Algorithm Analysis

โรงเรียนสามเสนวิทยาลัย – มหาวิทยาลัยธรรมศาสตร์

คอมพิวเตอร์โอลิมปิก สอวน.

Pakkaporn Saophan, Ph.D.
อาจารย์ ดร.ภัคพร เสาร์ฝั้น (หงส์)

Ph.D. in Knowledge Science

Japan Advanced Institute of Science and Technology (JAIST), Japan

M.Sc. in Management Mathematics

Sirindhorn International Institute of Technology (SIIT), Thammasat University

B.Sc. in Management Mathematics

Second Class Honors, Thammasat University

pakkp@tu.ac.th

RESEARCH INTEREST

Optimization, Management Mathematics, Machine Learning

OUTLINE

- Introduction
- Algorithmic Complexity / Asymptotic Notation
- Algorithm
  - Brute Force
  - Divide-and-Conquer
  - Decrease-and-Conquer
  - Transform-and-Conquer
  - String Matching Algorithms
  - Backtracking & Branch-and-Bound
  - Greedy Algorithms
  - Graph Algorithms
  - Dynamic Programming

Algorithm

Goals

- Learn several algorithm design techniques by studying a standard set of important algorithms from different areas of computing (different problem types)
- Discuss the general framework for efficiency analysis
- So that,
    - students can develop algorithms on their own when given a computational. problem
    - students can formally argue the correctness of their algorithms.
    - students can analyze the efficiency of their algorithms.

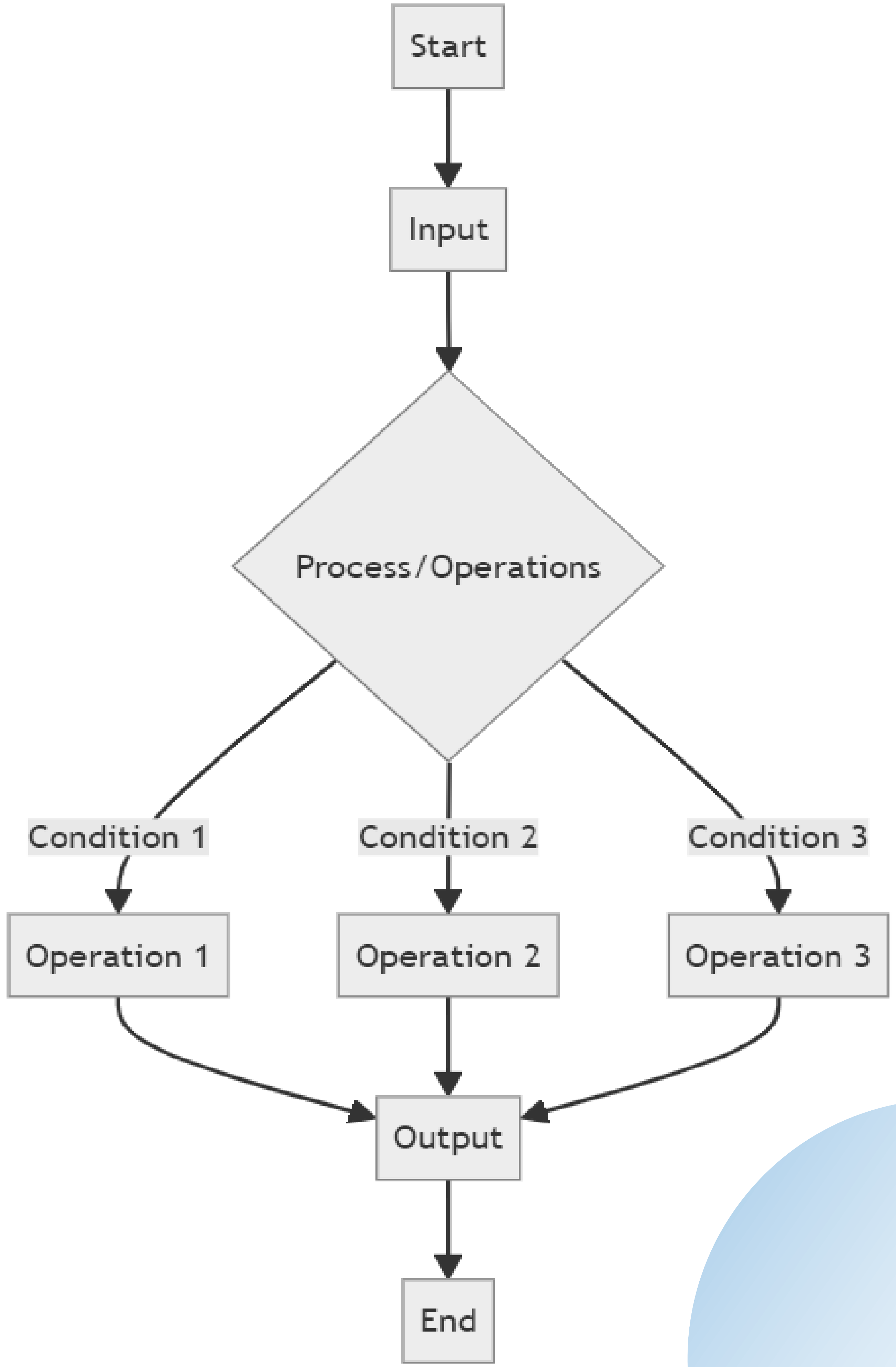
Algorithm

- An algorithm is a process that consists of various steps or a **set of instructions** designed to perform a specific task or **solve a particular problem**.
- An algorithm can be seen as a sequence of clear instructions, which, when executed correctly, will **lead to solving the problem or completing the task as specified.**
- The key characteristics of an algorithm are as follows:

1. **Finiteness**: An algorithm must have a limited number of steps and must terminate after a certain period (with conditions to stop).
2. **Definiteness**: Each step in the algorithm must be clearly defined and should not be open to interpretation in different forms.

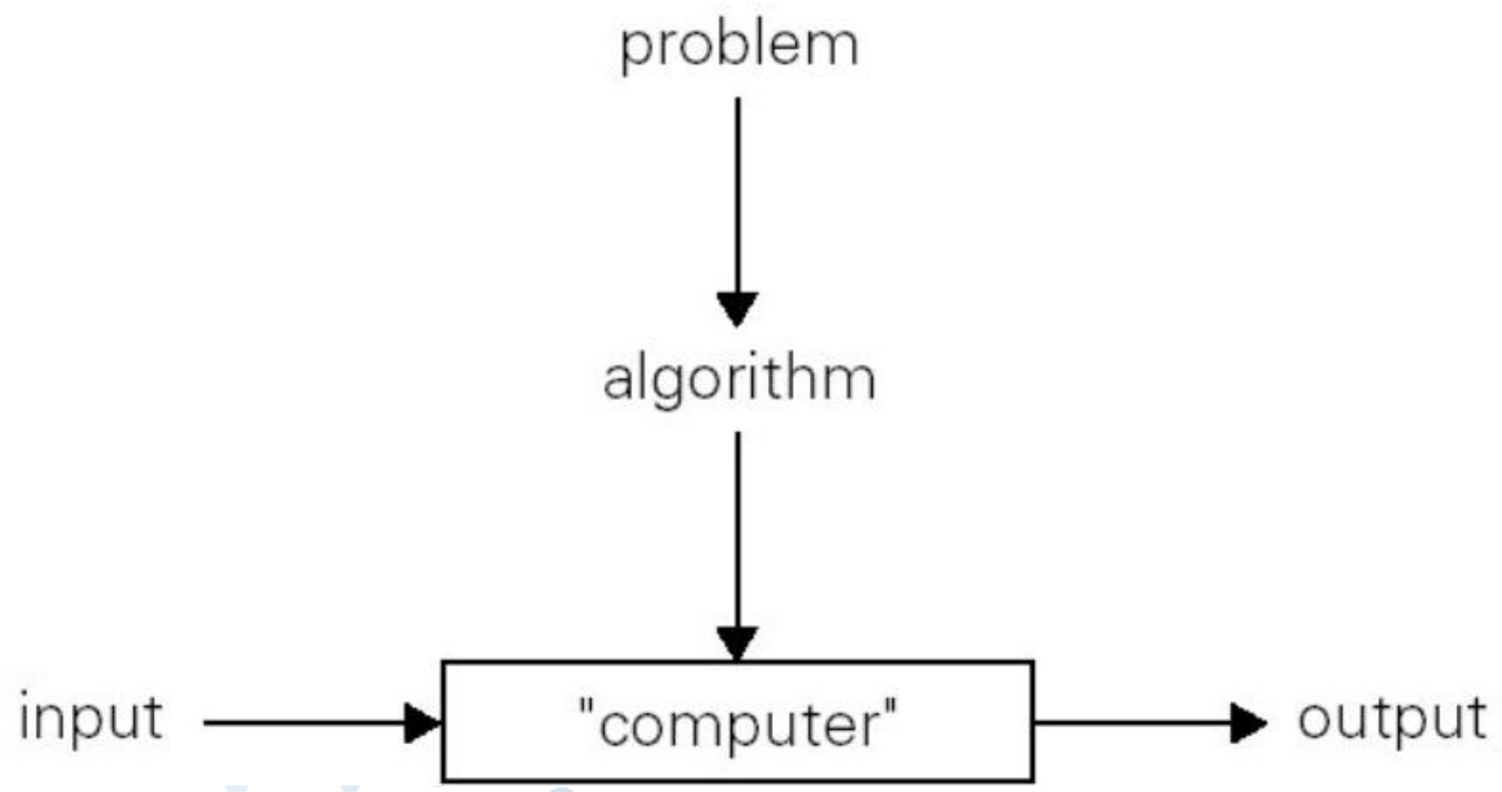
Algorithm

3. **Input:** An algorithm may require at least one input (data from the user), which is the information or initial value used for processing.
4. **Output:** After processing, the algorithm should produce at least one output, which is the desired result.
5. **Effectiveness:** The algorithm should produce the correct results for the given input, achieving the desired output according to the problem.



Algorithm

- Algorithms are used in a wide range of fields, including computer science, mathematics, engineering, and others.
- They are a fundamental part of computer programming and are crucial for enabling computers to solve complex problems, perform calculations, manage data processing, or automate tasks in daily life.



Algorithm

Computational Problems: *GCD(m, n)*

- Finding the greatest common divisor (gcd) of two non-negative, not both zero integers
- **Problem Statement:**
    - Input: two non-negative, not both zero integers called *m* and *n*, where $m \geq n$
    - Output: the greatest common divisor of *m* and *n*
- A problem instance: (18, 12), (60, 24), (12, 0)
- **Algorithms:** The intuitive algorithm, Euclid’s Algorithm

Algorithm

Computational Problems: *GCD(m, n)*

The Consecutive Integers Checking Algorithm

- Intuitively based on the definition of gcd

Algorithm gcd(m, n)

```
If n = 0, return m and stop.
Let g = n
While g > 1 do:
    If both m and n are divisible by g, return g and stop.
    Otherwise, decrease g by one.
Return g.
```

Algorithm

Computational Problems: *GCD(m, n)*

Euclid's Algorithm

- Formula:

  $\gcd(m,n) = \gcd(n, m \bmod n)$

  $\gcd(m,0) = m$

- Example Calculation: gcd(18,12)

  = gcd(12,6)

  =gcd(6,0)

  = 6

Algorithm gcd(m, n):

```
While n ≠ 0 do:
        r = m mod n
        m = n
        n = r
Return m.
```

Algorithm

Example: Calculating Fibonacci Sequence

- Fibonacci sequence
  - 0,1,1,2,3,5,8,13,21,34,55,89,144,…

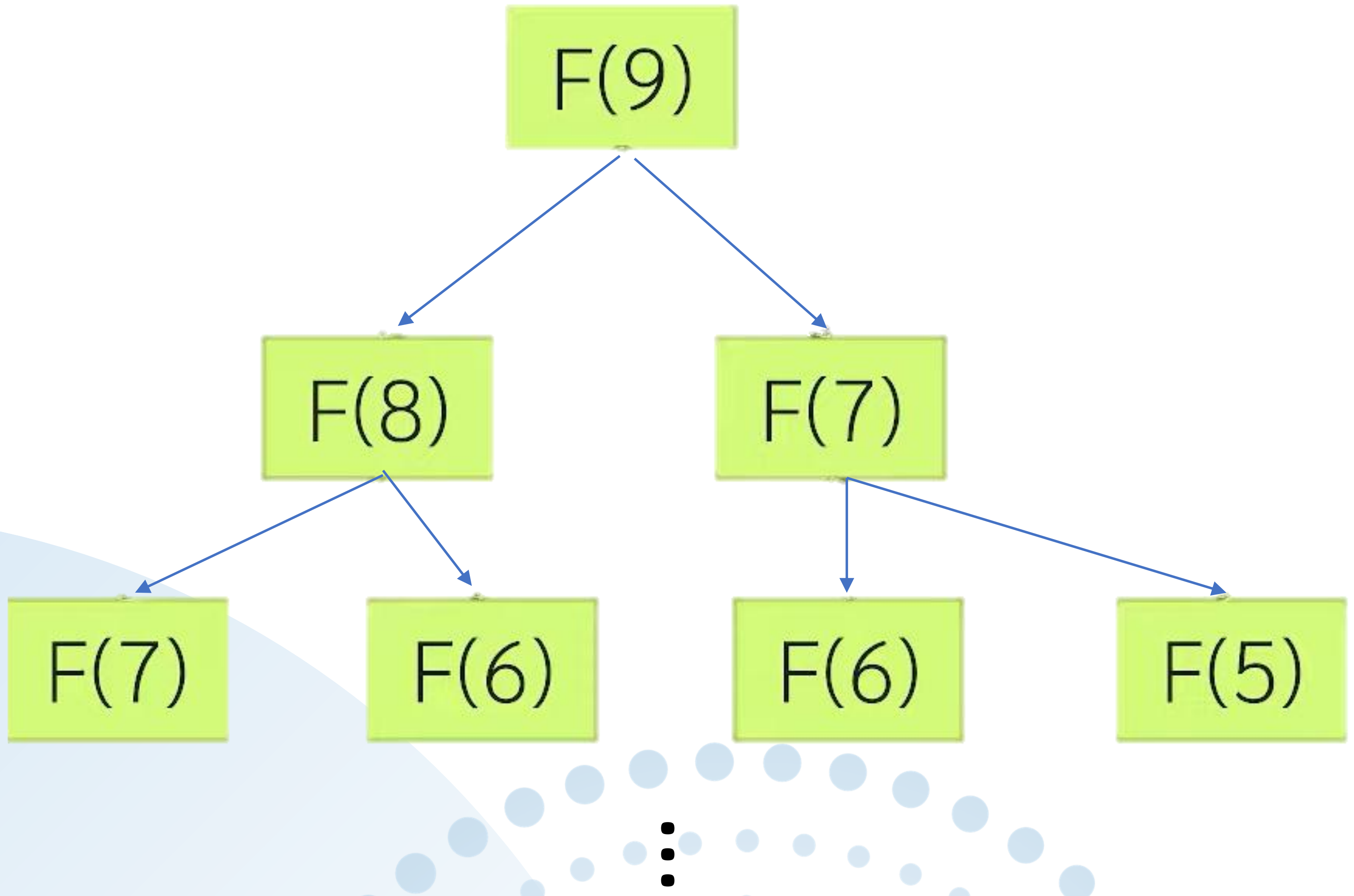
$$F_n = \begin{cases} 0 & ; n = 0 \\ 1 & ; n = 1 \\ F_{n-1} + F_{n-2} & ; n > 1 \end{cases}$$

