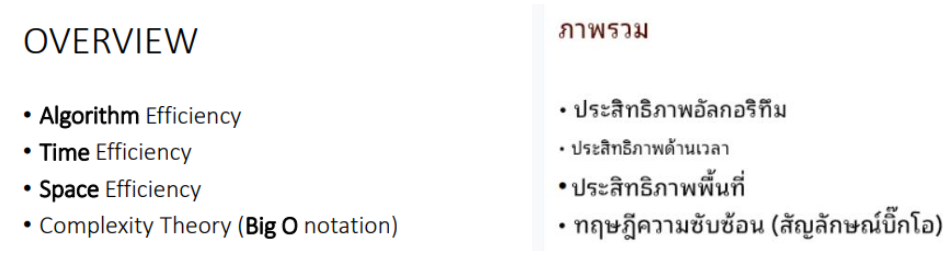
Chapter 3 Introduction to Efficiency



หน้า 1 ภาพรวมของความมีประสิทธิภาพ

- **Algorithm Efficiency** ทำความเข้าใจว่าประสิทธิภาพของอัลกอริทึมมีความสำคัญต่อการเลือกและการออกแบบโซลูชันทางคอมพิวเตอร์อย่างไร
- **Time Efficiency & Space Efficiency** รู้จักวัดประสิทธิภาพด้านเวลาและพื้นที่ของอัลกอริทึม ซึ่งส่งผลต่อการเลือกใช้งานในสถานการณ์ต่างๆ
- **Complexity Theory** เข้าใจความสำคัญของทฤษฎีความซับซ้อนและการใช้ Big O ในการวิเคราะห์ความซับซ้อนของอัลกอริทึม

หน้า 2-3 ความมีประสิทธิภาพของอัลกอริทึม

- การวัดประสิทธิภาพของอัลกอริทึม โดยพิจารณาจากทรัพยากรที่ใช้ เช่น เวลา, หน่วยความจำ และจำนวนการดำเนินการที่จำเป็น

หน้า 4-6 อัลกอริทึมคืออะไร?

- อธิบายคำจำกัดความของอัลกอริทึมและการใช้งานของมันในการแก้ปัญหาการคำนวณ

อัลกอริทึม (Algorithm)

- ถูกนิยามว่าเป็นชุดของขั้นตอนหรือกฎเกณฑ์ที่ใช้ในการแก้ไขปัญหาการคำนวณหรือการประมวลผลที่เฉพาะเจาะจง อัลกอริทึมช่วยให้สามารถแปลงข้อมูลนำเข้า (input) ผ่านชุดการดำเนินการที่กำหนดเพื่อได้ผลลัพธ์ (output) ที่ต้องการ
- ตัวอย่างของอัลกอริทึม เช่น อัลกอริทึมการค้นหา, อัลกอริทึมการเรียงลำดับ, หรืออัลกอริทึมการคำนวณพาธที่สั้นที่สุดในกราฟ

ปัญหาการคำนวณ (Computational Problem)

- เป็นชุดของคำถามหรือปัญหาที่อาจถูกแก้ไขโดยการใช้คอมพิวเตอร์ ปัญหาเหล่านี้มักจะมีลักษณะที่สามารถนิยามได้อย่างชัดเจนและมีเงื่อนไขหรือข้อมูลนำเข้าที่กำหนดไว้
- ตัวอย่างของปัญหาการคำนวณ เช่น การหาค่าใหญ่สุดหรือค่าน้อยสุด, การตรวจสอบว่าข้อมูลหนึ่งๆ อยู่ภายในข้อมูลชุดใหญ่หรือไม่, หรือการแปลงรูปแบบข้อมูลจากแบบหนึ่งไปอีกแบบหนึ่ง

หน้า 7-8 ความมีประสิทธิภาพของอัลกอริทึม

- สำรวจตัวชี้วัดที่สำคัญสำหรับประสิทธิภาพของอัลกอริทึม รวมถึงการวัดเวลา, การใช้หน่วยความจำ และทฤษฎีความซับซ้อน

คำนิยามของประสิทธิภาพของอัลกอริทึม

- ประสิทธิภาพของอัลกอริทึมคือการศึกษาเกี่ยวกับปริมาณทรัพยากรที่อัลกอริทึมใช้ในการแก้ปัญหา โดยทรัพยากรสามารถหมายถึงเวลา, หน่วยความจำ หรือจำนวนขั้นตอน/การดำเนินการที่จำเป็น
- หลักการพื้นฐานคือยิ่งอัลกอริทึมใช้ทรัพยากรน้อยลงเท่าไร ก็ยิ่งมีประสิทธิภาพมากขึ้นเท่านั้น

การวัดประสิทธิภาพ:

- **ประสิทธิภาพด้านเวลา (Time Efficiency)**: วัดจากเวลาที่อัลกอริทึมต้องการในการแก้ไขปัญหา
- **ประสิทธิภาพด้านพื้นที่ (Space Efficiency)**: วัดจากปริมาณหน่วยความจำที่อัลกอริทึมใช้ในการแก้ไขปัญหา
- ประสิทธิภาพเหล่านี้มักถูกเปรียบเทียบโดยใช้ทรัพยากรที่อัลกอริทึมต้องการ ซึ่งช่วยในการตัดสินใจเลือกอัลกอริทึมที่เหมาะสมสำหรับการใช้งานเฉพาะอย่าง

ทฤษฎีความซับซ้อน (Complexity Theory):

- เป็นการศึกษาวิธีการที่อัลกอริทึมแก้ปัญหาและผลกระทบของมันต่อการดำเนินงานโดยรวม โดยอาศัยการนับจำนวนคำสั่งหรือขั้นตอนที่อัลกอริทึมดำเนินการ
- **ฟังก์ชันต้นทุน (Cost Function)**: เป็นเครื่องมือที่ช่วยในการวัดปริมาณทรัพยากรที่อัลกอริทึมใช้ โดยพิจารณาจากขนาดของปัญหา nnn ตัวอย่างเช่น จำนวนคำสั่งที่ต้องดำเนินการภายในโปรแกรม

