leftarrow \pi * r^2$
3. Output $area$ as the answer.

We call this a "***<u>Pseudo-code</u>***" of the algorithm.

"Pseudo-code"
– A **complete description** on the ***<u>steps</u>*** of the computation procedure.

Sorting

- Given n numbers $a_1, a_2, \dots, a_n$,
  sort and output the numbers in non-descending order.

- Ex.

  For the input $3,\ 2,\ 8,\ 6,\ 10,\ 12,\ 2,\ 1,$

  You should output $1,\ 2,\ 2,\ 3,\ 6,\ 8,\ 10,\ 12$ in order.

Insertion Sort Algorithm - A Friendly Description

- InsertionSort($A[\ 1, 2, \ldots, n\ ], n$)

A. For $j \leftarrow 2$ to n, do the following.

   a) Find the largest index $i \in [1,2, \ldots, j-1]$ such that $A[i] < A[j]$.
      Set $i \leftarrow 0$ if no such index exists.

   b) Insert $A[j]$ at position $i+1$
      by moving $A[i+1, \ldots, j-1]$ to $A[i+2, \ldots, j]$.

A More Detailed Pseudo-Code

- InsertionSort($A[\,1, 2, \ldots, n\,], n$)

A. For $j \leftarrow 2$ to n, do the following.

   a) $key \leftarrow A[j]$.

   b) $i \leftarrow j - 1$.

   c) While $i > 0$ and $A[i] > key$, do the following.

      1) $A[i + 1] \leftarrow A[i]$.

      2) $i \leftarrow i - 1$.

   d) $A[i + 1] \leftarrow key$.

In this form, however,
they are **hard to read & follow**.

A (Perhaps) Even More Friendly Way

- InsertionSort($A[\,1, 2, \dots, n\,], n$)

A. For $j \leftarrow 2$ to n, do the following.

   // Consider $A[1, 2, \dots, j]$ as the segment that is already sorted in order

   a) Find the "correct" position for $A[j]$ in the sorted segment.

   b) Insert $A[j]$ at the position it should be.

This description is **intuitive** and **easy to understand**.
However, some parts are not 100% precise as before.

Algorithm Description

- An algorithm should be described in a way ***precise enough*** for a human being ***to verify & to understand***.
  - "*When is it precise enough?*" is a philosophical question whose answer depends on the actual scenario.

- A detailed description is generally very precise, but…
  - You don't want to make it hard for others to follow. Otherwise, he/she ***may not have the patience*** to read / to listen.

Please keep this in mind when preparing your answers for HW / Exams.

Algorithm Description

- In general, a good algorithm description is one that describes the computational steps *intuitively* and *precisely* ***in a concise way***.

Correctness &

Time / Space Complexity of an Algorithm

Correctness of an Algorithm

- An algorithm is a ***well-defined computational procedure*** that takes some input values and produces output values ***for a computation problem***.

- As a mean for solving a computation problem,
  it is essential to ensure that the procedure always produces a ***correct answer for every possible set of inputs***.
  - A ***rigid proof*** is usually required to prove the correctness of an algorithm.

Correctness of Insertion Sort Algorithm

- InsertionSort($A[\,1,2,\dots,n\,], n$)

A. For $j \leftarrow 2$ to n, do the following.

   a) Find the largest index $i \in [1,2,\dots,j-1]$ such that $A[i] < A[j]$.
      Set $i \leftarrow 0$ if no such index exists.

   b) Insert $A[j]$ at position $i+1$
      by moving $A[i+1,\dots,j-1]$ to $A[i+2,\dots,j]$.

- InsertionSort($A[\,1, 2, \dots, n\,], n$)

A. For $j \leftarrow 2$ to n, do the following.

   a) Find the largest index $i \in [1,2,\dots,j-1]$ such that $A[i] < A[j]$.
      Set $i \leftarrow 0$ if no such index exists.

   b) Insert $A[j]$ at position $i+1$
      by moving $A[i+1,\dots,j-1]$ to $A[i+2,\dots,j]$.

It suffices to prove the following lemma.

Lemma. (The **Invariant condition** of the algorithm)

At the end of each for-loop in step A.,
the numbers in $A[1,2,\dots,j]$ are always sorted in order.