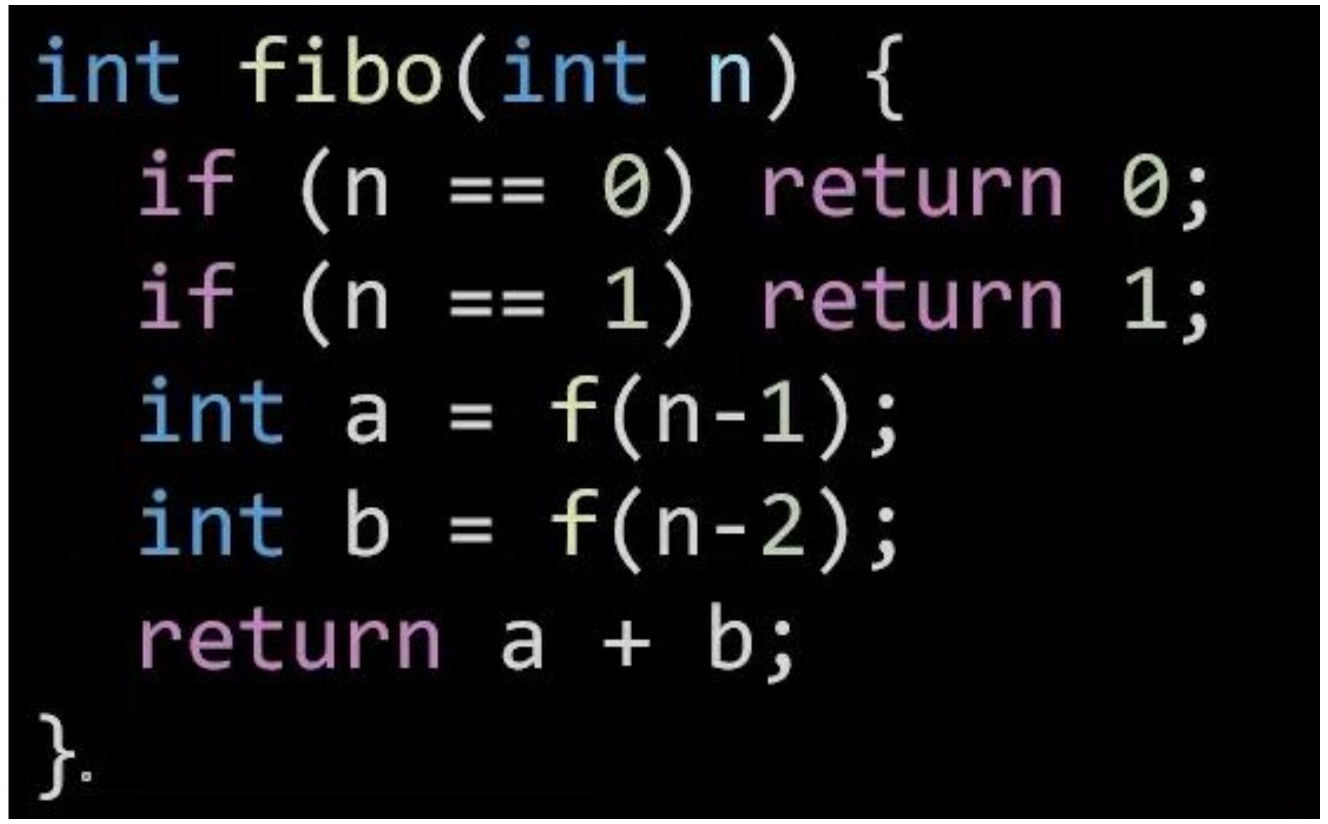
  - Input: a non-negative integer N
  - Output: F_n (the n^{th} Fibonacci Number)
  - What is N = 10, N = 15 ?

Algorithm

Example: Calculating Fibonacci Sequence

- Approach 1: Recursive, $O(2^n)$



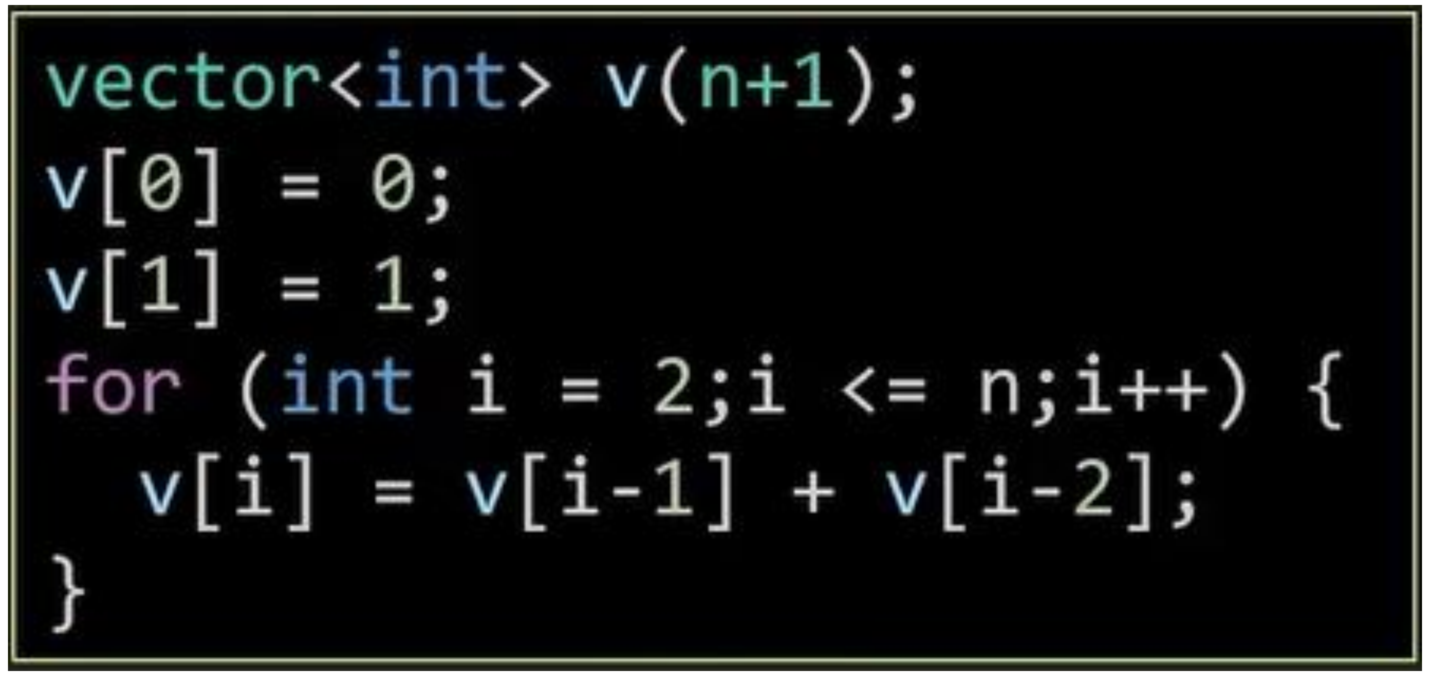


```
int fibo(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  int a = f(n-1);
  int b = f(n-2);
  return a + b;
}.
```

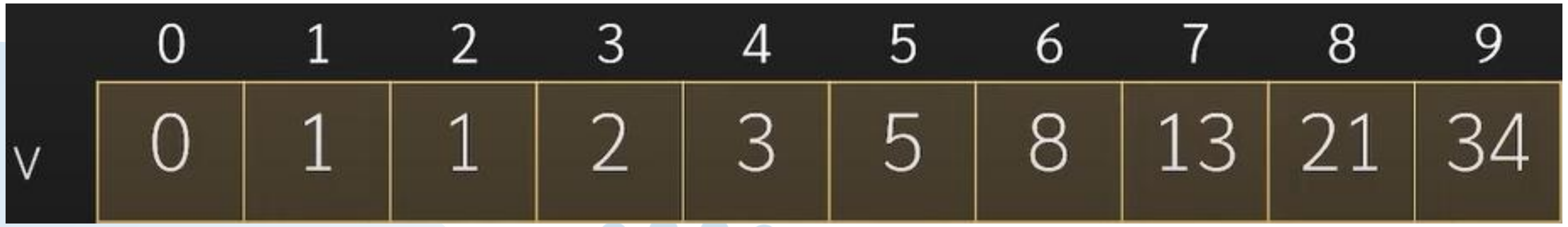
Algorithm

Example: Calculating Fibonacci Sequence

- Approach 2: Dynamic Programming, $O(n)$



```
vector<int> v(n+1);
v[0] = 0;
v[1] = 1;
for (int i = 2;i <= n;i++) {
  v[i] = v[i-1] + v[i-2];
}
```

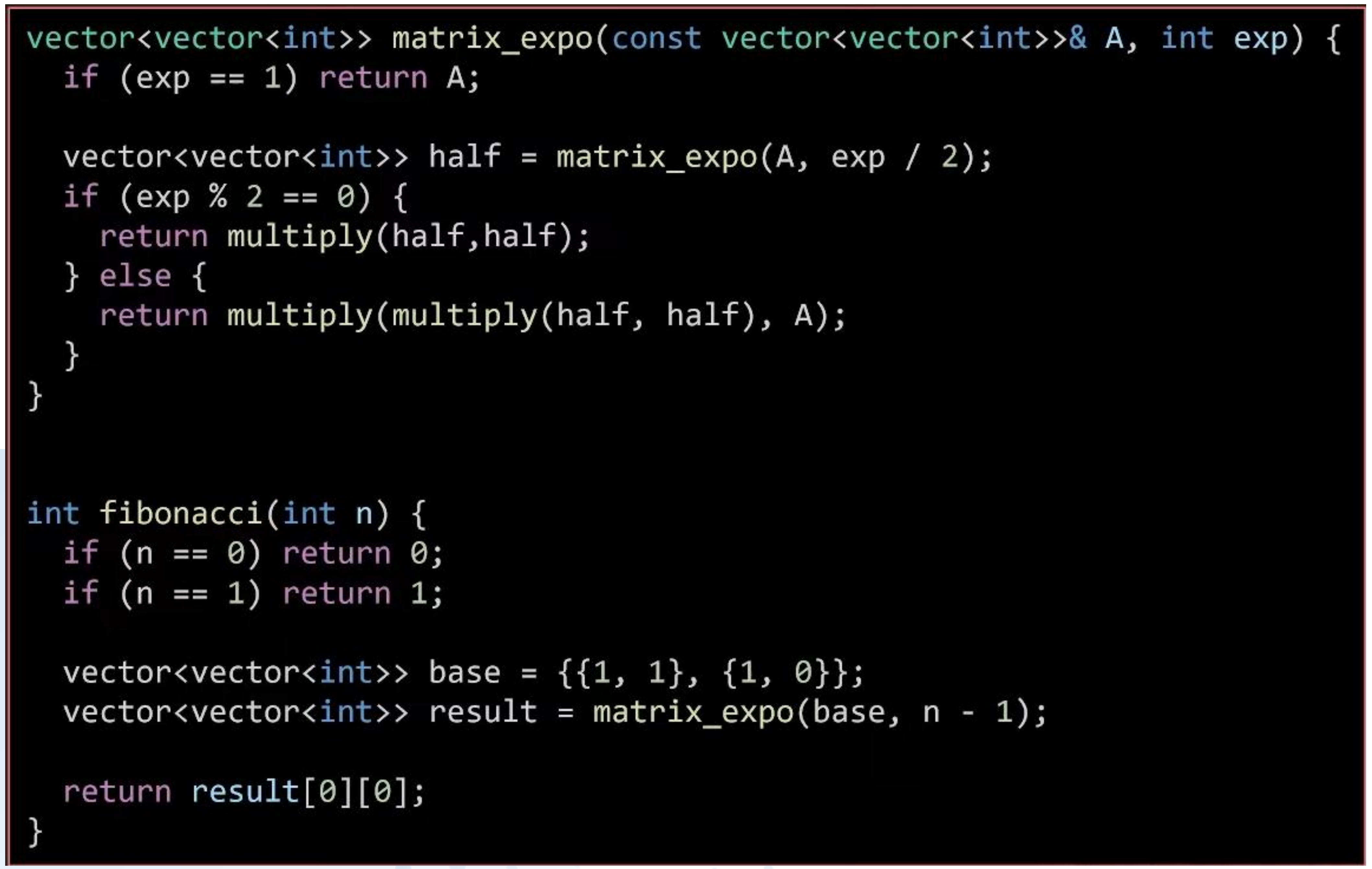


	0	1	2	3	4	5	6	7	8	9
V	0	1	1	2	3	5	8	13	21	34

Algorithm

Example: Calculating Fibonacci Sequence

- Approach 3: Divide and Conquer, $O(\log n)$



```
vector<vector<int>> matrix_expo(const vector<vector<int>>& A, int exp) {
  if (exp == 1) return A;

  vector<vector<int>> half = matrix_expo(A, exp / 2);
  if (exp % 2 == 0) {
    return multiply(half,half);
  } else {
    return multiply(multiply(half, half), A);
  }
}


int fibonacci(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;

  vector<vector<int>> base = {{1, 1}, {1, 0}};
  vector<vector<int>> result = matrix_expo(base, n - 1);

  return result[0][0];
}
```

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Algorithm

Example: Calculating Fibonacci Sequence

- Conclusion
  - Different design → Difference performance
  - This class emphasizes on designing efficient algorithm

Algorithm

Here are some examples of algorithms:

- **Sorting algorithms** (e.g., *bubble sort, merge sort, quicksort*) are used to arrange data in a desired order, such as sorting numbers in ascending order or arranging letters alphabetically.
- **Search algorithms** (e.g., *linear search, binary search*) are used to find specific data within a data structure, such as searching for a person's name in a list or finding a product in an online store system.
- **Encryption algorithms** (e.g., *AES, RSA*) are used to secure data and communications by converting the original information into a format that is not easily readable, helping to prevent data theft during transmission.