วิธีต่างๆ ในการกลับสตริง (reverse a string) ในการเขียนโปรแกรม

A. การใช้ Slicing เพื่อสร้างสำเนาของสตริงที่กลับด้าน

- ใช้เทคนิค slicing โดยใช้ syntax ที่เรียบง่าย [::-1] เพื่อสร้างสตริงใหม่ที่มีลำดับของตัวอักษรกลับด้านจากต้นฉบับ

```
# ประกาศฟังก์ชันที่ชื่อ reverse_slicing ที่รับพารามิเตอร์ s (สตริงที่ต้องการกลับด้าน)
def reverse_slicing(s):
    # ส่งคืนสตริง s ที่ได้จากการกลับด้านโดยใช้ slicing
    return s[::-1]  # ใช้ slicing เพื่อกลับด้านสตริง, โดย start:stop:step ไม่ระบุ (กลับสตริงทั้งหมด)

# ตัวแปร input_str มีสตริงที่ต้องการกลับด้าน
input_str = 'INE-KMUTNB'  # ตัวอย่างข้อมูลนำเข้าที่จะทดสอบฟังก์ชัน

# ส่วนของโค้ดนี้จะถูกเรียกใช้เมื่อโมดูลนี้ถูกเรียกใช้เป็นโปรแกรมหลัก
if __name__ == "__main__":
    # พิมพ์ผลลัพธ์จากการเรียกใช้ฟังก์ชัน reverse_slicing ที่ส่ง input_str เป็นอาร์กิวเมนต์
    print('Reverse String using slicing -', reverse_slicing(input_str))
    # คำสั่ง print จะแสดงสตริง "Reverse String using slicing -" ตามด้วยผลลัพธ์ของฟังก์ชัน
```

อธิบายผลลัพธ์

- ข้อความ "Reverse String using slicing - " เป็นส่วนที่กำหนดไว้ในคำสั่ง print สำหรับอธิบายว่าผลลัพธ์นี้ได้มาจากการใช้วิธีการ slicing ในการกลับสตริง
- ตามด้วย 'BNTUMK-ENI' ซึ่งเป็นสตริงที่กลับด้านจาก 'INE-KMUTNB' แสดงถึงสตริงต้นทางที่ได้รับการกลับด้านอย่างถูกต้อง

B. การใช้ for loop และการเพิ่มตัวอักษรในลำดับกลับด้าน

- ใช้ for loop ในการวนซ้ำผ่านแต่ละตัวอักษรของสตริง แล้วเพิ่มตัวอักษรนั้นๆ ลงในสตริงใหม่เริ่มจากตัวท้ายสุดไปหาตัวแรก

```
# ฟังก์ชันสำหรับกลับด้านสตริงโดยใช้ for loop
def reverse_for_loop(s):
    s1 = ''  # สร้างสตริงเปล่าสำหรับเก็บผลลัพธ์
    for c in s:  # วนลูปผ่านแต่ละตัวอักษรในสตริง s
        s1 = c + s1  # เพิ่มตัวอักษรทีละตัวในตำแหน่งแรกของ s1
    return s1  # คืนค่าสตริงที่ได้จากการกลับด้าน

# สตริงที่ต้องการกลับด้าน
input_str = 'INE-KMUTNB'

# ส่วนนี้จะถูกเรียกเมื่อไฟล์นี้รันโดยตรงไม่ได้ถูก import เป็น module
if __name__ == "__main__":
    # แสดงผลลัพธ์ของการกลับด้านสตริง
    print('Reverse String using for loop -', reverse_for_loop(input_str))
```

คำอธิบายผลลัพธ์

- **Reverse String using for loop -** เป็นส่วนของข้อความที่ระบุว่าผลลัพธ์ที่ได้มาจากการใช้งานฟังก์ชัน reverse_for_loop.
- **BNTUMK-ENI** คือสตริง INE-KMUTNB ที่ได้รับการกลับด้าน.

การกลับด้านสตริงโดยใช้ for loop เป็นวิธีที่ชัดเจนและง่ายต่อการติดตามกระบวนการ และยังเป็นวิธีที่ดีในการแสดงวิธีการทำงานของ loops ในภาษา Python

```
# ประกาศฟังก์ชันกลับสตริงโดยใช้ while loop
def reverse_while_loop(s):
    s1 = ''  # สร้างสตริงเปล่าสำหรับเก็บผลลัพธ์ที่กลับด้าน
    length = len(s) - 1  # กำหนดตัวแปร length เป็นดัชนีสุดท้ายของสตริง s

    # วนลูปตราบใดที่ length ไม่ต่ำกว่า 0
    while length >= 0:
        s1 = s1 + s[length]  # เพิ่มตัวอักษรจากด้านท้ายของ s ไปยัง s1
        length = length - 1  # ลดค่าดัชนีเพื่อย้อนกลับไปยังตัวอักษรก่อนหน้า

    return s1  # ส่งคืนสตริงที่กลับด้านแล้ว

# ตัวแปรสำหรับการทดสอบฟังก์ชันกลับสตริง
input_str = 'INE-KMUTNB'  # สตริงที่ต้องการกลับด้าน

# ส่วนของโค้ดนี้จะถูกเรียกใช้เมื่อไฟล์นี้รันโดยตรงไม่ได้ถูก import เป็น module
if __name__ == "__main__":
    # แสดงผลลัพธ์ของการกลับด้านสตริง
    print('Reverse String using while loop -', reverse_while_loop(input_str))
```

C. การใช้ while loop ในการวนซ้ำตัวอักษรของสตริงและเพิ่มเข้าไป

- ใช้ while loop ในการเข้าถึงแต่ละตัวอักษรของสตริงจากท้ายสุดกลับมาหาต้นสุด และนำมาเพิ่มเข้าไปในสตริงใหม่

```
# ประกาศฟังก์ชันกลับสตริงโดยใช้ while loop
def reverse_while_loop(s):
    s1 = ''  # สร้างสตริงเปล่าสำหรับเก็บผลลัพธ์ที่กลับด้าน
    length = len(s) - 1  # กำหนดตัวแปร length เป็นดัชนีสุดท้ายของสตริง s

    # วนลูปตราบใดที่ length ไม่ต่ำกว่า 0
    while length >= 0:
        s1 = s1 + s[length]  # เพิ่มตัวอักษรจากด้านท้ายของ s ไปยัง s1
        length = length - 1  # ลดค่าดัชนีเพื่อย้อนกลับไปยังตัวอักษรก่อนหน้า

    return s1  # ส่งคืนสตริงที่กลับด้านแล้ว

# ตัวแปรสำหรับการทดสอบฟังก์ชันกลับสตริง
input_str = 'INE-KMUTNB'  # สตริงที่ต้องการกลับด้าน

# ส่วนของโค้ดนี้จะถูกเรียกใช้เมื่อไฟล์นี้รันโดยตรงไม่ได้ถูก import เป็น module
if __name__ == "__main__":
    # แสดงผลลัพธ์ของการกลับด้านสตริง
    print('Reverse String using while loop -', reverse_while_loop(input_str))
```

เมื่อโค้ดถูกเรียกใช้, ผลลัพธ์จะแสดงข้อความ:

```arduino
Reverse String using while loop - BNTUMK-ENI
```

- สตริง `'INE-KMUTNB'` จะกลับด้านเป็น `'BNTUMK-ENI'`.
- คำสั่ง `print` แสดงผลลัพธ์นี้, บ่งบอกว่าการกลับสตริงด้วยวิธี `while loop` ทำงานได้อย่างถูกต้อง.

