**1. Searching and Sorting**

**1.1 Searching Algorithms (การค้นหา)**

การค้นหาคือกระบวนการในการหา ตำแหน่งของข้อมูลที่ต้องการในโครงสร้างข้อมูล เช่น อาร์เรย์ (Array), ลิสต์ (List), หรือ Tree

**(1) Linear Search** (การค้นหาเชิงเส้น)

* วิธีการ: ตรวจสอบทีละตัวจากต้นจนถึงปลาย จนกว่าจะเจอค่าที่ต้องการ
* ข้อดี: ใช้งานง่าย ไม่จำเป็นต้องเรียงข้อมูลก่อน
* ข้อเสีย: ทำงานช้าเมื่อข้อมูลมีขนาดใหญ่ (Time Complexity: O(n))

**(2) Binary Search** (การค้นหาแบบทวิภาค)

วิธีการ: แบ่งข้อมูลครึ่งหนึ่งในแต่ละรอบ และตรวจสอบว่าเป้าหมายอยู่ในครึ่งไหน (ทำซ้ำไปเรื่อย ๆ)

ข้อดี: ทำงานได้เร็วกว่า Linear Search (Time Complexity: O(log n)

ข้อเสีย: ต้องเรียงข้อมูลก่อนจึงจะใช้ได้

ตัวอย่างวิธีการค้นหาข้อมูลจากอาร์เรย์

**(1) Linear Search**

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i # คืนค่าตำแหน่งที่เจอ

return -1 # ถ้าไม่เจอให้คืนค่า -1

arr = [10, 25, 40, 2, 5, 8]

target = 40

print("Linear Search:", linear\_search(arr, target))

**(2) Binary Search** (ต้องใช้กับอาร์เรย์ที่เรียงลำดับแล้ว)

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

arr = sorted([10, 25, 40, 2, 5, 8])

target = 40

print("Binary Search:", binary\_search(arr, target))

**1.2 Sorting Algorithms (การเรียงลำดับ)**

การเรียงลำดับคือ กระบวนการจัดเรียงข้อมูลจากค่าน้อยไปมาก (ascending) หรือค่ามากไปน้อย (descending)

**(1) Bubble Sort (บับเบิลซอร์ท)**

วิธีการ: เปรียบเทียบค่า 2 ตัวที่อยู่ติดกัน แล้วสลับที่หากลำดับไม่ถูกต้อง ทำซ้ำจนกว่าข้อมูลจะเรียงเสร็จ

ข้อดี: เข้าใจง่าย

ข้อเสีย: ช้ามาก (O(n2)O(n^2)O(n2))

**(2) Quick Sort (ควิกซอร์ท)**

วิธีการ: เลือกจุด Pivot แล้วแบ่งข้อมูลออกเป็น 2 ส่วน (มากกว่า Pivot และน้อยกว่า Pivot) แล้วเรียงแยกในแต่ละส่วน

ข้อดี: ทำงานเร็ว (O(n log n) โดยเฉลี่ย)

ข้อเสีย: กรณีแย่สุดทำงานช้า (O(n2)O(n^2)O(n2))

ตัวอย่างการเรียงลำดับข้อมูล

**(1) Bubble Sort**

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n - i - 1):

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j] # Swap

return arr

arr = [64, 34, 25, 12, 22, 11, 90]

print("Bubble Sort:", bubble\_sort(arr))

**(2) Quick Sort**

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

arr = [10, 7, 8, 9, 1, 5]

print("Quick Sort:", quick\_sort(arr))

**2. Hatching (ไม่มีโปรแกรม)**

Hatching หมายถึงการแบ่งพื้นที่ลงตารางโดยใช้กลยุทธ์บางอย่าง เช่น First-Fit, Best-Fit หรือ Next-Fit ในการแก้ปัญหาการจัดวางข้อมูลในพื้นที่ที่จำกัด เช่น การจองหน่วยความจำในคอมพิวเตอร์

ตัวอย่างโจทย์:

* มีหน่วยความจำว่างขนาด 10 ช่อง
* ใส่ชุดข้อมูลขนาด 3, 2, 5 ลงไปในตาราง
* ใช้วิธี First-Fit (ใส่ในช่องแรกที่พอเหมาะ)

ตารางเริ่มต้น: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

ใส่ขนาด 3: [ 3, 3, 3, 0, 0, 0, 0, 0, 0, 0 ]

ใส่ขนาด 2: [ 3, 3, 3, 2, 2, 0, 0, 0, 0, 0 ]

ใส่ขนาด 5: [ 3, 3, 3, 2, 2, 5, 5, 5, 5, 5 ]

**3. Tree (ต้นไม้)**

**3.1 Binary Search Tree (BST) คืออะไร?**

Binary Search Tree (BST) เป็น โครงสร้างข้อมูลต้นไม้ (Tree) ที่มีเงื่อนไขพิเศษ:

* ค่าที่น้อยกว่า อยู่ทางซ้าย
* ค่าที่มากกว่า อยู่ทางขวา

**3.2 Insert (การเพิ่มข้อมูล)**

เมื่อเพิ่มค่าใหม่ลงใน BST จะต้องเปรียบเทียบกับโหนดปัจจุบัน แล้วตัดสินใจว่าจะไปทางซ้ายหรือขวา

**3.2 Insert ลง Binary Search Tree (BST)**

class Node:

def \_\_init\_\_(self, key):

self.left = self.right = None

self.val = key

def insert(root, key):

if root is None:

return Node(key)

if key < root.val:

root.left = insert(root.left, key)

else:

root.right = insert(root.right, key)

return root

# ตัวอย่างการ Insert

root = None

keys = [50, 30, 70, 20, 40, 60, 80]

for key in keys:

root = insert(root, key)

**3.3 Delete (การลบข้อมูล)**

มี 3 กรณีหลัก:

* ไม่มีลูก → ลบออกได้ทันที
* มีลูก 1 ตัว → ให้ลูกขึ้นมาแทน
* มีลูก 2 ตัว → หา "ค่าต่ำสุด" ใน subtree ขวา มาแทนค่าที่ถูกลบ

**3.3 Delete Node ใน BST**

def min\_value\_node(node):

current = node

while current.left is not None:

current = current.left

return current

def delete(root, key):

if root is None:

return root

if key < root.val:

root.left = delete(root.left, key)

elif key > root.val:

root.right = delete(root.right, key)

else:

if root.left is None:

return root.right

elif root.right is None:

return root.left

temp = min\_value\_node(root.right)

root.val = temp.val

root.right = delete(root.right, temp.val)

return root

**3.4 Tree Traversal (การท่องไปในต้นไม้)**

1. Inorder (LNR) → ซ้าย → โหนดปัจจุบัน → ขวา (ใช้ใน BST เพื่อให้ข้อมูลเรียงลำดับ)
2. Preorder (NLR) → โหนดปัจจุบัน → ซ้าย → ขวา
3. Postorder (LRN) → ซ้าย → ขวา → โหนดปัจจุบัน

### ****3.4 การเดิน Tree (Traversal)****

def inorder(root):

return inorder(root.left) + [root.val] + inorder(root.right) if root else []

print("Inorder Traversal:", inorder(root))

**4. Graph (กราฟ)**

**4.1 กราฟคืออะไร?**

กราฟ (Graph) เป็นโครงสร้างข้อมูลที่ใช้แทน ความสัมพันธ์ระหว่างวัตถุหรือข้อมูล มีองค์ประกอบหลักคือ:

* Vertex (จุด) เช่น เมือง สถานี
* Edge (เส้นเชื่อม) เช่น ถนน เส้นทางบิน
* Weight (น้ำหนักของเส้น) เช่น ระยะทาง ค่าตั๋วเครื่องบิน

**4.2 ประเภทของกราฟ**

1. Directed Graph (กราฟมีทิศทาง) → เส้นเชื่อมมีทิศทาง
2. Undirected Graph (กราฟไม่มีทิศทาง) → เส้นเชื่อมไม่มีทิศทาง
3. Weighted Graph (กราฟมีน้ำหนัก) → แต่ละเส้นเชื่อมมีค่า เช่น ค่าใช้จ่าย หรือระยะทาง

ใช้ networkx และ matplotlib สำหรับการจัดการกราฟ

import networkx as nx

import matplotlib.pyplot as plt

# สร้างกราฟ

G = nx.Graph()

G.add\_edge("A", "B", weight=4)

G.add\_edge("A", "C", weight=2)

G.add\_edge("B", "C", weight=5)

G.add\_edge("B", "D", weight=10)

Gladded("C", "D", weight=3)

# วาดกราฟ

pos = nx.spring\_layout(G)

weights = nx.get\_edge\_attributes(G, 'weight')

nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=2000, font\_size=12)

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=weights)

plt.show()

**4.3 Graph Traversal (การท่องไปในกราฟ)**

1. Breadth-First Search (BFS)
   * ใช้ Queue (FIFO)
   * ค้นหาจากโหนดที่อยู่ใกล้สุดก่อน
   * ใช้ในการหาเส้นทางที่สั้นสุด
   * Complexity: O(V+E

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=" ")

visited.add(node)

queue.extend(graph[node] - visited)

graph = {

"A": {"B", "C"},

"B": {"A", "D"},

"C": {"A", "D"},

"D": {"B", "C"}

}

print("BFS:", end=" ")

bfs(graph, "A")

1. Depth-First Search (DFS)
   * ใช้ Stack (หรือ Recursive Call)
   * ค้นหาต่อไปเรื่อย ๆ จนกว่าจะสุดทางก่อนย้อนกลับ
   * ใช้ในการตรวจจับ วงจร (Cycle Detection)
   * Complexity: O(V+E)

def dfs(graph, node, visited=None):

if visited is None:

visited = set()

if node not in visited:

print(node, end=" ")

visited.add(node)

for neighbor in graph[node]:

dfs(graph, neighbor, visited)

print("\nDFS:", end=" ")

dfs(graph, "A")

**Code เพิ่ม - ลดจังหวัด**