Algorithm

Here are some examples of algorithms:

- **Pathfinding algorithms** (e.g., *Dijkstra's algorithm, A* algorithm*) are used to find the shortest path between two points, such as finding a route on a navigation map or determining the sequence of moves in a chess game.
- **Compression algorithms** (e.g., *Huffman coding, LZW*) are used to reduce the size of data by replacing the original data with a smaller representation, helping to save storage space and increase data transmission speed.

Algorithms can be expressed in various forms, such as **pseudocode**, **flowcharts**, or different programming languages. The key challenge in algorithm design is to create an efficient algorithm, meaning one that uses the least amount of resources possible (e.g., time complexity, and space complexity).

Sort Algorithm

- **Searching Algorithms**: For instance, the Binary Search algorithm necessitates a pre-sorted list to execute the search effectively.
- **Data Processing and Analysis**: Organizing data through sorting significantly facilitates and accelerates data processing and analysis.
- **Database Operations**: Databases typically store data in a sorted order to enhance the efficiency of search and retrieval operations, effectively integrating sorting with search algorithms.

Sort Algorithm

Types of Sorting Algorithms

Sorting algorithms can be categorized into two main types:

1. **Comparison-based Sorting Algorithms**: These algorithms compare data elements to determine their relative order.
   - Example: bubble sort, insertion sort, selection sort, merge sort, quicksort, and heapsort.
2. **Non-comparison-based Sorting Algorithms**: These algorithms do not rely on comparisons to determine the order of data but instead leverage specific properties of the data, such as the range of values or the distribution of data, to guide the sorting process.
   - Example: counting sort, radix sort, and bucket sort.

Sort Algorithm

Comparison-based Sorting Algorithms

$O(n^2)$

- **Bubble Sort:** This method compares adjacent data elements and swaps them if they are in the wrong order. This process is repeated until all the data is sorted.
- **Insertion Sort:** This method gradually builds a sorted list one element at a time by inserting each element into its correct position within the already sorted portion of the list.
- **Selection Sort:** This method repeatedly finds the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted portion. This process is repeated until all the data is sorted.

Sort Algorithm

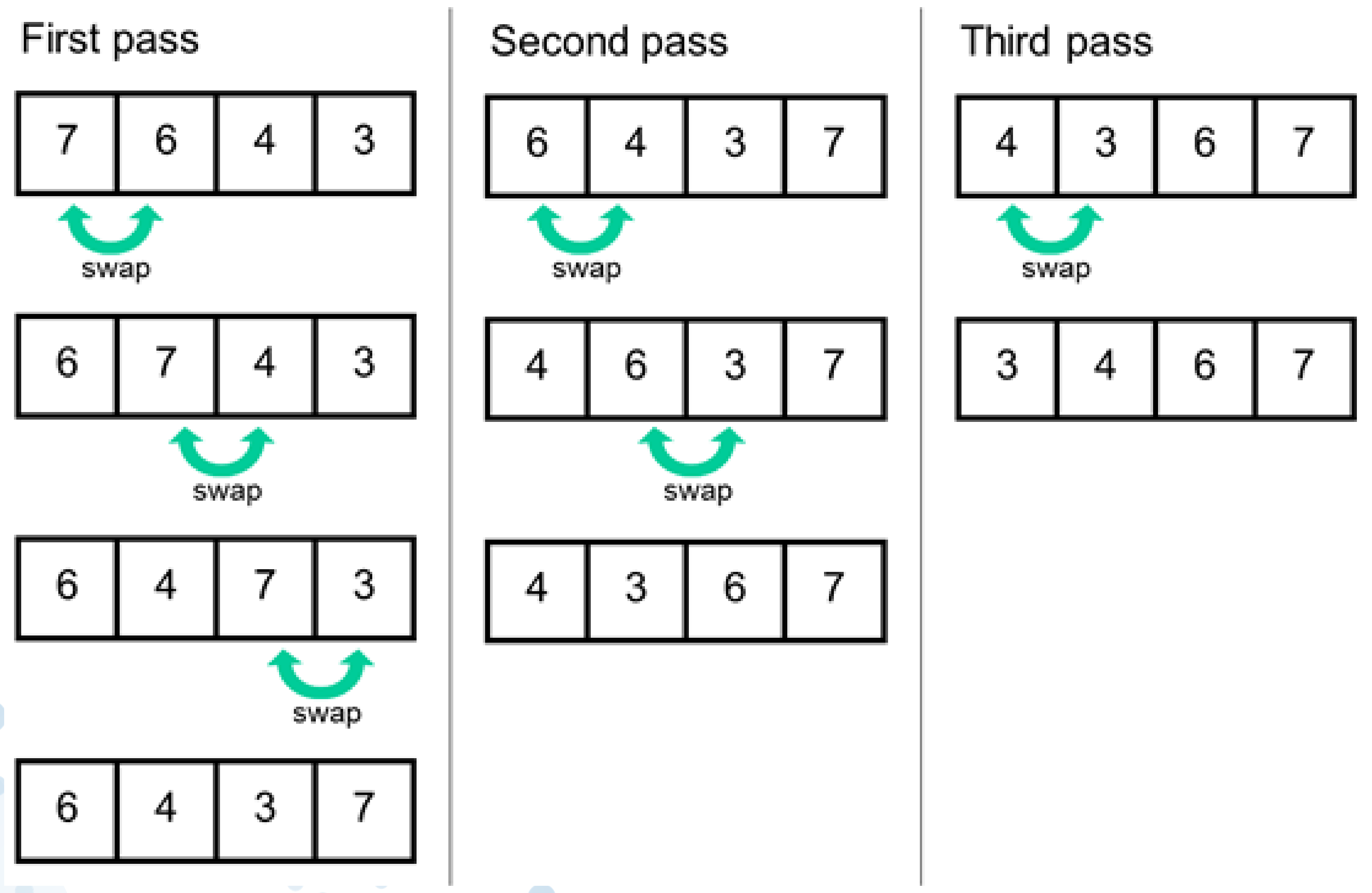
Comparison-based Sorting Algorithms

$\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$

- **Merge Sort:** This method uses a *divide-and-conquer* approach by dividing the data into two halves, recursively sorting both halves, and then merging the sorted halves to produce a fully sorted list.
- **Quicksort:** This method also uses a *divide-and-conquer* approach by selecting a pivot element, dividing the data into two parts—elements less than the pivot and elements greater than or equal to the pivot—and then recursively sorting both parts.

Bubble Sort

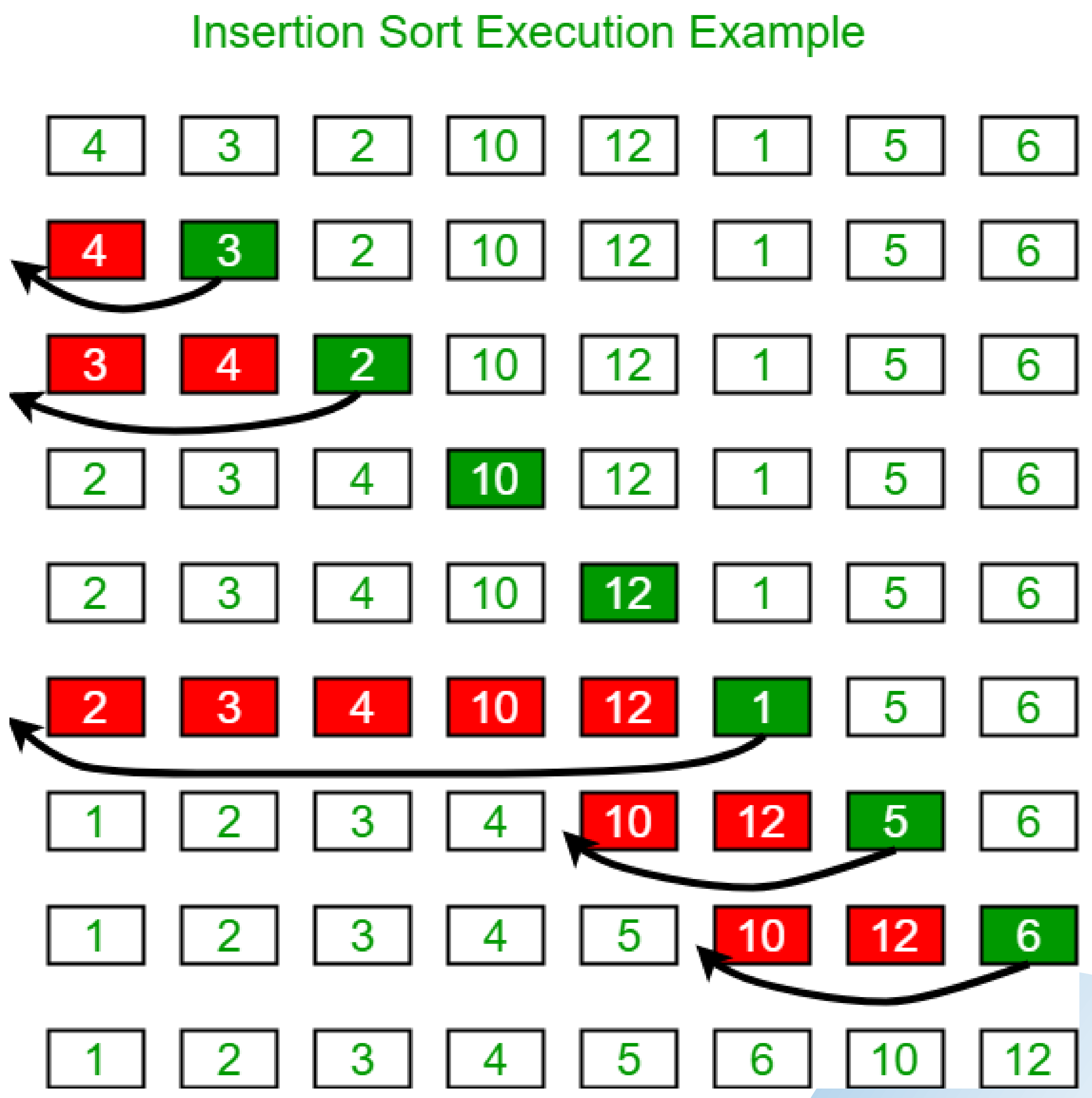
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.



https://www.computersciencebytes.com/sorting-algorithms/bubble-sort/

Insertion Sort

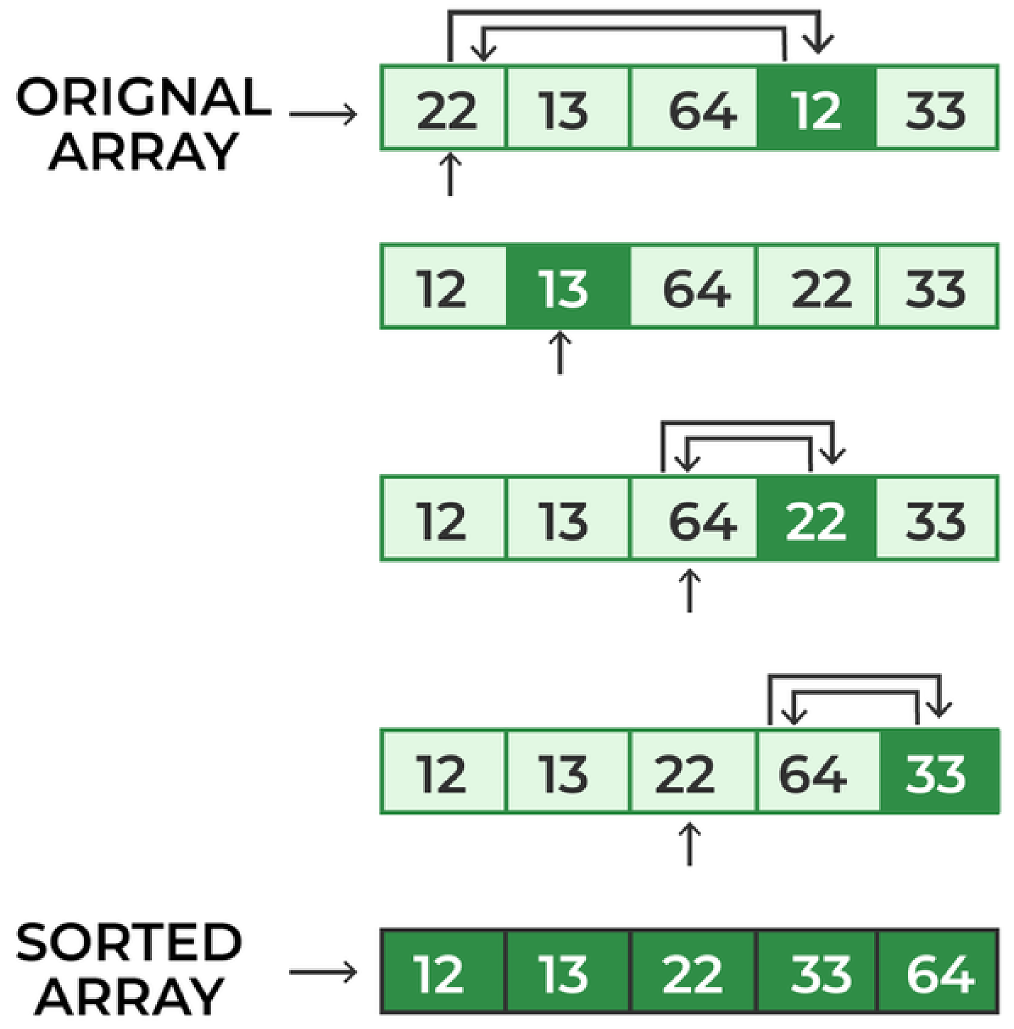
Insertion sort is a simple and efficient sorting algorithm that builds the final sorted array one element at a time. Insertion sort provides several advantages such as simple implementation, efficient for small data sets, and more efficient in practice than most other simple quadratic algorithms like selection sort or bubble sort.