การใช้ `while loop` เพื่อกลับสตริงเป็นวิธีที่ชัดเจนและง่ายต่อการเข้าใจในการจัดการกับสตริงในลำดับย้อนกลับ, เหมาะสำหรับการประยุกต์ใช้ในหลายๆ สถานการณ์.

D. การใช้ฟังก์ชัน string join() ร่วมกับ reversed() iterator

- ใช้ฟังก์ชัน join() ร่วมกับ reversed() ซึ่งเป็น iterator ที่ได้รับสตริงและกลับลำดับของมัน ช่วยให้สามารถรวมตัวอักษรที่ได้จากการกลับด้านเข้าด้วยกันเป็นสตริงเดียว

```python
# ประกาศฟังก์ชันที่ชื่อ reverse_str_join สำหรับกลับด้านสตริง
def reverse_str_join(s):
    # ใช้เมธอด join() ร่วมกับฟังก์ชัน reversed() เพื่อกลับด้านสตริง
    s1 = ''.join(reversed(s))  # reversed(s) คือการสร้าง iterator ที่ย้อนกลับสตริง s, ''.join() คือการรวมสตริงจาก iterator นั้น
    return s1  # คืนค่าสตริงที่กลับด้านแล้ว

# สตริงที่ต้องการกลับด้าน
input_str = 'INE-KMUTNB'  # กำหนดสตริงเริ่มต้น

# ตรวจสอบว่าโมดูลถูกเรียกใช้เป็นสคริปต์หลักหรือไม่
if __name__ == "__main__":
    # แสดงผลลัพธ์ของการกลับด้านสตริง
    print('Reverse String using sting join -', reverse_str_join(input_str))  # พิมพ์ผลลัพธ์ที่ได้จากฟังก์ชัน reverse_str_join
```

ผลลัพธ์ที่แสดง:

เมื่อโค้ดถูกเรียกใช้, ผลลัพธ์จะแสดงข้อความ:

```vbnet
Reverse String using sting join - BNTUMK-ENI
```

- คำสั่ง `print` แสดงว่าการกลับสตริงด้วยวิธีการใช้ `string join` และ `reversed()` ทำงานได้ถูกต้องและเป็นไปตามที่คาดการณ์ไว้, โดยสตริง `'INE-KMUTNB'` ถูกกลับด้านเป็น `'BNTUMK-ENI'`.

E. การสร้าง list จากสตริงและเรียกใช้ฟังก์ชัน reverse()

- แปลงสตริงเป็น list ของตัวอักษร แล้วใช้ฟังก์ชัน reverse() ที่เป็น built-in ของ list เพื่อกลับลำดับของตัวอักษรใน list นั้น

```python
# ประกาศฟังก์ชันที่ชื่อ reverse_list สำหรับกลับด้านสตริง
def reverse_list(s):
    temp_list = list(s)  # แปลงสตริง s เป็น list เพื่อให้สามารถใช้เมธอด reverse() ได้
    temp_list.reverse()  # ใช้เมธอด reverse() เพื่อกลับด้านข้อมูลใน list
    return ''.join(temp_list)  # ใช้เมธอด join() เพื่อรวมสมาชิกใน list กลับเป็นสตริง

# สตริงที่ต้องการกลับด้าน
input_str = 'INE-KMUTNB'  # กำหนดสตริงเริ่มต้น

# ตรวจสอบว่าโมดูลถูกเรียกใช้เป็นสคริปต์หลักหรือไม่
if __name__ == "__main__":
    # แสดงผลลัพธ์ของการกลับด้านสตริง
    print('Reverse String using list -', reverse_list(input_str))  # พิมพ์ผลลัพธ์ที่ได้จากฟังก์ชัน reverse_list
```

ผลลัพธ์ที่แสดง:

เมื่อโค้ดถูกเรียกใช้, ผลลัพธ์จะแสดงข้อความ:

```arduino
Reverse String using list - BNTUMK-ENI
```

- คำสั่ง `print` แสดงว่าการกลับสตริงด้วยวิธีการใช้ list และ `reverse()` ทำงานได้อย่างถูกต้องและได้ผลลัพธ์ที่คาดหวัง, โดยสตริง `'INE-KMUTNB'` ถูกกลับด้านเป็น `'BNTUMK-ENI'`.

F. การใช้ Recursion

- ใช้เทคนิคการเรียกฟังก์ชันตัวเอง (recursion) เพื่อแยกสตริงออกเป็นหน่วยย่อยๆ และเรียงลำดับกลับด้านจากตัวท้ายสุดไปตัวแรก

```python
# ประกาศฟังก์ชันกลับสตริงโดยใช้เทคนิคการเรียกฟังก์ชันตัวเอง (recursion)
def reverse_recursion(s):
    if len(s) == 0:  # ตรวจสอบว่าสตริงว่างหรือไม่
        return s  # ถ้าสตริงว่าง ก็คืนค่าสตริงนั้น (กรณีพื้นฐาน)
    else:
        # เรียกฟังก์ชันตัวเองด้วยส่วนที่เหลือของสตริงหลังตัดตัวแรกออก แล้วเพิ่มตัวแรกไปที่ตำแหน่งสุดท้าย
        return reverse_recursion(s[1:]) + s[0]  # s[1:] คือสตริงที่เหลือหลังตัดตัวแรกออก, s[0] คือตัวแรกของสตริง

# สตริงที่ต้องการกลับด้าน
input_str = 'INE-KMUTNB'  # กำหนดสตริงเริ่มต้น

# ตรวจสอบว่าโมดูลถูกเรียกใช้เป็นสคริปต์หลักหรือไม่
if __name__ == "__main__":
    # แสดงผลลัพธ์ของการกลับด้านสตริง
    print('Reverse String using recursive -', reverse_recursion(input_str))  # พิมพ์ผลลัพธ์ที่ได้จากฟังก์ชัน reverse_recursion
```

ผลลัพธ์ที่แสดง:

เมื่อโค้ดถูกเรียกใช้, ผลลัพธ์จะแสดงข้อความ:

```arduino
Reverse String using recursive - BNTUMK-ENI
```

- **คำสั่ง `print` แสดงว่าการกลับสตริงด้วยวิธีการเรียกฟังก์ชันตัวเอง** (recursion) **ทำงานได้อย่างถูกต้องและได้ผลลัพธ์ที่คาดหวัง, โดยสตริง `'INE-KMUTNB'` ถูกกลับด้านเป็น `'BNTUMK-ENI'`**.