Time Complexity (Efficiency) of an Algorithm

- The running time / time complexity / efficiency of an algorithm is the number of ***"logical atomic computation steps"*** it takes to compute the answer for the input instance.
  - As the number of steps may vary with different input instances, one primary measure is to consider the

    "worst-case running time"

    of the algorithm.
  - This is usually measured ***in terms of the size of the input instance***.

Running Time of Insertion Sort

- InsertionSort($A[\,1, 2, \dots, n\,], n$)

A. For $j \leftarrow 2$ to n, do the following.

   a) Find the largest index $i \in [1,2, \dots, j-1]$ such that $A[i] < A[j]$.
      Set $i \leftarrow 0$ if no such index exists.

   b) Insert $A[j]$ at position $i+1$
      by moving $A[i+1, \dots, j-1]$ to $A[i+2, \dots, j]$.

Running Time of Insertion Sort

- The *worst-case running time* of insertion sort on n input numbers is

$$\sum_{2 \le j \le n} 2 \cdot (j-1) = n(n-1) = O(n^2).$$

This says, "roughly at most n^2".
We will define what this means next lecture.

  - The original pseudo-code takes $n(n-1)/2$.
    Here we use the version that is easier to understand.
  - ***The analysis is tight***, as there is indeed an instance that makes InsertionSort to take this number of steps.

Can you point out one of such instances?

Algorithms

What are algorithms?

Why do we need Algorithms?

Why do we need (Better) Algorithms?

- **As a mean of solving practical problems efficiently**.
  - Consider the following computation problem.

> Sort the IDs of all Taiwanese citizens according to alphabetical order.

  - The number of legal citizens in Taiwan is roughly 2.3×10^7.
  - If we use the InsertionSort algorithm,
    it'd take ***more than a week*** to sort all the IDs.
  - However, with a more cleaver algorithm,
    we can do this in ***less than 5 secs***.

Why do we need (Better) Algorithms?

- **As a mean of solving practical problems efficiently**.
  - If we use the InsertionSort algorithm,
    it'd take ***more than a week*** to sort all the IDs.
  - However, with a more cleaver algorithm,
    we can do this in ***less than 5 secs***.

- Good algorithms are indispensable **in time-critical applications**.
  - Google maps, navigation systems,
    train scheduling systems, flight scheduling systems, etc.

The Merge-Sort Algorithm

The Merge-Sort Algorithm

- Let $a_1, a_2, \ldots, a_n$ be the input numbers.

- The merge-sort algorithm works as follows.

1. Partition the input numbers into two subsets

$$L = \{a_1, \ldots, a_{\lfloor n/2 \rfloor}\} \text{ and } R = \{a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n\}$$

   of ***roughly equal sizes***.

2. Sort L and R *(recursively)* ***using Merge-Sort algorithm***.

3. Merge L and R into a sorted list.

The Merge-Sort Algorithm

- A more detailed pseudo-code for this algorithm.

Algorithm MergeSort($A[1,2,\ldots,n], \text{left}, \text{right}$)