https://www.geeksforgeeks.org/php-program-for-insertion-sort/

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

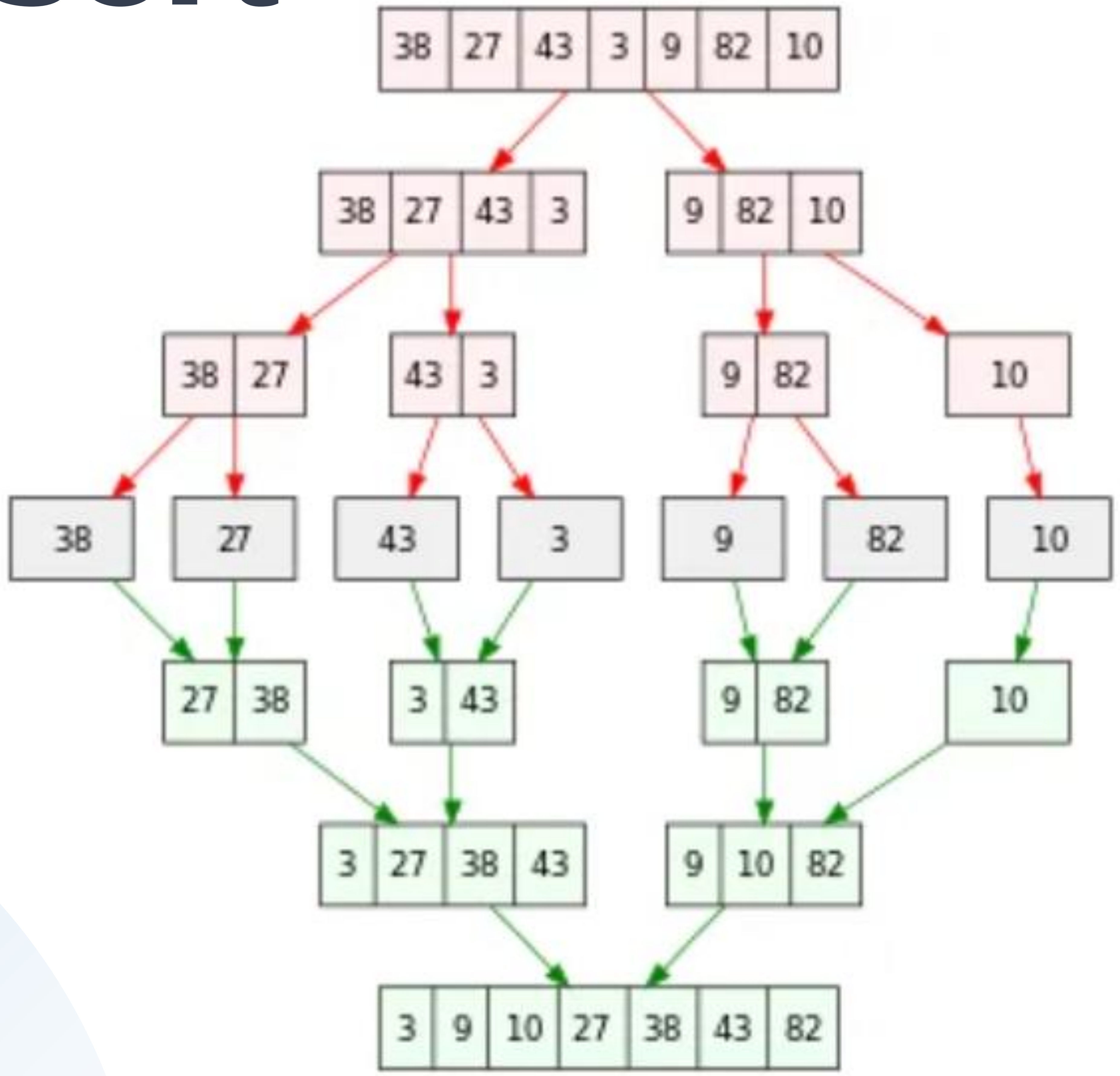


https://www.geeksforgeeks.org/java-program-for-selection-sort/

Merge Sort

- Merge Sort is a classic sorting algorithm that uses the Divide and Conquer strategy to sort an array or list of elements.
- Merge Sort is known for its efficiency and simplicity, especially when dealing with large datasets.
- Merge Sort has a time complexity of $O(n \log n)$, making it more efficient than some other sorting algorithms, such as Bubble Sort or Insertion Sort, for larger data sets.

Merge Sort

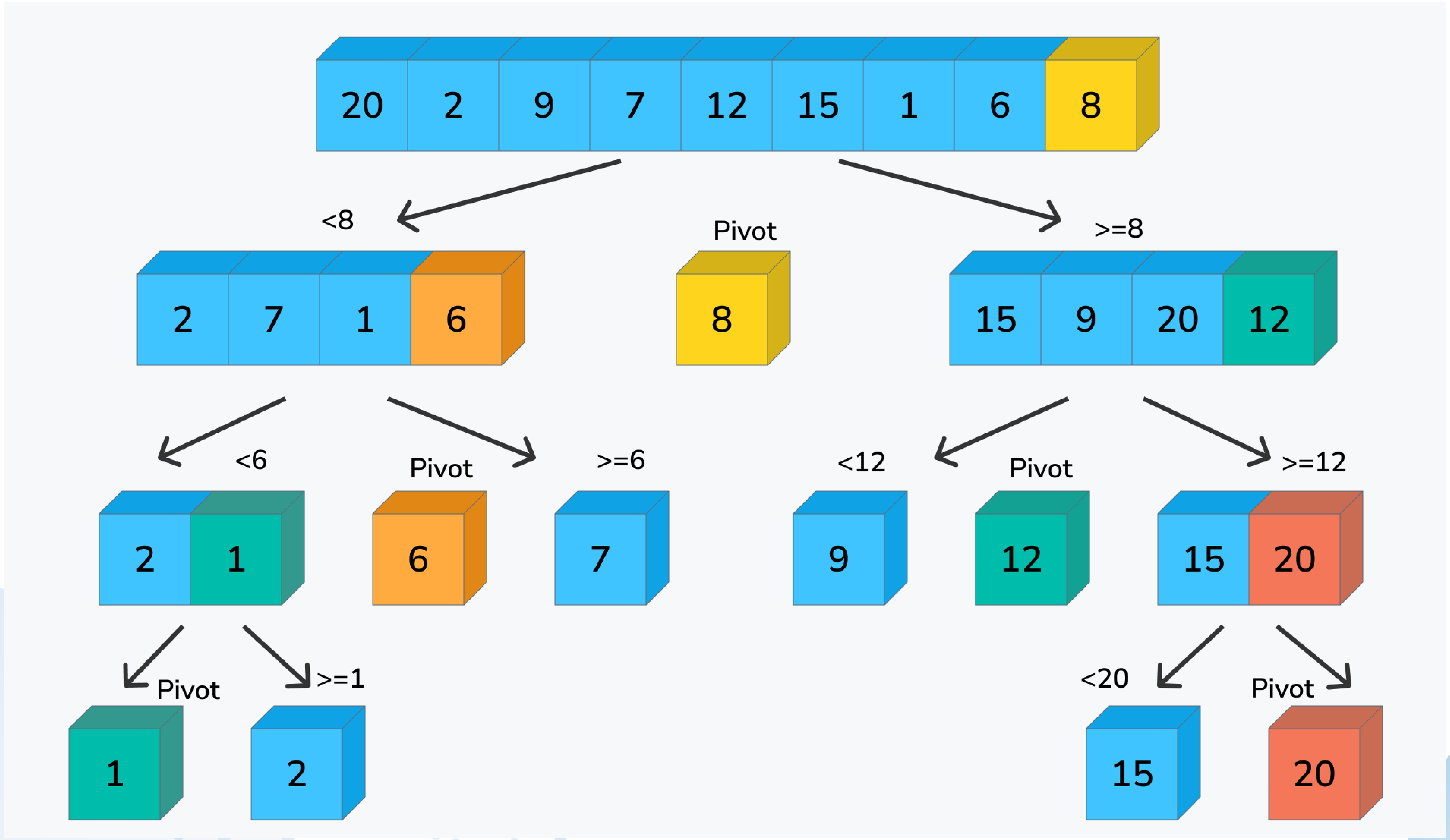


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Quick Sort

- Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
- The steps are as follows:
  1. Select one element as the pivot.
  2. Partition the data into two parts: elements less than or equal to the pivot are moved to the left of the pivot, and elements greater than the pivot are moved to the right.
  3. Repeat steps 1 and 2 for the data on the left and right of the pivot until each partition contains one or zero elements.
  4. Combine the sorted partitions together.

Quick Sort



https://workat.tech/problem-solving/tutorial/sorting-algorithms-quick-sort-merge-sort-dsa-tutorials-6j3h98lk6j2w

Quick Sort

The Pivot

- With the Quick Sort basics out of the way, technically, you can choose any value in the array as the pivot. However, some options can result in less optimal runtimes.

Naive Options:

- Always choose the first element of the current array subsection.
- Always choose the last element of the current array subsection.

- Both of these will work and can be a bit simpler to implement. However, they run the risk of worse time complexity. On a sorted or nearly sorted array, these will both have $O(n^2)$ runtime as each iteration of sorting will have to sort all of the remaining elements.

Quick Sort

Pivot Improvements:

- **Random value from the array as pivot**
- **Median Value of the array**

These options can improve the runtime and increase the likelihood of an $O(n \log n)$ runtime, including for sorted arrays. We'll discuss using a median value.

Quick Sort

Pivot Improvements:

Median Value

- Median refers to the middle value in a sorted set of data, with an equal number of items on either side of it.
- Ideally, every time we run Quick Sort, we want to pivot around the median value of our current array subsection, rather than skewing the implementation towards the highest or lowest values.
- Unfortunately, we don't actually know the median value because the array is not yet sorted. Instead we can use the "**Median of Three**" Strategy.

Quick Sort

Pivot Improvements:

Median of Three

- To implement the "Median of Three" strategy, select the first, last, and middle elements from your current array (or array portion as the recursive calls begin).
- Take the median (middle) value of those three elements to use for the current pivot.

The picture can't be displayed.

https://www.chegg.com/homework-help/questions-and-answers/modify-given-code-pivot-chosen-using-median-three-instead-def-inplacequicksort-s-b-sort-li-q54937142

Search Algorithm

A Search Algorithm is a method used to locate specific data or values within a dataset or data structure, such as an array or linked list, with the objective of finding the position or index of the desired data.

Common types of Search Algorithms include:

- **Linear Search**: This is the simplest search algorithm, where each element in the dataset is checked one by one from the beginning to the end until the desired data is found. It is suitable for unsorted data. $O(n)$

Search Algorithm

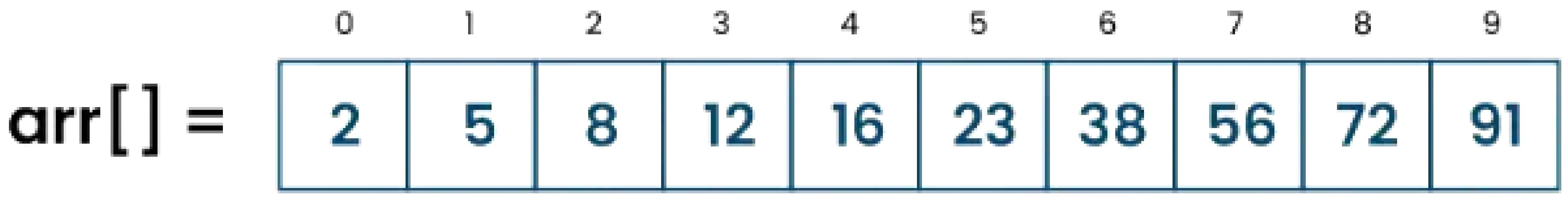
- **Binary Search:** This algorithm is used for sorted data. It works by dividing the dataset into two parts and comparing the middle element with the target value. If the value is not found, the search is repeated in the appropriate half of the dataset. This method is more efficient than Linear Search due to its divide-and-conquer approach. $O(\log n)$
- **Jump Search:** An improvement over Linear Search, this algorithm jumps over sections of data instead of checking each element one by one. It is useful for large, sorted datasets.
- **Depth-First Search (DFS)** and **Breadth-First Search (BFS):** These algorithms are used for searching in Trees or Graphs to determine the location of a specific value within the structure

Binary Search

- **Binary Search** is an efficient algorithm for finding an item from a sorted list of items.
- Binary Search works by repeatedly dividing the search interval in half and comparing the target value to the middle element of the list.
  - If the target value is equal to the middle element, you've found the target, and the search ends.
  - If the target value is less than the middle element, discard the right half of the list and continue the search with the left half.
  - If the target value is greater than the middle element, discard the left half of the list and continue the search with the right half.

Binary Search

Initially | Find Key = 23 using Binary Search



arr[] =

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

Complexity

What is Complexity?

- An algorithm is a step-by-step procedure or process for solving a problem in a structured manner, with a clear starting and ending point. When followed correctly, it produces the desired result.
- **Algorithm Complexity** refers to measuring the efficiency of an algorithm in terms of how complex it is. This is evaluated based on various factors, such as:
  - **Time Complexity:** The amount of time an algorithm takes to process different sizes of input data.
  - **Space Complexity:** The amount of memory an algorithm requires for execution.

Complexity

Definition:

- The feasibility of solving a problem within a reasonable amount of time using computation.
- Tractable problems can be solved using algorithms with polynomial-time complexity.

Characteristics:

- Tractable Problems: Problems solvable in polynomial time ($O(n^k)$ for some constant k).
- Intractable Problems: Problems that cannot be solved in polynomial time, often requiring exponential ($O(2^n)$) or factorial time ($O(n!)$).

Examples:

- Tractable: Sorting data using merge sort ($O(n \log n)$).
- Intractable: The Traveling Salesman Problem (TSP) in its general form ($O(n!)$).

Time Complexity

- Computational Tractability and Time Complexity are related concepts used to evaluate the efficiency and feasibility of solving problems with algorithms.
- Problems with polynomial time complexity ($O(n^k)$) are generally considered tractable.
- Computational Tractability: Focuses on whether a problem can be solved in a reasonable amount of time (polynomial time) and involves determining if an algorithm is practical for real-world use.
- Time Complexity: Focuses on measuring the number of computational steps needed to solve a problem as a function of input size, using Big O Notation to express the growth rate of the running time.

Asymptotic order of growth

- **Asymptotic order of growth** is a concept in computer science and mathematics that describes the behavior of functions as their input size grows towards infinity. It is used to classify algorithms according to **their running time or space requirements** in terms of their input size. This classification helps in understanding and comparing the efficiency of different algorithms.
- The primary purpose of asymptotic analysis is to **provide a high-level understanding of the algorithm's efficiency** without getting bogged down by machine-specific details. It abstracts away constants and lower-order terms, focusing only on the dominant term that dictates the growth rate as the input size increases.