แต่ละวิธีมีข้อดีและข้อเสียที่แตกต่างกัน เช่น เรื่องของประสิทธิภาพด้านเวลา, ความซับซ้อนของโค้ด และทรัพยากรหน่วยความจำที่ใช้ การเลือกใช้วิธีใดวิธีหนึ่งขึ้นอยู่กับสถานการณ์หรือเงื่อนไขของปัญหาที่ต้องการแก้ไข

หน้า 18-22 ประสิทธิภาพทางเวลาและพื้นที่

- การอธิบายและวิเคราะห์ความซับซ้อนของการคำนวณที่เกี่ยวข้องกับเวลาและหน่วยความจำที่อัลกอริทึมใช้

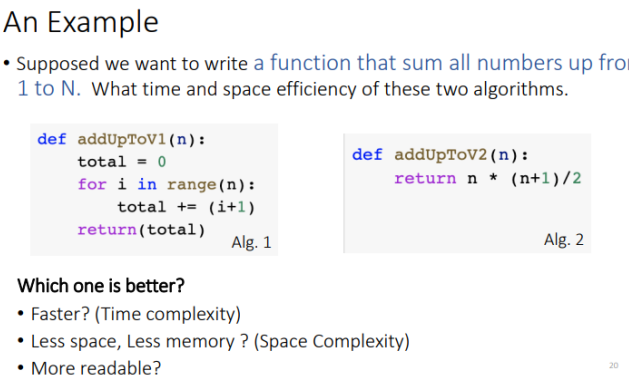
ประสิทธิภาพด้านเวลาและพื้นที่ (Time and Space Efficiency)

- หน้านี้อธิบายถึงความสำคัญของการวัดประสิทธิภาพด้านเวลาและพื้นที่ในการใช้งานอัลกอริทึม ความสามารถในการประมวลผลอย่างรวดเร็วและใช้หน่วยความจำอย่างมีประสิทธิภาพเป็นสิ่งสำคัญสำหรับการออกแบบระบบหรือโปรแกรมที่มีประสิทธิผล
- การวัดประสิทธิภาพเหล่านี้ช่วยให้ผู้พัฒนาสามารถตัดสินใจได้ว่าอัลกอริทึมใดเหมาะสมที่สุดสำหรับการใช้งานในสภาวะต่างๆ โดยพิจารณาจากเงื่อนไขของระบบหรืออุปกรณ์ที่ใช้

ประสิทธิภาพของอัลกอริทึม ใครสนใจ? (Algorithm Efficiency WHO CARE?)

- ส่วนนี้ชี้ให้เห็นถึงการสนทนาและการวิเคราะห์ประสิทธิภาพของอัลกอริทึมว่ามีความสำคัญอย่างไรในแง่ของการประเมินผลการทำงานของโค้ด
- มุ่งเน้นไปที่:
    - **การประเมินผลการทำงานของโค้ด**: การวิเคราะห์ว่าโค้ดทำงานได้ดีแค่ไหน มีประสิทธิภาพอย่างไร
    - **การเปรียบเทียบระหว่างวิธีการต่างๆ**: พิจารณาถึงข้อดีข้อเสียระหว่างวิธีการที่แตกต่างกัน

- **การระบุส่วนของโค้ดที่ไม่มีประสิทธิภาพ**: ช่วยให้สามารถเห็นจุดที่อาจต้องปรับปรุงเพื่อให้โค้ดทำงานได้ดียิ่งขึ้น



Algorithm 1: addUpToV1(n)

- ใช้ for loop ในการวนรอบตัวเลขจาก 0 ถึง n-1 และเพิ่มค่า i+1 ไปยังตัวแปร total ที่ใช้เก็บผลรวม. หลังจากนั้นจึงคืนค่า total.
- Time Complexity: O(n) เพราะต้องวนลูป n ครั้ง
- Space Complexity: O(1) เพราะใช้พื้นที่เก็บข้อมูลคงที่ (ตัวแปรเดียวคือ total)

Algorithm 2: addUpToV2(n)

- ใช้สูตรคณิตศาสตร์ (n*(n+1))/2 ในการคำนวณผลรวมของตัวเลขจาก 1 ถึง n และคืนค่าผลลัพธ์ทันที.
- Time Complexity: O(1) เพราะสูตรนี้ไม่ต้องการ loop หรือ recursion
- Space Complexity: O(1) เช่นเดียวกันเพราะไม่ต้องการพื้นที่เพิ่มเติมในการคำนวณนอกจากพื้นที่สำหรับเก็บค่าตัวแปร n และผลลัพธ์

ผลลัพธ์ที่คาดว่าจะได้:

- จากการทดสอบด้วยค่า `n = 5000000`, `addUpToV1(n)` และ `addUpToV2(n)` ทั้งคู่คำนวณได้ผลลัพธ์ที่ถูกต้องเหมือนกันคือ `12500002500000`.
- อย่างไรก็ตาม, เวลาที่ใช้ในการทำงานของ `addUpToV1(n)` มากกว่ามาก ประมาณ `69.85998153686523` มิลลิวินาที, ในขณะที่ `addUpToV2(n)` ใช้เวลาเพียง `0.9305477142333984` มิลลิวินาที.

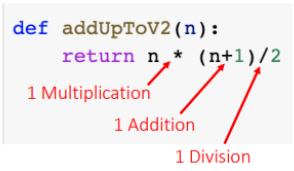
สรุป:

- **Algorithm 1** (`addUpToV1`) มีประสิทธิภาพเวลาแย่กว่ามากเมื่อเทียบกับ **Algorithm 2** (`addUpToV2`) ที่ใช้เวลาน้อยกว่ามากในการคำนวณ. ทั้งนี้เป็นเพราะว่า Algorithm 1 ต้องวนลูปหลายครั้งขณะที่ Algorithm 2 ใช้สูตรคณิตศาสตร์ในการคำนวณผลลัพธ์ทันที.

```python
import time  # นำเข้าโมดูล time สำหรับการวัดเวลา

# ฟังก์ชันที่ 1: ใช้ for loop ในการคำนวณผลรวม
def addUpToV1(n):
    total = 0  # ตั้งตัวแปรสำหรับเก็บผลรวม
    for i in range(n):  # วนรอบตั้งแต่ 0 ถึง n-1
        total += (i+1)  # เพิ่ม i+1 เข้าไปใน total เพราะ range จะเริ่มที่ 0
    return total  # คืนค่าผลรวม

# ฟัง (function) def addUpToV2(n: Any) -> Any
def addUpToV2(n):
    return n * (n+1) / 2  # คืนค่าผลรวมโดยตรงโดยใช้สูตรการหาผลรวม

# รับค่า n จากผู้ใช้
n = int(input('Input Value:'))

# วัดเวลาการทำงานของฟังก์ชันที่ 1
start = time.time()  # เริ่มต้นการนับเวลา
print('answer V1:', addUpToV1(n))  # แสดงผลลัพธ์ของฟังก์ชันที่ 1
print('time V1:', (time.time() - start) * 1000)  # แสดงเวลาที่ใช้ในการคำนวณ (มิลลิวินาที)

# วัดเวลาการทำงานของฟังก์ชันที่ 2
start = time.time()  # เริ่มต้นการนับเวลาใหม่
print('answer V2:', addUpToV2(n))  # แสดงผลลัพธ์ของฟังก์ชันที่ 2
print('time V2:', (time.time() - start) * 1000)  # แสดงเวลาที่ใช้ในการคำนวณ (มิลลิวินาที)
```

อธิบายและเปรียบเทียบจำนวนการดำเนินการ (operations) ที่เกิดขึ้นภายในสองฟังก์ชันที่มีจุดประสงค์เดียวกันในการคำนวณผลรวมของตัวเลขตั้งแต่ 1 ถึง n แต่ใช้วิธีการที่แตกต่างกัน

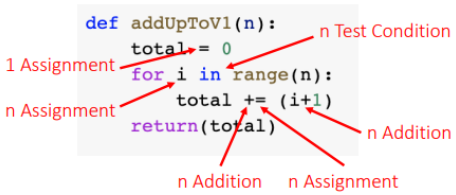
Counting Operations



1. ฟังก์ชัน addUpToV2(n) ที่ใช้สูตรคณิตศาสตร์

- การดำเนินการที่ใช้:
    - 1 Multiplication: n * (n+1) เป็นการคูณ n ด้วย n+1
    - 1 Addition: (n+1) คือการเพิ่มค่า 1 กับ n
    - 1 Division: (n*(n+1))/2 คือการหารผลลัพธ์ของการคูณด้วย 2
- จำนวนการดำเนินการรวม: 3 การดำเนินการ

Counting Operations



2. ฟังก์ชัน addUpToV1(n) ที่ใช้ลูป

- การดำเนินการที่ใช้:
    - 1 Assignment: กำหนดค่าเริ่มต้น total = 0
    - n Assignments ในลูป: แต่ละรอบของลูปจะมีการกำหนดค่า total
    - n Additions ในลูป: แต่ละรอบของลูปจะมีการเพิ่ม i+1 เข้าไปใน total
    - 1 Test Condition ในลูป: ตรวจสอบเงื่อนไขลูปว่ายังควรทำงานต่อหรือไม่
    - n+1 Tests for Loop Termination: การเช็คเงื่อนไขการหยุดลูปในแต่ละรอบ และเพิ่มเติมอีกครั้งหลังจบลูป

การวิเคราะห์:

- **ฟังก์ชัน addUpToV2** มีจำนวนการดำเนินการน้อยกว่ามาก และเป็นแบบคงที่ไม่ขึ้นกับ n (O(1)).
- **ฟังก์ชัน addUpToV1** มีจำนวนการดำเนินการขึ้นอยู่กับค่า n ทำให้ความซับซ้อนเวลาเป็น O(n) ซึ่งหมายความว่าเวลาที่ใช้จะเพิ่มขึ้นตามค่า n.

หน้า 23-29 ทฤษฎีความซับซ้อน

- การนำเสนอ Big O notation ซึ่งเป็นวิธีการวัดความซับซ้อนของอัลกอริทึมอย่างละเอียด และการนำไปใช้ในการประเมินและเปรียบเทียบอัลกอริทึม

คำนิยามของ Big O Notation

- **Big O Notation** เป็นวิธีทางคณิตศาสตร์ที่ใช้ในการอธิบายความซับซ้อนของอัลกอริทึมในแง่ของเวลาที่ใช้ในการดำเนินงานหรือจำนวนการดำเนินการที่จำเป็นตามขนาดของข้อมูลนำเข้า

- มันช่วยให้นักพัฒนาสามารถประเมินและเปรียบเทียบประสิทธิภาพของอัลกอริทึมต่างๆ ได้โดยไม่ต้องอิงจากฮาร์ดแวร์หรือรายละเอียดการประมวลผลที่แท้จริง

วัตถุประสงค์ของ Big O Notation

- **การประเมินเวลาการดำเนินงาน**: Big O Notation ช่วยให้เราสามารถคาดการณ์เวลาที่จำเป็นสำหรับการดำเนินงานของอัลกอริทึมได้ตามขนาดของข้อมูลนำเข้า ซึ่งมีความสำคัญในการออกแบบระบบที่ต้องรับมือกับข้อมูลขนาดใหญ่
- **การประเมินทรัพยากรที่ใช้**: นอกจากเวลาแล้ว Big O ยังช่วยให้เราประเมินหน่วยความจำหรือทรัพยากรการประมวลผลอื่นๆ ที่อัลกอริทึมต้องการใช้

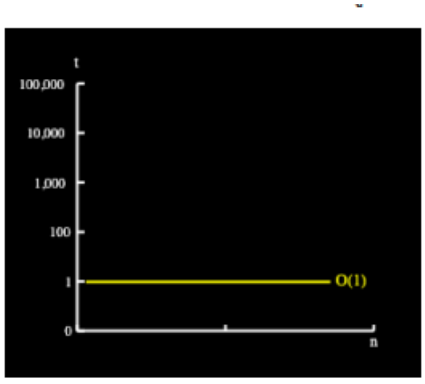
ประโยชน์ในการใช้งาน

- ใช้ในการวางแผนและออกแบบโปรแกรมเพื่อให้มีประสิทธิภาพและคาดการณ์ประสิทธิผลได้ดีขึ้น โดยเฉพาะอย่างยิ่งเมื่อต้องรับมือกับปริมาณข้อมูลที่เพิ่มขึ้น
- ช่วยให้นักพัฒนาเข้าใจถึงข้อจำกัดและข้อเสียของอัลกอริทึมต่างๆ เพื่อเลือกหรือปรับปรุงอัลกอริทึมให้เหมาะสมกับปัญหาที่ต้องแก้ไข