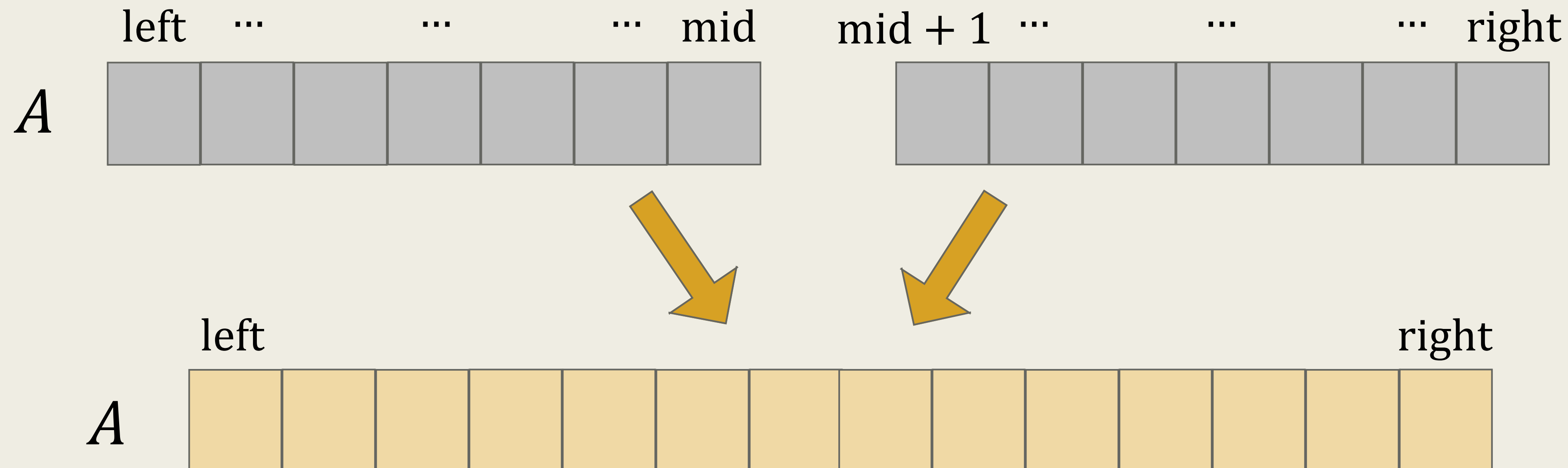
1. If $\text{left} = \text{right}$, then return.
2. Let $\text{mid} \leftarrow \lfloor(\text{left} + \text{right})/2\rfloor$.
3. Call MergeSort(A, left, mid) and MergeSort(A, mid+1, right).
4. Merge $A[\text{left},\ldots,\text{mid}]$ and $A[\text{mid}+1,\ldots,\text{right}]$ with the procedure Merge($A, \text{left}, \text{mid}, \text{right}$).

The Procedure Merge($A, \mathrm{left}, \mathrm{mid}, \mathrm{right}$)

- The procedure takes two sorted lists

$$L \coloneqq A[\mathrm{left}, \dots, \mathrm{mid}] \quad \text{and} \quad R \coloneqq A[\mathrm{mid}+1, \dots, \mathrm{right}]$$

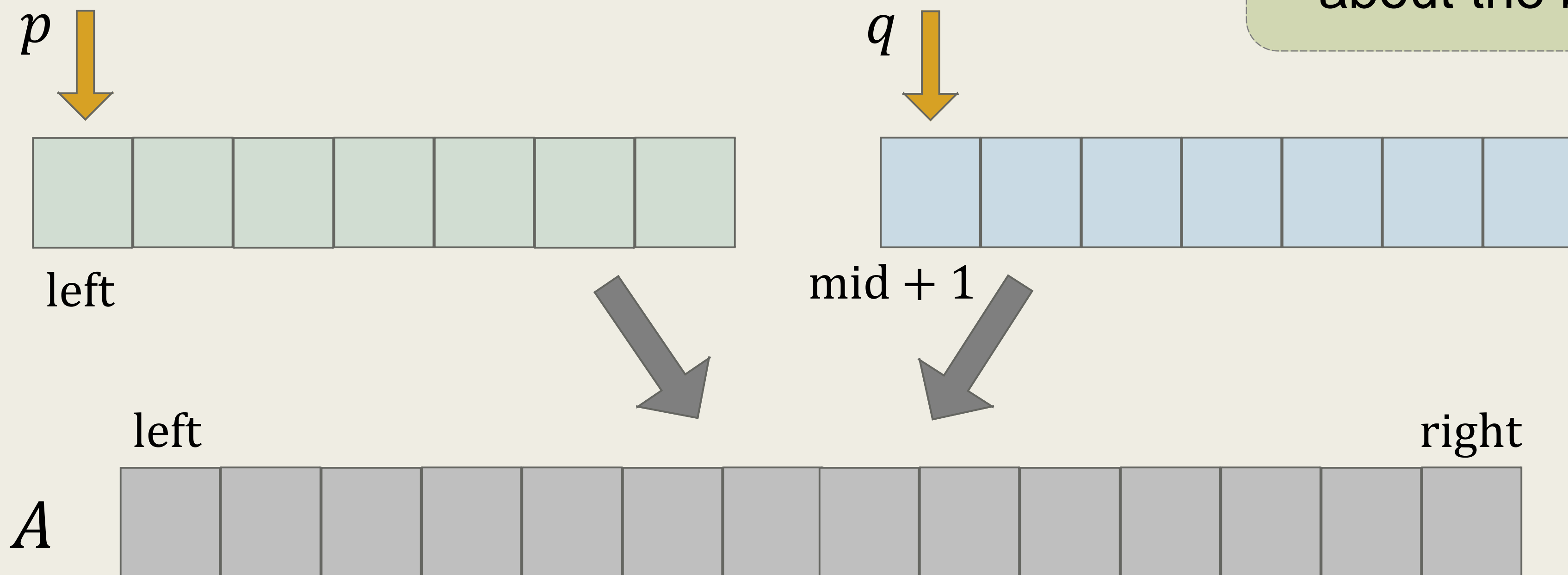
and merge them into one sorted list $A[\mathrm{left}, \dots, \mathrm{right}]$.