Notations

The most commonly used asymptotic notations are:

1. **Big O Notation (O):** Represents the upper bound of the growth rate of a function. It provides an upper limit on the time or space complexity.
   - Example: $O(n^2)$ means the function's growth rate will not exceed n2n 2 for large n.
2. **Big Theta Notation (Θ):** Represents the tight bound of the growth rate. It provides both an upper and a lower limit on the time or space complexity.
   - Example: $\Theta(n \log n)$ means the function's growth rate is both upper and lower bounded by $n \log n$.
3. **Big Omega Notation (Ω):** Represents the lower bound of the growth rate. It provides a minimum limit on the time or space complexity.
   - Example: $\Omega(n)$ means the function's growth rate will not be less than n for large n.

Notations

4. **Little o Notation (o):** Indicates that a function grows slower than another when the input size grows. It is similar to Big O but without the equality.
   - Example: $o(n^2)$ means the function grows slower than n^2 and not equal to n^2.
5. **Little Omega Notation (ω):** Indicates that a function grows faster than another when the input size grows. It is similar to Big Omega but without the equality.
   - Example: $\omega(n)$ means the function grows faster than n and not equal to n.

Big O Notation

- A mathematical notation to describe the upper bound of an algorithm's running time.
- Helps in analyzing the performance and scalability of algorithms.
- Focuses on the worst-case scenario.
- Predicts the performance for large inputs.
- Helps in comparing different algorithms.
- Essential for optimizing code and improving efficiency.

Basic concepts

- Time Complexity: Measures the time an algorithm takes to complete as a function of the input size.
- Space Complexity: Measures the memory an algorithm uses as a function of the input size.

Big O Notation

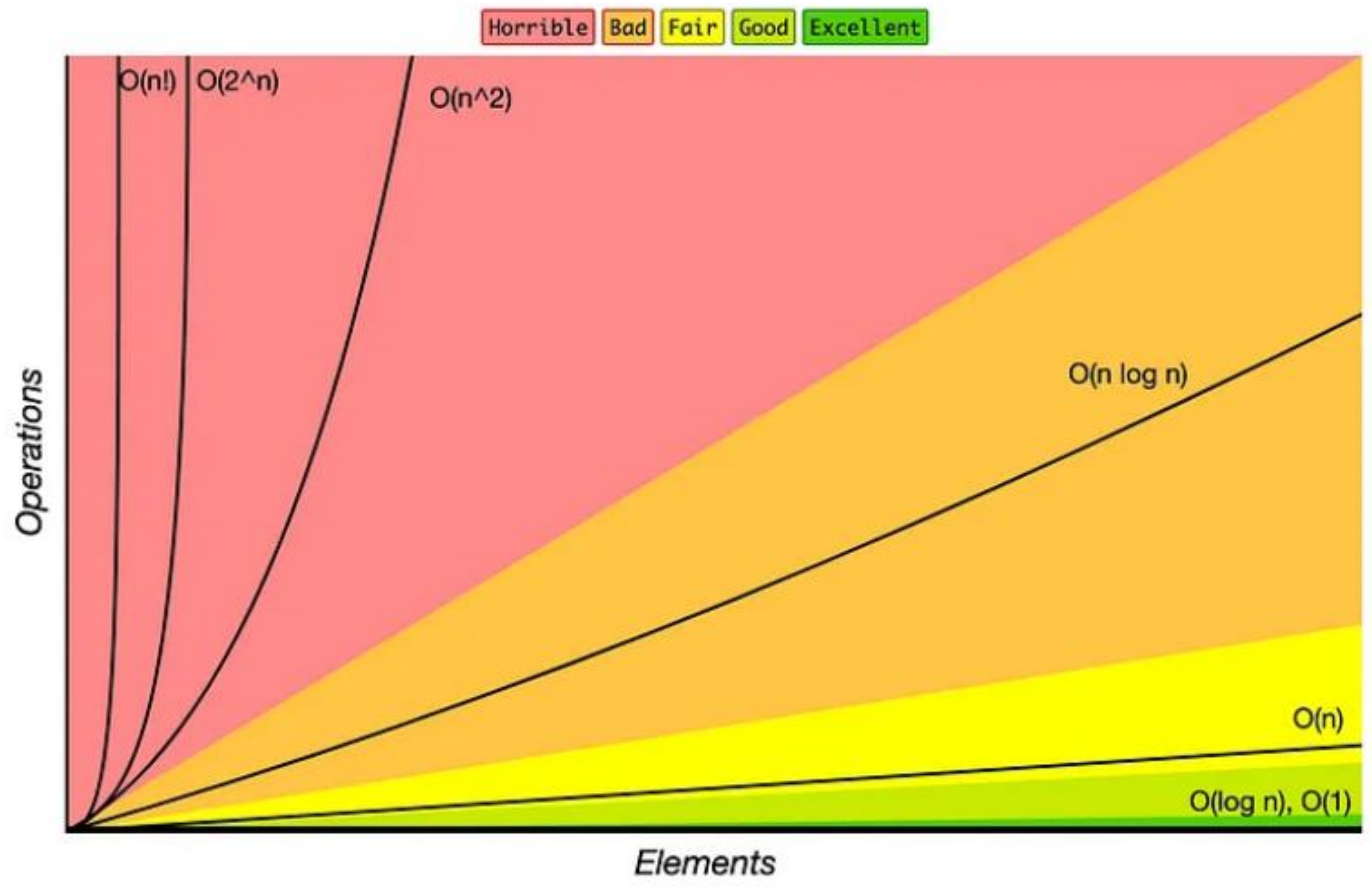
- The complexity of an algorithm is often measured using "Big-O Notation", which provides an approximation of complexity when the input size becomes very large.
- For example:
  - O(1): Constant Complexity – The execution time remains the same regardless of input size.
  - O(n): Linear Complexity – The execution time increases proportionally to the input size.
  - O(n^2): Quadratic Complexity – The execution time grows quadratically as the input size increases.
  - O(log n): Logarithmic Complexity – More efficient than linear complexity, commonly seen in algorithms like binary search.

Big O Notation

Name	**Notation**	n **= 1**	n **= 2**	n **= 4**	n **= 8**	n **= 16**	n **= 1024**
constant	O(1)	1	1	1	1	1	1
logarithmic	O($\log n$)	1	1	2	3	4	10
linear	O(n)	1	2	4	8	16	1024
linearithmic	O($n \log n$)	1	2	8	24	64	10,240
quadratic	O(n^2)	1	4	16	64	256	1,048,576
cubic	O(n^3)	1	8	64	512	4,096	1,073,741,824
exponential	O(2^n)	2	4	16	256	65,536	2^1024
factorial	O($n!$)	1	2	24	40,320	20,922,789,888,000	1024 x 1023 x 1022 ... 3 x 2 x 1

Big O (Big O Notation) is a method for measuring the efficiency of an algorithm by evaluating the maximum number of operations it performs as the input size increases.

Big O Notation



https://www.bigocheatsheet.com/

Big O Notation

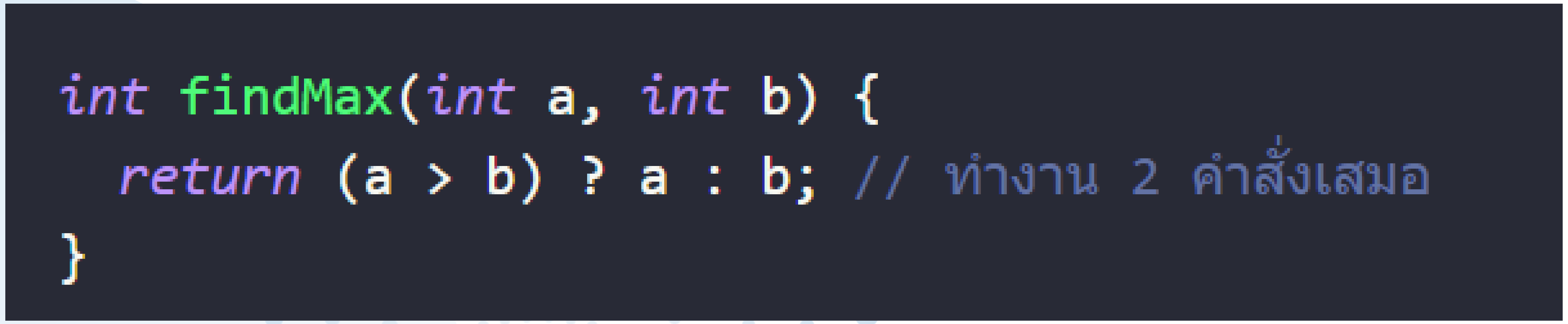
ความสำคัญของ Big O

- ช่วยให้เราสามารถเปรียบเทียบประสิทธิภาพของอัลกอริทึมต่างๆ ได้ โดยไม่ต้องพึ่งพาปัจจัยอื่นๆ เช่น hardware ระบบปฏิบัติการ ภาษาที่ใช้เขียน เป็นต้น
- ช่วยให้เราสามารถเลือกใช้ Algorithm ที่มีประสิทธิภาพสูงสุดสำหรับงานนั้นๆ ได้
- ในการพัฒนาซอฟต์แวร์ที่มีผู้ใช้งานจำนวนมาก การเลือกใช้อัลกอริทึมที่มีประสิทธิภาพสูงจะช่วยประหยัดต้นทุนในการดำเนินงาน (Operating Cost) ได้มาก
- แนวคิดของ Big O Notation มาจากทฤษฎี Asymptotic Analysis ในคณิตศาสตร์ ซึ่งใช้ในการวิเคราะห์พฤติกรรมขีดจำกัดของ function เมื่อตัวแปรมีค่ามากๆ หรือน้อยๆ มากๆ โดย Big O notation แสดงถึง "กรณีที่เลวร้ายที่สุด" (Worst case Scenario) บนขอบเขตของเวลาการทำงานของ Algorithm มันอธิบายถึงเวลาสูงสุดที่ Algorithm อาจใช้สำหรับข้อมูลขนาด n ใดๆออกมา

Big O Notation

Constant (O(1))

- O(1) is something that developers generally prefer the most. Regardless of the amount of data, the processing time will always take just one cycle.
- เป็น Big O ที่ดีที่สุด ไม่ว่า input จะมีขนาดเท่าไร ระยะเวลาในการประมวลผลจะคงที่เสมอ
- ในกรณีนี้ จำนวนคำสั่งที่ทำงานไม่ขึ้นกับขนาดของ input ดังนั้นเวลาในการประมวลผลจึงคงที่ไม่ว่า input จะมีขนาดเท่าใด
- ตัวอย่างเช่น การเข้าถึงข้อมูลใน array ด้วย index, arithmetic operations (+, -, *, /)



```
int findMax(int a, int b) {
  return (a > b) ? a : b; // ทำงาน 2 คำสั่งเสมอ
}
```

Big O Notation

Constant (O(1))

```
def find_character_h():
    characters = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", 
"Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
    print(characters[7])
find_character_h()
```

In this example, we already know where the character H is, so we can immediately retrieve that data.

Big O Notation

Logarithmic (O(log n))

- A typical example of an O(log n) algorithm is Binary Search, which works by repeatedly dividing the search interval in half. The advantage is that it is much faster than O(n), but the data must be sorted first.
- เป็น Big O ที่มีประสิทธิภาพดีมาก โดยในแต่ละรอบของ loop จะลดขนาดของปัญหาลงครึ่งหนึ่ง
- ตัวอย่างเช่น binary search, การหารากที่สอง (square root)

Big O Notation

Logarithmic (O(log n))

In this example, the Binary Search finds the number 24 in just a few steps.

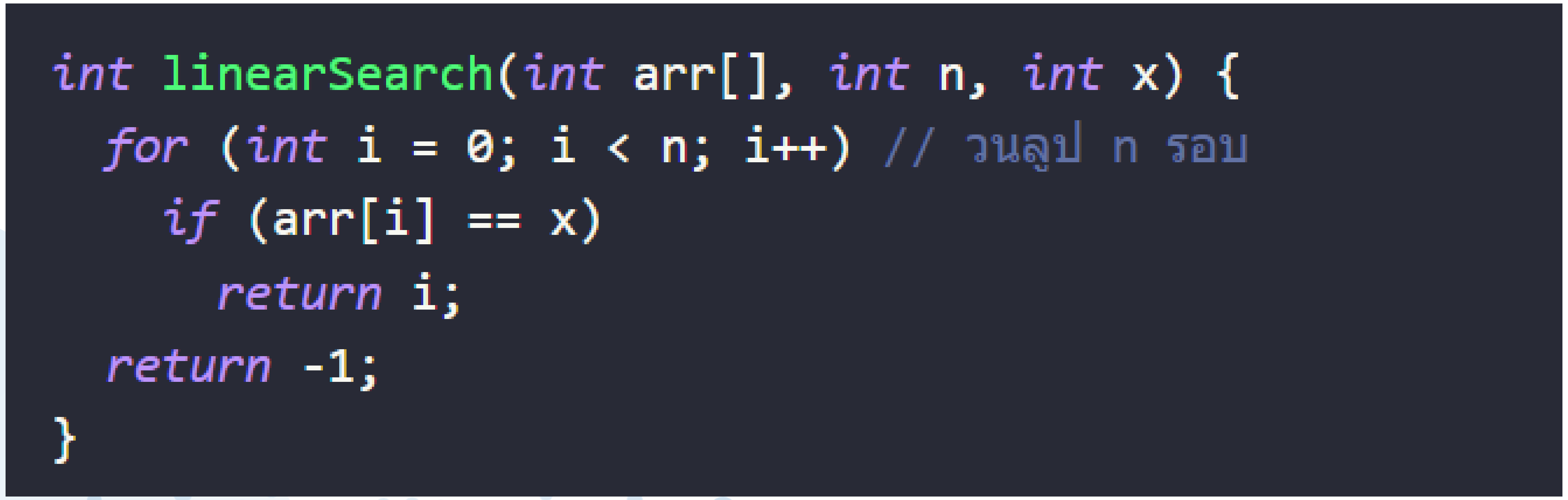
```
def binary_search(arr, l, r, x):
    if r >= l:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, l, mid - 1, x)
        else:
            return binary_search(arr, mid + 1, r, x)
    else:
        return -1

def main():
    arr = [2, 4, 6, 8, 11, 24, 36]
    x = 24
    result = binary_search(arr, 0, len(arr) - 1, x)
    print("Result at index:", result)
main()
```

Big O Notation

Linear (O(n))

- An O(n) algorithm takes time proportional to the input size. In the worst case, it will take as many steps as there are elements.
- ในกรณีนี้ จำนวนคำสั่งที่ทำงานจะเป็นสัดส่วนโดยตรงกับขนาดของ input ตัวอย่างเช่น Linear Search



```
int linearSearch(int arr[], int n, int x) {
  for (int i = 0; i < n; i++) // วนลูป n รอบ
    if (arr[i] == x)
      return i;
  return -1;
}
```

Big O Notation

Linear (O(n))

```
def find_character_h():
    characters = ["C", "B", "D", "E", "F", "I", "J", "L", "M", "A", "N", "O", "P", "K", "Q", "R",
"S", "T", "U", "G", "V", "W", "X", "Y", "Z", "H"]
    for i in range(len(characters)):
        if characters[i] == "H":
            print(characters[i])
            print(i)

find_character_h()
```

In this example, the characters are shuffled, and finding "H" takes 26 steps in the worst case.