หน้า 30-39 ชนิดของ Big O Notation

- คำอธิบายชนิดต่างๆ ของ Big O Notation และตัวอย่างการประยุกต์ใช้ในสถานการณ์การเขียนโค้ดจริง

1. O(1) : Constant



สำหรับ Big O ตัวแรกนี้เป็น Big O ที่ดีที่สุด Input จะเป็นอะไรก็ใช้เวลาทำงานเพียง 1 ครั้งเท่าเดิมเสมอ เช่น การเช็คว่าตัวเลข Input เป็นเลขคู่หรือเลขคี่ เมื่อใส่ค่า 0 หรือ 999 ก็ mod ด้วย 2 ครั้งเดียวเท่ากัน

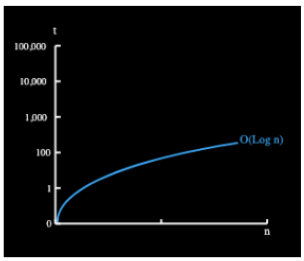
```python
def evenOrOdd(num):
    if (num % 2 == 0):  # ตรวจสอบว่า num หารด้วย 2 มีเศษเท่ากับ 0 หรือไม่
        print("even")  # ถ้าเศษเท่ากับ 0 แสดงว่าเป็นจำนวนคู่
    else:
        print("odd")  # ถ้าไม่ใช่แสดงว่าเป็นจำนวนคี่
```

โค้ดนี้ตรวจสอบค่า num % 2 เพื่อดูว่ามีเศษเท่ากับ 0 หรือไม่ ซึ่งเป็นการดำเนินการทางคณิตศาสตร์ที่ใช้เวลาคงที่ไม่ว่าค่า num จะเป็นเท่าใด

ความซับซ้อน O(1) - คงที่

- **ความหมาย** ความซับซ้อนของเวลาเป็น O(1) หมายความว่าเวลาการดำเนินการของอัลกอริธึมนั้นไม่ขึ้นกับขนาดของข้อมูลนำเข้า (n). นั่นคือ ไม่ว่าข้อมูลจะมากหรือน้อยเท่าไหร่ อัลกอริธึมก็จะทำงานใช้เวลาประมาณเท่าเดิมเสมอ.

2. **O(log n) : Logarithmic** เป็น Big O ที่ดีการทำงานจะลดจำนวนลู ปลงครึ่ งนึง ทุกครั้งที่ทำเสร็จไป 1 รอบ



ความซับซ้อน O(log n) - ลอการิทึม

- **ความหมาย:** ความซับซ้อนของเวลาเป็น O(log n) หมายความว่าเวลาการดำเนินการของอัลกอริธึมนั้นเพิ่มขึ้นอย่างช้าๆ ตามขนาดของข้อมูลนำเข้า (n) และมักจะเกี่ยวข้องกับการตัดครึ่งหนึ่งของข้อมูลในแต่ละขั้นตอนของอัลกอริธึม (เช่น การค้นหาทวิภาค)

ตัวอย่างการค้นหาทวิภาค (Binary Search)

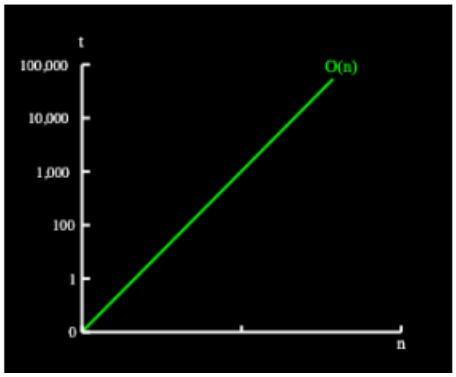
- Binary Search เป็นวิธีการค้นหาที่เริ่มจากการตรวจสอบข้อมูลตรงกลางของอาร์เรย์:
    - ถ้าข้อมูลที่ต้องการหาตรงกับข้อมูลตรงกลาง การค้นหาจะสิ้นสุดทันที.
    - ถ้าข้อมูลที่ต้องการหาน้อยกว่าข้อมูลตรงกลาง การค้นหาจะดำเนินต่อในส่วนครึ่งแรกของอาร์เรย์.
    - ถ้าข้อมูลที่ต้องการหามากกว่าข้อมูลตรงกลาง การค้นหาจะดำเนินต่อในส่วนครึ่งหลังของอาร์เรย์.
- ในแต่ละขั้นตอนของการค้นหา ขนาดของช่วงการค้นหาจะลดลงเป็นครึ่งหนึ่ง ทำให้จำนวนขั้นตอนที่ต้องดำเนินการเพื่อหาข้อมูลนั้นขึ้นอยู่กับลอการิทึมของขนาดข้อมูลนำเข้า (log n)

```python
def binarySearch(arr, value, first, last):
    if last >= first:  # เงื่อนไขว่ายังมีส่วนที่จะค้นหาหรือไม่
        mid = first + (last - first) // 2  # คำนวณหาตำแหน่งกลางของส่วนที่ต้องการค้นหา
        if arr[mid] == value:  # ถ้าค่าที่ตำแหน่งกลางตรงกับค่าที่ต้องการค้นหา
            return mid  # คืนค่าตำแหน่งนั้นออกไป
        elif arr[mid] > value:  # ถ้าค่าที่ตำแหน่งกลางมากกว่าค่าที่ต้องการค้นหา
            return binarySearch(arr, value, first, mid - 1)  # ค้นหาในช่วงครึ่งแรก
        else:  # ถ้าค่าที่ตำแหน่งกลางน้อยกว่าค่าที่ต้องการค้นหา
            return binarySearch(arr, value, mid + 1, last)  # ค้นหาในช่วงครึ่งหลัง
    else:
        return -1  # ถ้าไม่พบค่าที่ต้องการค้นหา คืนค่า -1

numArray = [1, 2, 7, 10, 22, 31]  # อาร์เรย์ที่มีการเรียงลำดับไว้แล้ว
number = 22  # ตัวเลขที่ต้องการค้นหา
result = binarySearch(numArray, number, 0, len(numArray) - 1)  # เรียกใช้ฟังก์ชันและเก็บผลลัพธ์
print(result)  # พิมพ์ผลลัพธ์
```

ผลลัพธ์ที่คาดว่าจะได้

- โค้ดนี้จะทำการค้นหาตัวเลข 22 ในอาร์เรย์ numArray ที่มีการเรียงลำดับไว้แล้ว
- โดยใช้การค้นหาทวิภาค ซึ่งจะคืนค่าดัชนีของตัวเลข 22 ในอาร์เรย์ (คาดว่าจะได้ดัชนีเป็น 4 เนื่องจาก 22 อยู่ที่ตำแหน่งที่ 5 ของอาร์เรย์ที่มีการนับตั้งแต่ 0)