- The procedure uses two pointers p and q to iterate over L and R.
  - In each iteration, it picks the smaller between p and q to the new sequence and advances it.
  - Repeats until L and R are scanned.

The idea is simple.

Have to be careful about the boundary cases.



- The procedure Merge(·) uses an extra array $\mathrm{temp}[1, \dots, n]$.

<u>Procedure Merge</u>($A[1,2, \dots, n], \mathrm{left}, \mathrm{mid}, \mathrm{right}$)

1. Copy $A[\mathrm{left}, \dots, \mathrm{right}]$ to $\mathrm{temp}[\mathrm{left}, \dots, \mathrm{right}]$.

   $p \leftarrow \mathrm{left},\ q \leftarrow \mathrm{mid} + 1,\ \mathrm{pos} \leftarrow \mathrm{left}$.

2. While $p \leq \mathrm{mid}$ and $q \leq \mathrm{right}$, do the following.

   - If $\mathrm{temp}[p] < \mathrm{temp}[q]$, then set $A[\mathrm{pos}++] \leftarrow \mathrm{temp}[p++]$.

     Otherwise, set $A[\mathrm{pos}++] \leftarrow \mathrm{temp}[q++]$.

3. While $p \leq \mathrm{mid}$, set $A[\mathrm{pos}++] \leftarrow \mathrm{temp}[p++]$.

4. While $q \leq \mathrm{right}$, set $A[\mathrm{pos}++] \leftarrow \mathrm{temp}[q++]$.

Analysis of the Procedure Merge(·)

- Why is this procedure correct?
  - Provided that L and R are already sorted,
    the smaller of $\mathrm{temp}[p]$ and $\mathrm{temp}[q]$ must be the smallest element among $\mathrm{temp}[p, \ldots, \mathrm{mid}]$ and $\mathrm{temp}[q, \ldots, \mathrm{right}]$.

- The time complexity of this procedure is

$$2 \cdot (\mathrm{right} - \mathrm{left} + 1) = O(\mathrm{right} - \mathrm{left} + 1),$$

  i.e., linear in the number of elements.

Analysis of the Algorithm MergeSort(·)

- Why is this algorithm correct?
  - Proved by induction on $m \coloneqq \mathrm{right} - \mathrm{left} + 1$.
  - When $m = 1$, the procedure MergeSort(A, left, right) clearly sorts $A[\mathrm{left}]$ correctly.
  - When $m > 1$,
    by induction hypothesis, MergeSort sorts L and R correctly.
    Then, we have shown that the procedure Merge(·) merges L and R into a sorted list.