Big O Notation

Linearithmic (O(n log n))

- O(n log n) algorithms usually involve nested loops where one of the loops performs a logarithmic operation. Common examples include Merge Sort, Heap Sort, and Quick Sort. => Divide and conquer
- Algorithm ที่มีความซับซ้อนในระดับนี้ จะมี loop แบบ n รอบ ที่มี loop แบบ log n รอบอยู่ข้างใน ตัวอย่างเช่น Merge Sort

Big O Notation

Quadratic ($O(n^2)$)

- $O(n^2)$ algorithms involve a double nested loop. Examples include Bubble Sort, Insertion Sort, and Selection Sort.
- ในกรณีนี้ จำนวนคำสั่งที่ทำงานจะเป็นกำลังสองของขนาด input ตัวอย่างเช่น Bubble Sort

```
def quadratic_example(n):
    for i in range(1, n+1):
        for j in range(1, n+1):
            print("I:", i, "J:", j)
```

In this example, if n is 10, it loops 100 times. If n is 100, it loops 10,000 times.

Big O Notation

Cubic (O(n³))

- O(n³) algorithms involve a triple nested loop, making them significantly slower.

```
def cubic_example(n):
    for i in range(1, n+1):
        for j in range(1, n+1):
            for k in range(1, n+1):
                print("I:", i, "J:", j, "K:", k)

cubic_example(10)
```

For n = 10, this example will loop 1,000 times. For n = 100, it will loop 1,000,000 times.

Big O Notation

Exponential (O(2^n))

- O(2^n) algorithms grow exponentially with the input size. These should be avoided whenever possible.
- Algorithm ที่มีความซับซ้อนในระดับนี้จะมีเวลาทำงานเพิ่มขึ้นเป็นเลขชี้กำลัง ตัวอย่างเช่น การหาค่า Fibonacci แบบ recursive

```
def exponential_example(n):
    for i in range(1, 2**n + 1):
        print("round:", i)

exponential_example(4)
```

For n = 4, it will loop 16 times. For n = 16, it will loop 65,536 times.

Big O Notation

Factorial (O(n!))

O(n!) algorithms grow factorially with the input size. A classic example is the Traveling Salesman Problem (TSP).

```
def factorial(n):
    f = 1
    for k in range(2, n+1):
        f *= k
    return f

print(factorial(8))
```

For n = 8, this example computes 40,320 iterations.

Big O Notation: abuses

One-way "equality." $O(g(n))$ is a set of functions, but computer scientists often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.

Ex. Consider $g_1(n) = 5n^3$ and $g_2(n) = 3n^2$

- We have $g_1(n) = O(n^3)$ and $g_2(n) = O(n^3)$
- But, do not conclude $g_1(n) = g_2(n)$

Domain and codomain. f and g are real-valued functions.

- The domain is typically the natural numbers: $\mathbb{N} \rightarrow \mathbb{R}$.
- Sometimes we extend to the reals: $\mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$.
- Or restrict to a subset.

Bottom line. OK to abuse notation in this way; not OK to misuse it.

Big O Notation

- Big-O notation is used to describe the **upper bound** of the running time of an algorithm when the size of the input changes.
- Generally, this notation indicates the relationship between time and the size of the input or the input size of one unit.
- The time used to run the algorithm at most (Upper bound) is how much. This is why Big-O notation is commonly used in analyzing the efficiency of algorithms.
- $O(n)$ is the notation indicating the maximum running time $(<= \ n)$.

Big O Notation

คำถามสำคัญเลยของ Big O คือ “ทำไมเราจึงสามารถใช้ Worst Case อย่าง Big O (จำนวนการทำงานสูงสุดของ Algorithm) มาวิเคราะห์แทนที่จะใช้การคำนวนทั้งหมดออกมาได้” เราจึงจำเป็นต้องพิสูจน์ว่า Big O นั้นได้ cover “จำนวน operation สูงสุดของ algorithm นั้น” ไว้เป็นที่เรียบร้อย

เพื่อให้เกิดความเข้าใจเพิ่มขึ้น เราจะนิยาม algorithm แบบเดียวกันกับ function โดยให้

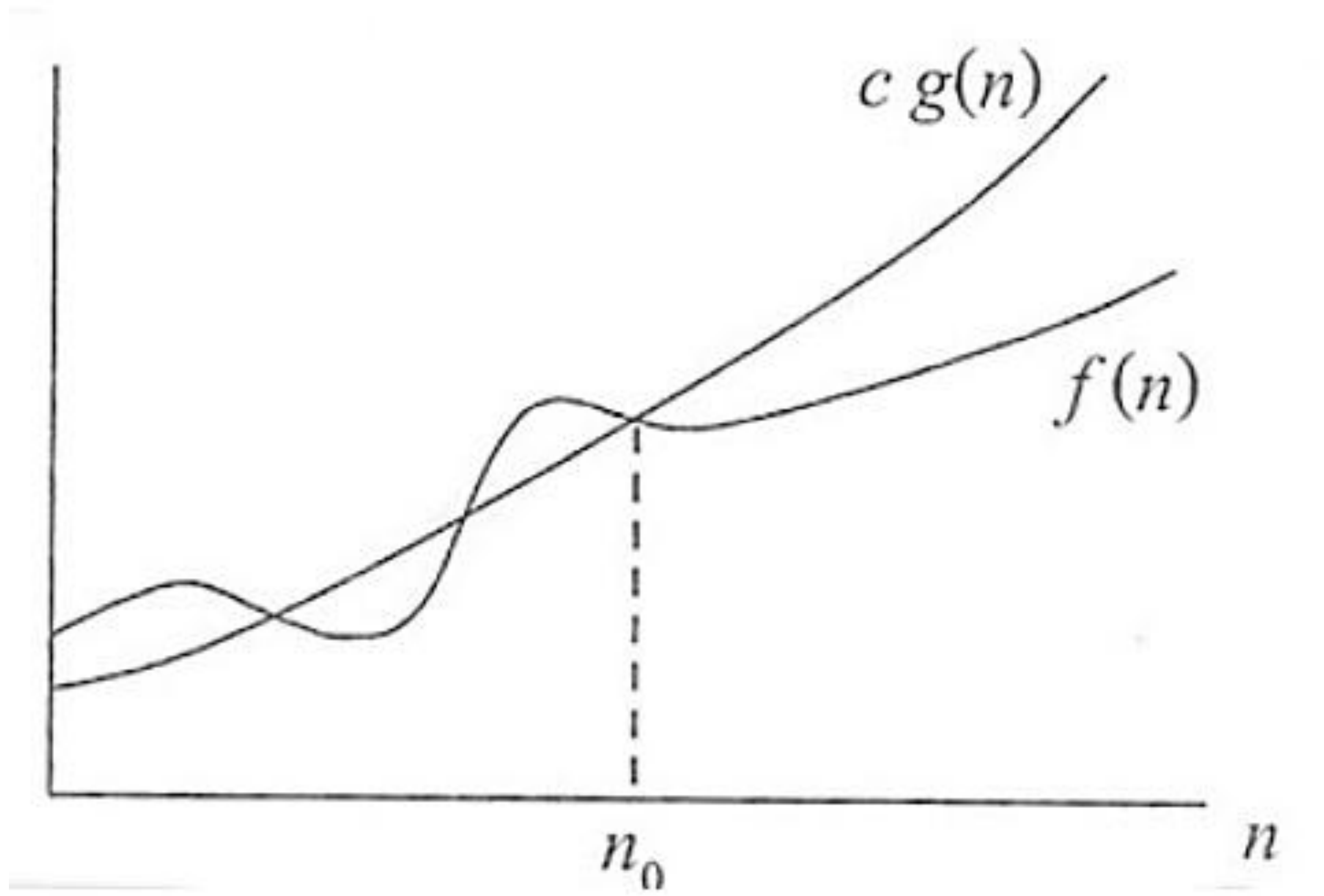
- f(n) คือจำนวนขั้นตอนการทำงานสูงสุดของอัลกอริทึม
- g(n) คือ function ที่ใช้ประมาณค่าของ f(n) ด้วย Big O Notation [นี่คือตัวที่เราจะใช้] ซึ่งก็คือ Worst case
- n คือขนาดของข้อมูลนำเข้า

ค่าคงที่ c ที่ทำให้จำนวนขั้นตอนการทำงานสูงสุดของ algorithm f(n) มีค่าไม่เกิน c คูณกับ g(n) เมื่อขนาดของข้อมูลนำเข้า n มีค่ามากพอ

Big O notation

Example

If *algorithm A* has a performance of $O(n^2)$ when n = 10, then *algorithm A* will take at most 100 units of time (it might not take exactly 100, but definitely no more than 100).



We write $f(n) \in O(g(n))$ to state that $f(n)$ is a function that grows no faster than $g(n)$.

Big O notation with multiple variables

Upper bounds. $f(m,n)$ is $O(g(m,n))$ if there exist constants $c > 0, m_0 \geq 0$, and $n_0 \geq 0$ such that $0 \leq f(m,n) \leq c \cdot g\ (m,n)$ for all $n \geq n_0$ or $m \geq m_0$.

Ex. $f(m,n) = 32mn^2 + 17mn + 32n^3$.

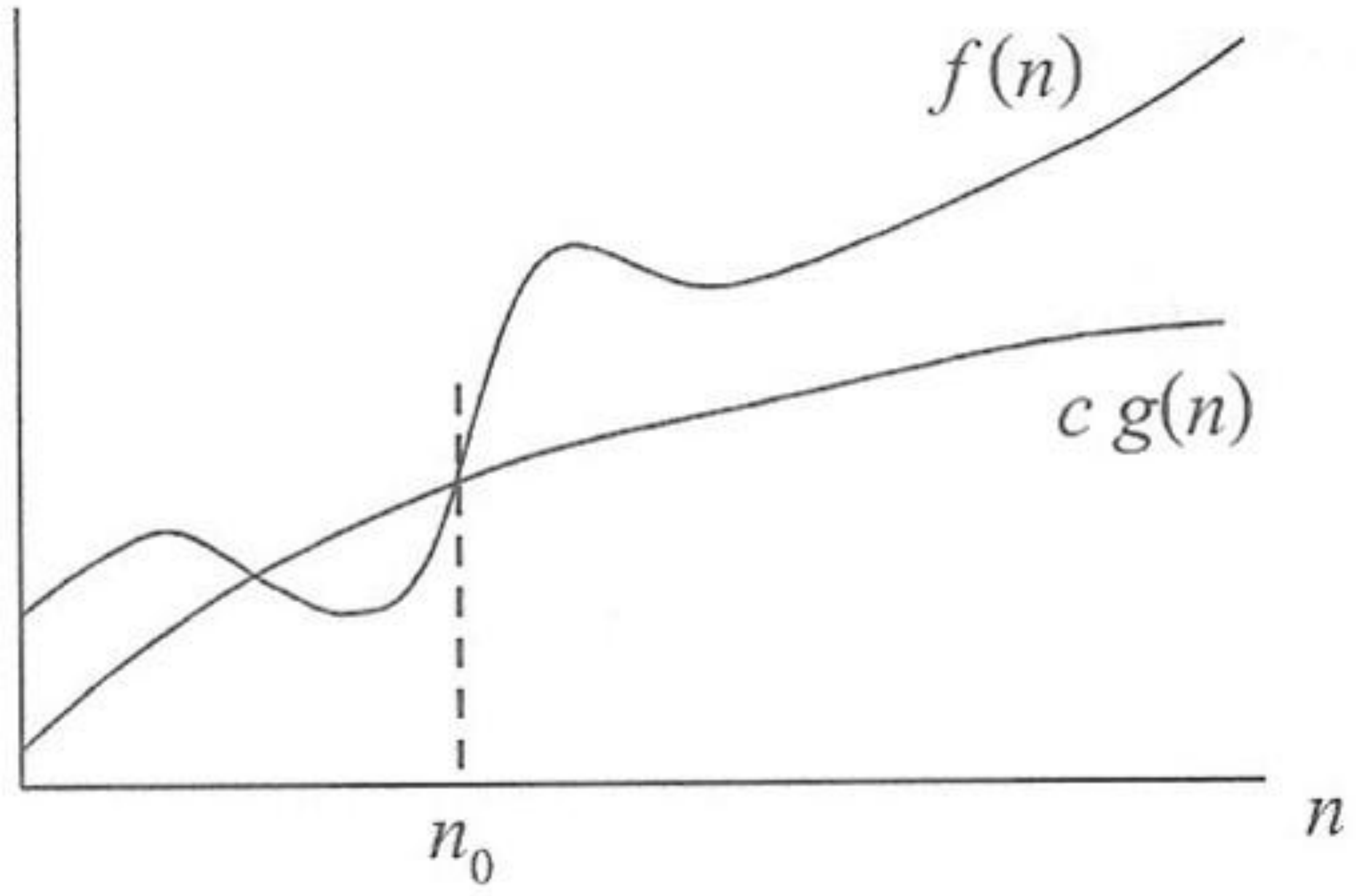
- $f(m,n)$ is in both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $f(m,n)$ is in $O(n^3)$ if a precondition to the problem implies $m \ \leq n$.
- $f(m,n)$ is neither in $O(n^3)$ nor in $O(mn^2)$.

Big Omega notation: Ω

- Big-Omega notation is used to describe the **lower bound** of the running time of an algorithm when given an input of size n.
- $\Omega(n)$ is the notation used to indicate the **best-case running** time of an algorithm.