3. **O(n) : Linear** Big O ชนิดนี้จะใช้เวลาในการทำงานเท่ากับ Input



ความซับซ้อน O(n) - เชิงเส้น

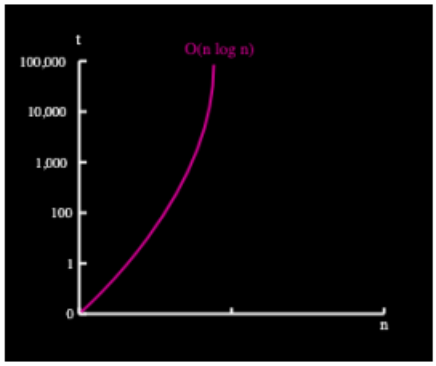
- ความหมาย: ความซับซ้อนของเวลาเป็น O(n) หมายความว่าเวลาการดำเนินการของอัลกอริธึมนั้นเพิ่มขึ้นเชิงเส้นตามขนาดของข้อมูลนำเข้า (n). หากข้อมูลนำเข้าเพิ่มขึ้นสองเท่า, เวลาในการดำเนินการก็จะเพิ่มขึ้นสองเท่าเช่นกัน

```python
def searchNumber(arr, value):
    for i in range(len(arr)):  # วนลูปผ่านทุกตัวเลขในอาร์เรย์
        if arr[i] == value:  # ตรวจสอบว่าตัวเลขนั้นตรงกับค่าที่ต้องการหาหรือไม่
            return i  # ถ้าตรงก็คืนค่าดัชนี
    return -1  # ถ้าวนลูปจบแล้วไม่พบ คืนค่า -1

numArray = [7, 16, 2, 0, 5, 1, 30]  # อาร์เรย์ตัวอย่าง
number = 30  # ตัวเลขที่ต้องการหา
result = searchNumber(numArray, number)  # เรียกใช้ฟังก์ชันและเก็บผลลัพธ์
print(result)  # พิมพ์ผลลัพธ์
```

ผลลัพธ์ที่คาดว่าจะได้

- โค้ดนี้จะทำการค้นหาตัวเลข 30 ในอาร์เรย์ numArray.
- ฟังก์ชัน searchNumber จะวนลูปผ่านอาร์เรย์ทั้งหมดเพื่อหาตัวเลข 30 และคืนค่าดัชนีที่พบ (คาดว่าจะได้ดัชนี 6 เพราะ 30 อยู่ที่ตำแหน่งสุดท้ายของอาร์เรย์ที่มีการนับตั้งแต่ 0).
- หากไม่พบตัวเลขในอาร์เรย์ จะคืนค่า -1

4. O(n log n) : Linearithmic Big O ชนิดนี้มีความเร็วระดับกลางๆ



ความซับซ้อน O(n log n) - เชิงเส้นลอการิทึม

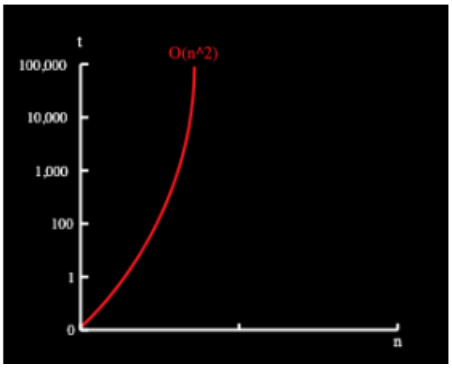
- **ความหมาย:** ความซับซ้อน O(n log n) หมายถึงเวลาที่ใช้ในการดำเนินการเพิ่มขึ้นตามจำนวนข้อมูล (n) และแต่ละการดำเนินการก็จะเพิ่มขึ้นแบบลอการิทึมตามขนาดของข้อมูลนั้นๆ โดยทั่วไปจะเกิดขึ้นในอัลกอริธึมการเรียงลำดับแบบมีการแบ่งข้อมูลเช่น Merge Sort, Heap Sort และ Quick Sort.

```python
for c in range(0, 500, 1):  # วนลูปจาก 0 ถึง 499
    print(0)  # พิมพ์ 0
    i = 1  # ตั้งค่าเริ่มต้นของ i
    while i < c:  # ขณะที่ i น้อยกว่า c
        print(i, 'hello')  # พิมพ์ i และ 'hello'
        i *= 2  # เพิ่มค่า i โดยการคูณด้วย 2
```

ผลลัพธ์ที่คาดว่าจะได้

- คำสั่ง print(0) จะถูกเรียกใช้งานทุกรอบของลูป for ซึ่งทำงานทั้งหมด 500 ครั้ง.
- ในคำสั่ง while, ค่า i เริ่มต้นที่ 1 และคูณสองเป็นเงื่อนไขการหยุดซึ่งเพิ่มขึ้นแบบลอการิทึม (เช่น 1, 2, 4, 8, ...) จนกระทั่ง i มากกว่าหรือเท่ากับ c. การเพิ่มขึ้นของ i ในลักษณะนี้หมายถึงการทำงานที่เพิ่มขึ้นแบบลอการิทึมในแต่ละรอบของลูป for

5. O(n2) : Quadratic Big O ชนิดนี้ใช้เวลาในการทำงานกลางๆ จนเริ่มเข้าขั้นแย่ เพราะว่าเวลาในการทำงานจะเพิ่มขึ้น 4 เท่า เมื่อข้อมูลเพิ่มขึ้น 2 เท่า



ความซับซ้อน O(n^2) - เชิงกำลังสอง

- ความหมาย: ความซับซ้อนของเวลาเป็น O(n^2) หมายความว่าเวลาการดำเนินการของอัลกอริธึมนั้นเพิ่มขึ้นตามกำลังสองของขนาดข้อมูลนำเข้า (n). โดยทั่วไปจะเกิดขึ้นในอัลกอริธึมที่มีการใช้ลูปซ้อนกันสองรอบเพื่อจัดการกับข้อมูล

```
def duplicateCheck(arr):
    for i in range(len(arr)):  # วนลูปสำหรับแต่ละองค์ประกอบในอาร์เรย์
        a = arr[i]  # เก็บค่าที่ตำแหน่ง i
        for j in range(i+1, len(arr)):  # วนลูปเพื่อเปรียบเทียบกับองค์ประกอบที่เหลือ
            b = arr[j]  # เก็บค่าที่ตำแหน่ง j
            print(a, b)  # พิมพ์ค่า a และ b เพื่อแสดงการเปรียบเทียบ
            if a == b:  # ถ้าค่าที่ตำแหน่ง i และ j เท่ากัน
                return "duplicated"  # คืนค่า "duplicated"
    return "not duplicate"  # ถ้าไม่พบค่าซ้ำ, คืนค่า "not duplicate"

numArray = [1, 3, 5, 9]  # อาร์เรย์ตัวอย่าง
number = 30  # ตัวเลขที่ต้องการตรวจสอบ
result = duplicateCheck(numArray)  # เรียกใช้ฟังก์ชันและเก็บผลลัพธ์
print(result)  # พิมพ์ผลลัพธ์
```