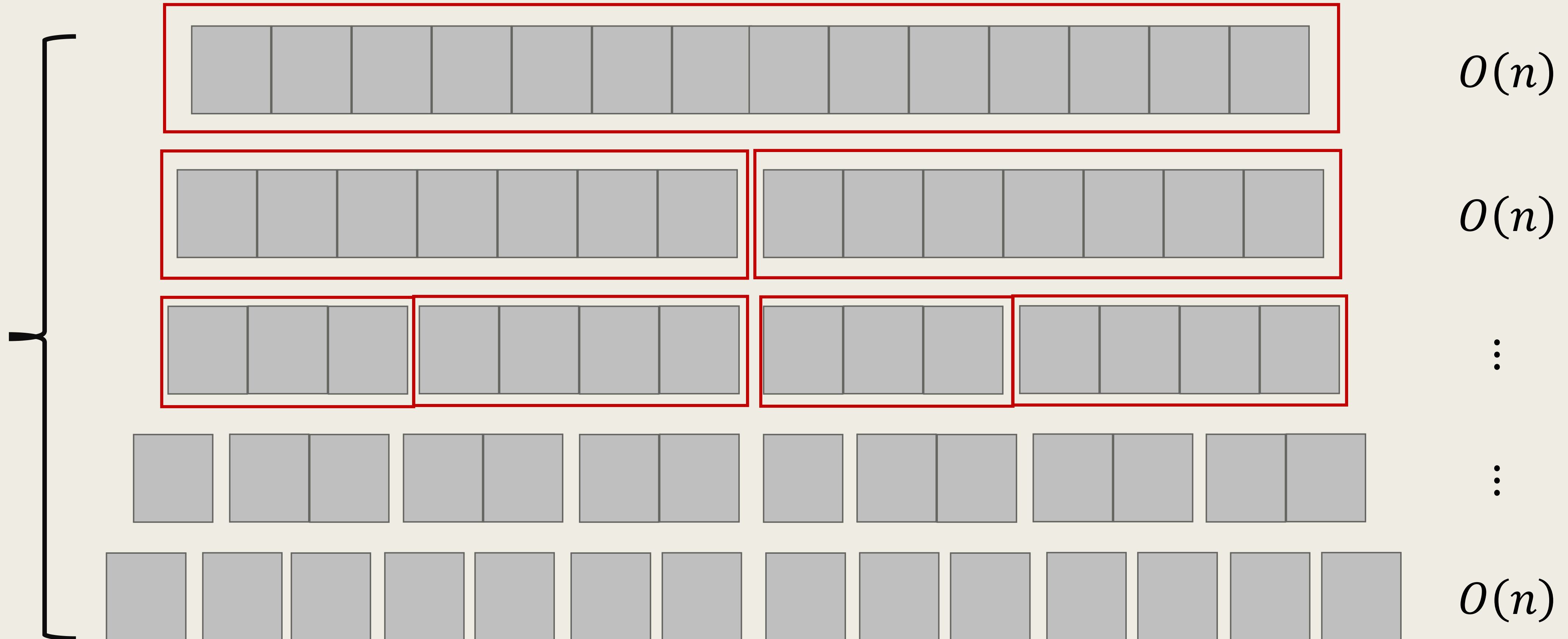
Analysis of the Algorithm MergeSort(·)

- Time complexity of Merge-Sort.
  - For any $n \geq 1$,
    let $T(n)$ be the number of steps required by MergeSort algorithm.
  - Then, we have

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1, \\ 2 \cdot T\left(\frac{n}{2}\right) + O(n), & \text{otherwise.} \end{cases}$$

$O(\log n)$ levels in total

Total time taken by Merge(·)



In total, it takes $O(n \log n)$ time.

Analysis of the Algorithm MergeSort(·)

- Time complexity of Merge-Sort.
  - For any $n \geq 1$,
    let $T(n)$ be the number of steps required by MergeSort algorithm.
  - Then, we have

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1, \\ 2 \cdot T\left(\frac{n}{2}\right) + O(n), & \text{otherwise.} \end{cases}$$

And $T(n) = O(n \log n)$.

The Divide-and-Conquer Paradigm

- The merge-sort algorithm illustrates the usage of a classic *algorithm design paradigm*

  Divide-and-Conquer.

- The divide-and-conquer is a *powerful technique* commonly used ***for designing efficient algorithms***.

  It consists of three steps.

  - ***Divide*** –
    to divide the problem instance into sub-instances of smaller sizes.
  - ***Conquer*** – to conquer the sub-instances separately.
  - ***Merge*** –
    to merge the answer of the sub-instances for the original instance.

Divide-n-Conquer in Merge-Sort

We will see more examples in future lectures.

- A more detailed pseudo-code for this algorithm.

Algorithm MergeSort($A[1,2,\ldots,n]$, left, right)

1. If $\text{left} = \text{right}$, then return.
2. Let $\text{mid} \leftarrow \lfloor(\text{left} + \text{right})/2\rfloor$.
3. MergeSort(A, left, mid) and MergeSort(A, mid+1, right).
4. Merge $A[\text{left},\ldots,\text{mid}]$ and $A[\text{mid}+1,\ldots,\text{right}]$ with the procedure Merge(A, left, mid, right).

Dividing the instance into two halves.

Conquer them separately.

Merge the results.