Big Omega notation: Ω

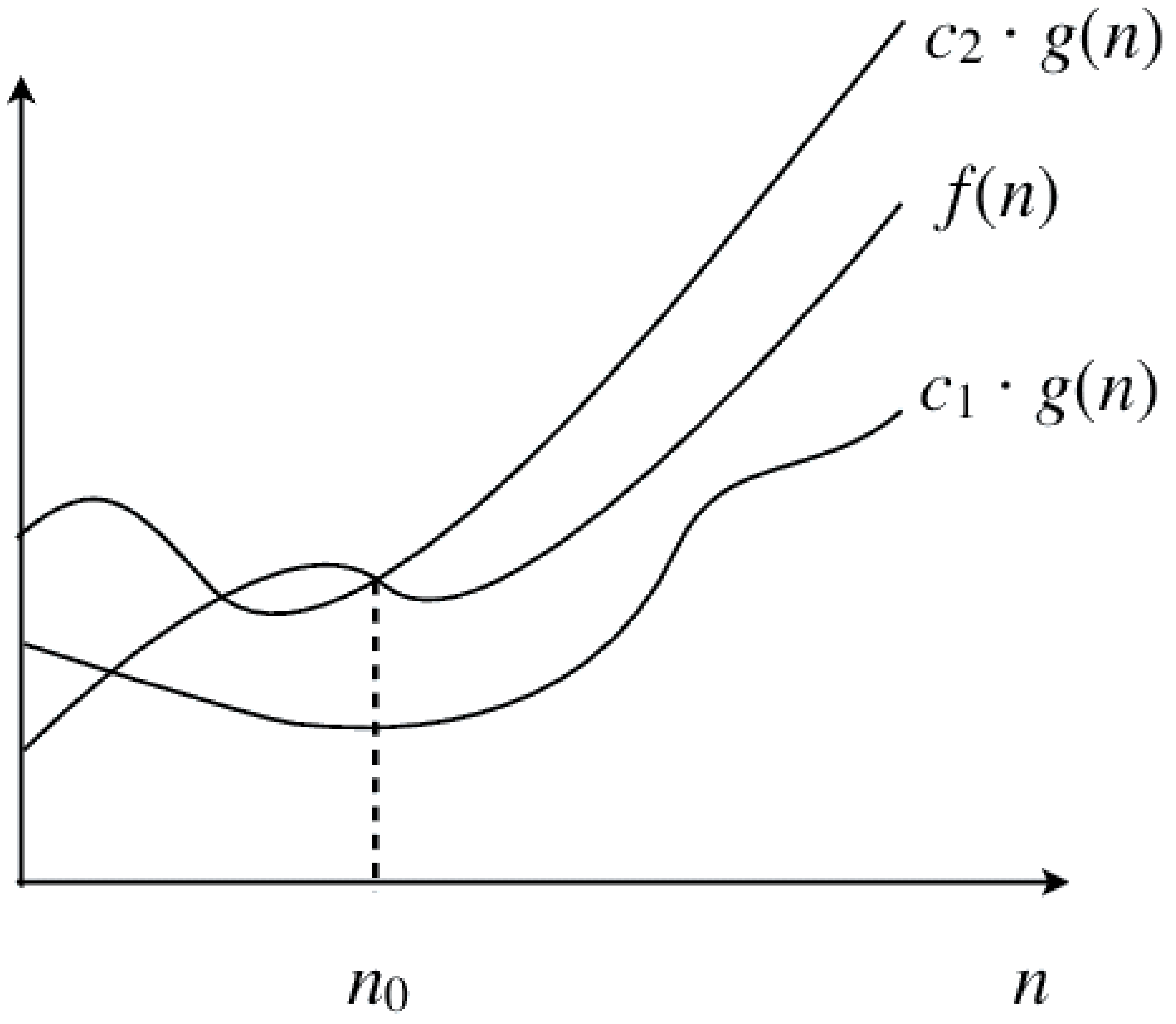
Example
If an *algorithm A* has a performance of $\Omega(n)$ when n = 10, then *algorithm A* will take at least 10 units of time (essentially, it will not take less time than this) to process an input of size n.



We write $f(n) \in \Omega(g(n))$ to state that $f(n)$ is a function that grows at least as fast as $g(n)$.

Big Theta notation: Θ

Big-Theta notation is used to describe the relationship between the running time of a function $f(n)$ and another function $g(n)$ such that $f(n) = \Theta(g(n))$, which means ***f*(*n*)=*O*(*g*(*n*)) and *f*(*n*)=Ω(*g*(*n*))**. This indicates both upper and lower bounds.



Big Theta notation: Θ

Tight bounds $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0, c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is in $\Theta(n^2)$
- $f(n)$ is neither in $\Theta(n)$ nor in $\Theta(n^3)$

Typical usage. Mergesort makes $\Theta(n \log n)$ compares to sort n elements.

Summary of definitions

Def 1. Upper bound: A function $f(n)$ is in the set $O(g(n))$ if there are constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n > n_0, f(n) \le c \cdot g(n)$

Def 2. Lower bounds: A function $f(n)$ is in the set $\Omega(g(n))$ if there are constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n_0 > n_0, f(n) \ge c \cdot g(n)$

Def 3. Tight bounds: A function $f(n)$ is in the set $\Theta(g(n))$ if there are constants $c_1 > 0, c_2 > 0$, and $n_0 \in \mathbb{N}$ such that for all $n > n_0, \ c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$

Def 4. A function $f(n)$ is in $o(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \notin \Omega(g(n))$

Def 5. A function $f(n)$ is in $\omega(g(n))$ if $f(n) \in \Omega(g(n))$ and $f(n) \notin O(g(n))$

Limit Theorem

In the context of asymptotic notation, the Limit Theorem helps **determine the relationship between two functions $f(n)$ and $g(n)$** as n approaches infinity. The theorem uses limits to classify the growth rates of these functions, which is crucial for analyzing the efficiency of algorithms.

- The Limit Theorem states that given two functions $f(n)$ and $g(n)$ that are positive for all sufficiently large n, the limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

helps determine the asymptotic relationship between $f(n)$ and $g(n)$.

Limit Theorem

Cases of the Limit Theorem:

1. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
   - This implies $f(n)$ grows **slower** than $g(n)$, then $f(n) \in O(g(n))$ and $f(n) \notin \Omega(g(n))$
   - This means $f(n)$ is asymptotically smaller than $g(n)$.
2. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
   - This implies $f(n)$ grows **faster** than $g(n)$, then $f(n) \in \Omega(g(n))$ and $f(n) \notin O\big(g(n)\big)$
   - This means $f(n)$ is asymptotically larger than $g(n)$
3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $c > 0$
   - This implies $f(n)$ is **equal** to $c \cdot g(n)$, and we write $f(n) \in \Theta(g(n))$

Limit Theorem

Example: Use the limit theorem to determine how the following pairs of functions compare asymptotically:

1. $2n^4 + 4n^3 + n^2$ vs $9n^3 + 7n^2 + 6n$
2. $n^{1/2}$ vs $n^{1/4}$
3. $\log_2(n)$ vs $\log_3(n)$
4. $\log_2(n)$ vs $\log_2(n^2)$
5. $\log_2(n)$ vs $(\log_2(n))^2$

Master Method

The Master Theorem (or Master Method) is a widely used tool in the analysis of algorithms, particularly for solving recurrence relations that commonly arise in the analysis of divide-and-conquer algorithms. The theorem provides a way to determine the time complexity of recurrences of the form:

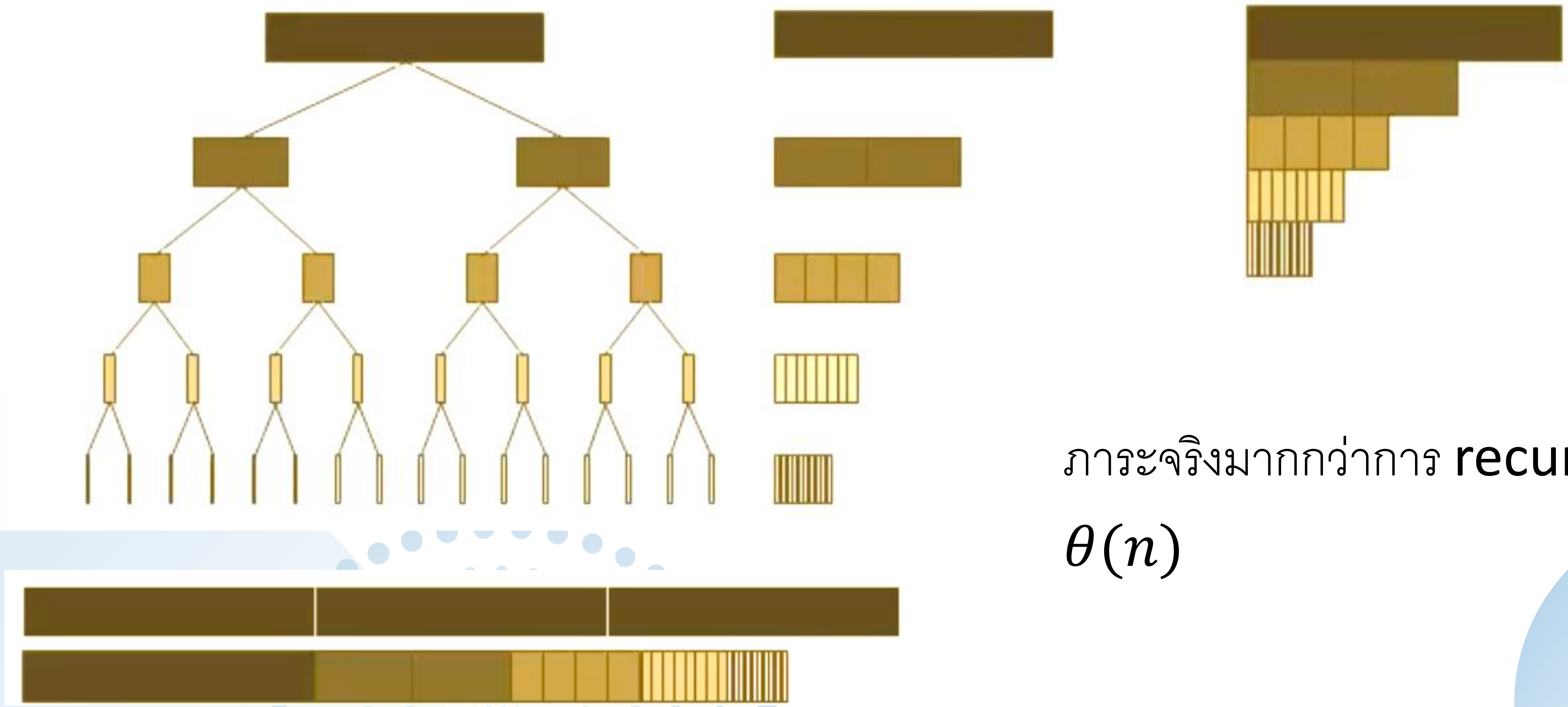
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $a \geq 1$ is the number of subproblems,
- n/b is the size of each subproblem (assuming that the input size is divided evenly),
- $f(n)$ is the cost of the work done outside the recursive calls (e.g., partitioning, merging).

Master Method

$$T(n) = 2T\left(\frac{n}{3}\right) + n$$

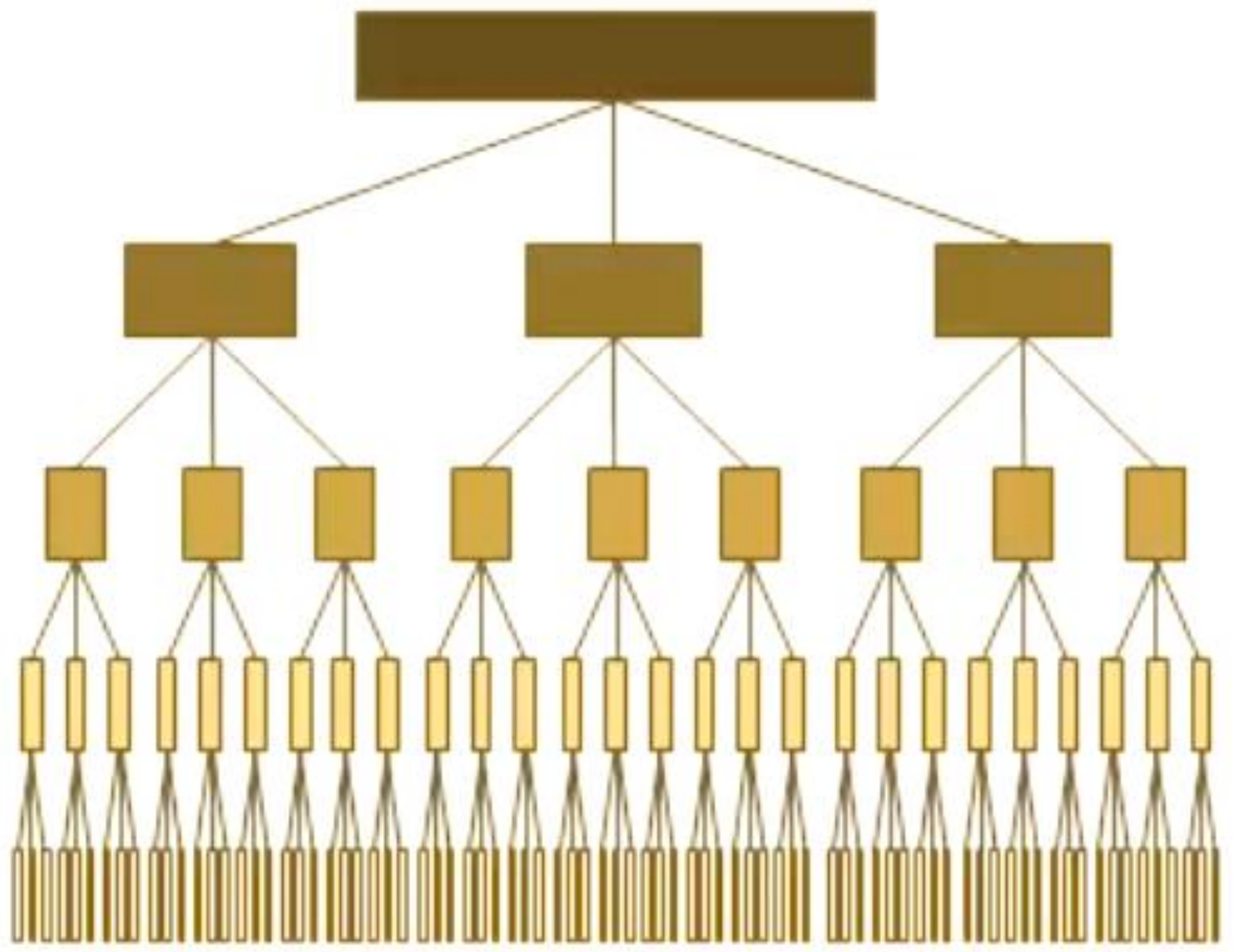


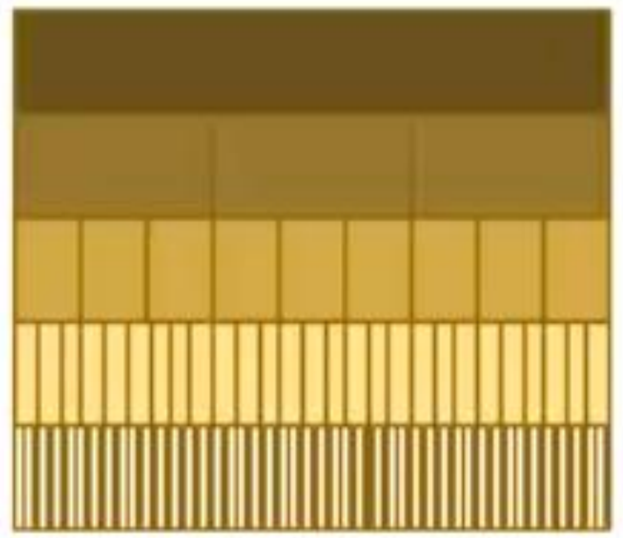
ภาระจริงมากกว่าการ recurrence

$\theta(n)$

Master Method

$$T(n) = 3T\left(\frac{n}{3}\right) + n$$



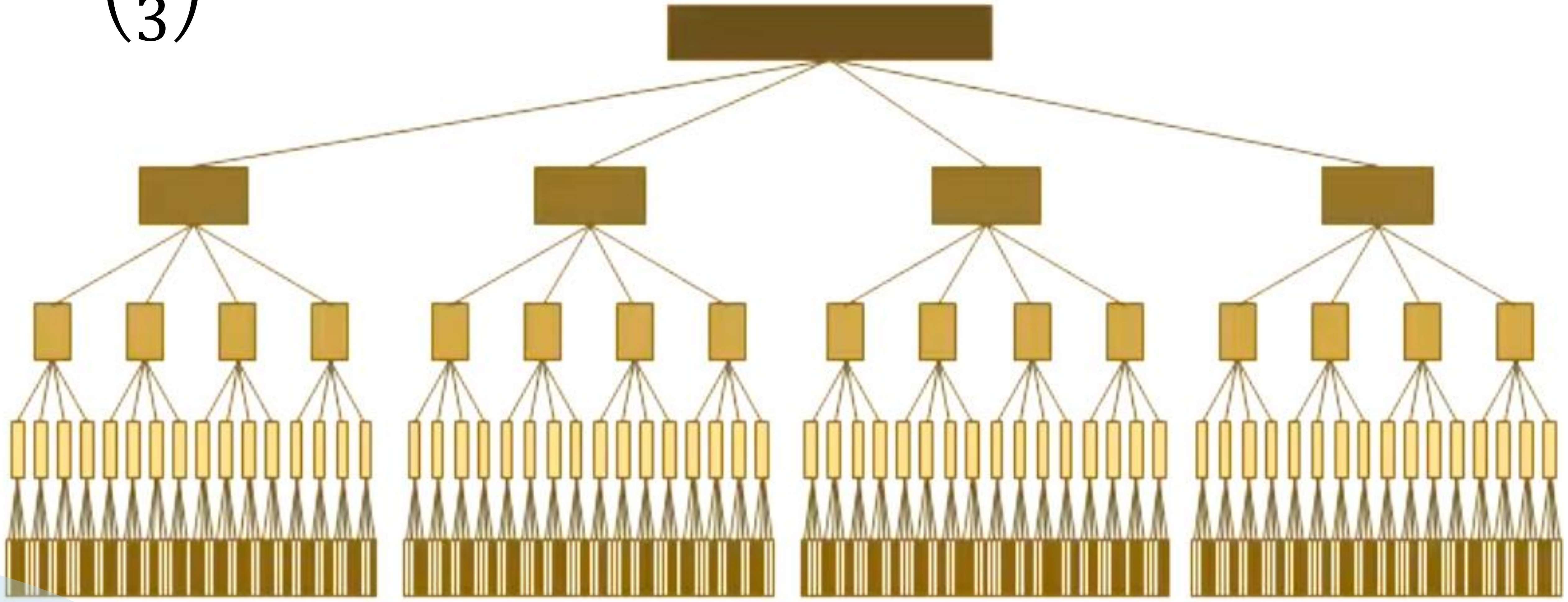


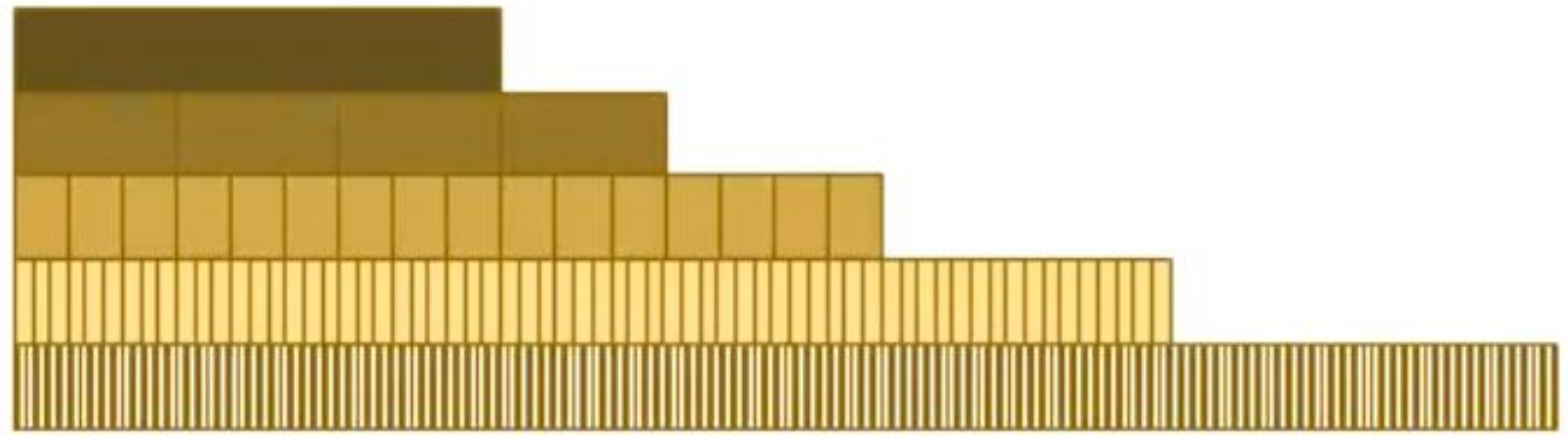
ภาระจริงเท่ากับการ recurrence

$\theta(n \log n)$

Master Method

$$T(n) = 4T\left(\frac{n}{3}\right) + n$$





ภาระจริงน้อยกว่าการ recurrence

$\theta(n^{\log_3 4})$

Master Method

$$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right) \qquad a \geq 1, b > 1, d \geq 0$$

$$c = \log_b a$$

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

Master Method

Ex.

1. $T(n) = T\left(\frac{n}{2}\right) + O(1)$

2. $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$

3. $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

4. $T(n) = 4T\left(\frac{n}{2}\right) + O(1)$

5. $T(n) = T\left(\frac{n}{3}\right) + O(1)$

6. $T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$

7. $T(n) = 9T\left(\frac{n}{3}\right) + O(n)$

8. $T(n) = 9T\left(\frac{n}{3}\right) + O(1)$

OUTLINE

- Introduction
- Algorithmic Complexity / Asymptotic Notation
- Algorithm
  - Brute Force
  - Divide-and-Conquer
  - Decrease-and-Conquer
  - Transform-and-Conquer
  - String Matching Algorithms
  - Backtracking & Branch-and-Bound
  - Greedy Algorithms
  - Graph Algorithms
  - Dynamic Programming

Brute Force

Brute force is a straightforward approach to solving a problem that involves systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. It is often considered a "trial and error" method.

Characteristics:

- Exhaustive search
- Simple to implement
- Guaranteed solution if enough resources are available
- Often inefficient for large problems

Brute Force

Key Characteristics of Brute Force

1. **Exhaustive Search**: Brute force involves checking all possible solutions to find the correct one. This means generating every possible configuration and evaluating each one to see if it meets the desired criteria.
2. **Simplicity**: Brute force algorithms are usually simple to implement, as they do not require complex logic or data structures. They are often used as a baseline or initial approach to problem-solving.

Brute Force

Key Characteristics of Brute Force

3. **Time Complexity**: Brute force algorithms can be very inefficient, especially for large input sizes. The time complexity is often exponential or factorial in nature, making them impractical for large problems.
4. **Guaranteed Solution**: If there is a solution to be found, a brute force approach will eventually find it, provided there is enough time and computational resources.

Brute Force

When to Use Brute Force

Advantages:

- Simplicity and ease of implementation.
- Provides a baseline for comparison with more complex algorithms.
- Useful for small problem sizes or when optimal solution is critical.

In the real world, brute force is often used when no smarter or more efficient methods are available e.g., when dealing with passwords, brute force involves trying every possible password until the correct one is found.

Disadvantages:

- Inefficient for large datasets.
- High computational cost.

Brute Force

Example 1: Finding Maximum Value in an Array

Problem: Find the maximum value in an array of integers.

Algorithm: - Initialize the maximum value.
- Iterate through the array and update the maximum value.

```
def find_max_brute_force(arr):
    max_value = arr[0]
    for num in arr:
        if num > max_value:
            max_value = num
    return max_value

arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(find_max_brute_force(arr))  # Output: 9
```

Brute Force

Example 1: Finding Maximum Value in an Array

Complexity Analysis

Time Complexity: $O(n)$

- This linear time complexity indicates that the algorithm scales well with larger input sizes, making it efficient for this particular task.

Discussion:

- Efficient for this specific problem but not representative of all brute force algorithms.

Brute Force

Example 2: Traveling Salesman Problem (TSP)

Problem: Find the shortest possible route that visits each city exactly once and returns to the origin city.

Algorithm:
- Generate all permutations of cities.
- Calculate the total distance for each permutation.
- Select the shortest distance.

```
def calculate_distance(route, distance_matrix):
    total_distance = 0
    for i in range(len(route) - 1):
            total_distance += distance_matrix[route[i]][route[i+1]]
    total_distance += distance_matrix[route[-1]][route[0]]  # Return to start
    return total_distance
```

Brute Force

Example 2: Traveling Salesman Problem (TSP)

```
def tsp_brute_force(cities, distance_matrix):
    min_distance = float('inf')
    best_route = None
    for route in itertools.permutations(cities):
        current_distance = calculate_distance(route, distance_matrix)
        if current_distance < min_distance:
            min_distance = current_distance
            best_route = route
    return best_route, min_distance
```

```
cities = [0, 1, 2, 3]
distance_matrix = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
```

Brute Force

Example 2: Traveling Salesman Problem (TSP)

Complexity Analysis

Time Complexity: $O(n!)$

- All possible permutations of the cities => for n cities, there are $n!$ permutations.
- For example, if there are 3 cities, the permutations would be 3! = 6: [A, B, C], [A, C, B], [B, A, C], [B, C, A], [C, A, B], [C, B, A].

Discussion: Infeasible for large numbers of cities due to factorial growth.

Brute Force

Candidate Solution: **Combination and Permutation**

- Often, the candidate set consists of permutation of a sequence, or a combination of a set
- Permutation of a sequence is an arrangement of a sequence
  - E.g., for a sequence [1,2,3], there are 6 permutations: [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]
- Combination of a set is a selection of members of the set
  - E.g., for a set {a,b,c}, there are 8 combinations of its members, {}, {a}, {b}, {c}, {a,b}, {b,c}, {a,c}, {a,b,c}
- Enumerating all combinations or permutation can be done easily by doing recursion

Brute Force

Combination Example

- Subset sum problem
- **Task:** find a subset of a given array such that its sum is K
- **Input:** An array A[1,…,n], an integer K
- **Output:** a set $\{i_1, i_2, \ldots, i_m\}$ such that
  - A[i_1] + A [i_2] + … + A[i_m] = K
  - 0 <= i_1< i_2< … < i_m<= n

Candidate Solution	**Set of candidate solution**	**Satisfaction condition**
$\{i_1, i_2, \ldots, i_m\}$	Power set of {1,2,…,N}	A[i_1] + A [i_2] + … + A[i_m] = K

Brute Force

Combination Example

- Ex1: A = [9,4,5], K = 9
  - Solution
    - {1}   (A[1] = 9)
    - {2,3}   (A[2] + A[3] = 9)

- Ex2: A = [10,40,30,20], K = 60
  - Solutions
    - {2,4}   (A[2] + A[4] = 60)
    - (1,3,4}   (A[1] + A[3] + A[4] = 60)

Brute Force

Permutation Example

- **Task:** find a path in a graph
- **Input:** A graph G = (V,E), two vertices p and q
- **Output:** A path in the graph that starts with p and ends with q

Candidate Solution	**Set of candidate solution**	**Satisfaction condition**
$\{V_1, V_2, \dots, V_k\}$	Every permutation of size 1…|V| of vertices	(V_i, V_{i+1}) is an edge for every i from 1 to k-1 V_1 = p V_k = q

Brute Force

Real-World Applications

- **Cryptography:** Brute force attacks to crack passwords.
- **Search Problems:** Exhaustive search in small datasets.
- **Optimization Problems:** Finding exact solutions when approximate ones are insufficient.

Optimizing Brute Force

- **Pruning:** Reduce the search space by eliminating impossible or suboptimal solutions early.
- **Heuristics:** Use rules of thumb to guide the search process.
- **Parallel Processing:** Distribute the search across multiple processors to speed up computation.

Brute Force

Case Study: Password Cracking

- **Description:** Using brute force to guess passwords by trying all possible combinations.
- **Challenges:** High computational cost, especially for complex passwords.
- **Mitigation:** Use of salt and hash functions to increase difficulty.

Case Study: Knapsack Problem

- **Problem:** Given a set of items, each with a weight and a value, determine the most valuable subset of items that can fit in a knapsack of limited capacity.
- **Brute Force Approach:** Evaluate all possible subsets of items to find the optimal solution.
- **Discussion:** Inefficient for large numbers of items due to exponential growth.

Brute Force

Brute Force in AI and Machine Learning

- **Example:** Hyperparameter tuning by exhaustive search.
- **Discussion:** Often impractical for large models or datasets, but useful as a starting point.

When to Avoid Brute Force

- **Large Datasets:** Infeasible due to time and resource constraints.
- **High Complexity Problems:** Better algorithms exist that are more efficient.
- **Scalability Requirements:** Need for solutions that scale with data size and complexity.

Brute Force

Alternatives to Brute Force

- **Greedy Algorithms:** Make locally optimal choices at each step.
- **Dynamic Programming:** Break down problems into simpler subproblems.
- **Heuristic Methods:** Use rules of thumb or educated guesses to find good enough solutions.
- **Backtracking:** Systematically explore and eliminate partial solutions.

Brute force is a fundamental but often inefficient method for problem-solving. Useful as a baseline or for small problems.