ผลลัพธ์ที่คาดว่าจะได้

- ฟังก์ชัน duplicateCheck จะวนลูปผ่านอาร์เรย์ numArray และเปรียบเทียบแต่ละคู่ขององค์ประกอบในอาร์เรย์
- ถ้าพบคู่ขององค์ประกอบที่มีค่าเท่ากัน (เช่น 1==1, 3==3, ...), ฟังก์ชันจะคืนค่า "duplicated"
- ในกรณีตัวอย่างนี้ไม่พบข้อมูลซ้ำในอาร์เรย์ [1, 3, 5, 9], ผลลัพธ์จะเป็น "not duplicate"

6. O(2^n) : Exponential Big O ชนิดนี้เข้าขั้นวิกฤตแล้ว ข้อมูลเพียงนิดเดียว ก็มีการทำงานที่มหาศาล

ความซับซ้อน O(2^n) - เชิงชี้กำลัง

- ความหมาย: ความซับซ้อนของเวลาเป็น O(2^n) หมายความว่าเวลาการดำเนินการของอัลกอริธึมนั้นเพิ่มขึ้นอย่างรวดเร็วตามกำลังสองของขนาดข้อมูลนำเข้า (n). ความซับซ้อนประเภทนี้พบได้ในอัลกอริธึมที่มีการเรียกตัวเองซ้ำหลายครั้งเช่นการคำนวณ Fibonacci แบบเรียกตัวเอง

```python
def fibonacci(n):
    if n == 1:
        return 0  # ฟีโบนัชชีคำนวณถึงตัวที่ 1 คืนค่า 0
    elif n == 2:
        return 1  # ฟีโบนัชชีคำนวณถึงตัวที่ 2 คืนค่า 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)  # คำนวณฟีโบนัชชีโดยการเรียกฟังก์ชันเองสำหรับ n-1 และ n-2

print(fibonacci(5))  # พิมพ์ค่าฟีโบนัชชีตัวที่ 5
print(fibonacci(21))  # พิมพ์ค่าฟีโบนัชชีตัวที่ 21
```

ผลลัพธ์ที่คาดว่าจะได้

- ฟังก์ชัน fibonacci จะคืนค่าผลลัพธ์ของลำดับฟีโบนัชชีตำแหน่งที่ 5 และ 21:
    - Fibonacci(5): การคำนวณจะใช้การเรียกฟังก์ชันซ้ำหลายครั้งจนได้ค่าเป็น 3 (ลำดับฟีโบนัชชีคือ 0, 1, 1, 2, 3)
    - Fibonacci(21): การคำนวณจะใช้การเรียกฟังก์ชันซ้ำจำนวนมากมายและใช้เวลาค่อนข้างนานเนื่องจากความซับซ้อนที่สูง ผลลัพธ์คือ 6765

7. O(n!) : Factorial Big O ชนิดนี้เข้าขั้นเลวร้ายที่สุด

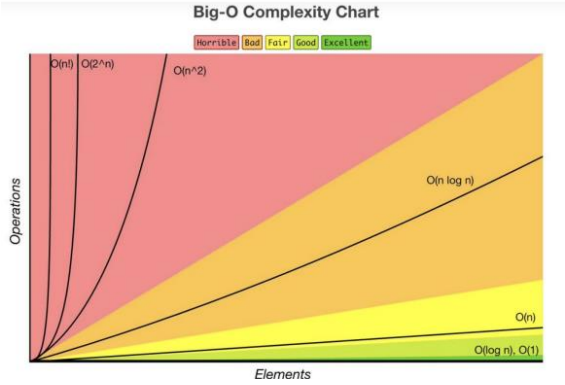
ความซับซ้อน O(n!) - เชิงแฟกทอเรียล

- **ความหมาย:** ความซับซ้อนของเวลาเป็น O(n!) หมายความว่าเวลาการดำเนินการของอัลกอริธึมนั้นเพิ่มขึ้นตามแฟกทอเรียลของขนาดข้อมูลนำเข้า (n). โดยทั่วไปจะเกิดขึ้นในอัลกอริธึมที่มีการเรียกตัวเองหลายครั้งเช่นการคำนวณแฟกทอเรียล

```python
def factorial(num):
    if num == 1:  # ถ้า num คือ 1
        return 1  # คืนค่า 1 เพราะ 1! = 1
    else:
        return num * factorial(num-1)  # คำนวณ num * factorial ของ num-1

print(factorial(9))  # พิมพ์ค่าแฟกทอเรียลของ 9
```

ผลลัพธ์ที่คาดว่าจะได้:

- ฟังก์ชัน factorial จะคืนค่าผลลัพธ์ของการคำนวณแฟกทอเรียลของ 9:
    - **Factorial(9):** จะคำนวณผลลัพธ์ของ 9 * 8 * 7 * ... * 1 ซึ่งเป็น 362880



การจำแนกความซับซ้อนของ Big-O:

1. **O(1) และ O(log n)** - แสดงเป็นสีเขียวอ่อนที่ด้านล่างสุดของแผนภูมิ:
    - **O(1) (Constant):** ความซับซ้อนที่ดีที่สุด การดำเนินการคงที่ไม่ว่าจำนวนข้อมูลจะเปลี่ยนแปลงอย่างไรก็ตาม.
    - **O(log n) (Logarithmic):** ยังมีประสิทธิภาพดี การดำเนินการเพิ่มขึ้นเล็กน้อยเมื่อขนาดข้อมูลเพิ่มขึ้น.
2. **O(n) (Linear):** แสดงเป็นสีเหลือง การดำเนินการเพิ่มขึ้นเชิงเส้นตามขนาดข้อมูลนำเข้า.
3. **O(n log n) (Linearithmic):** แสดงเป็นสีส้ม ความซับซ้อนนี้พบในอัลกอริธึมการเรียงลำดับบางแบบ เช่น quicksort หรือ mergesort มักถือว่ายังคงมีประสิทธิภาพดี.
4. **O(n^2) (Quadratic):** แสดงเป็นสีชมพูอ่อน ความซับซ้อนนี้พบในอัลกอริธึมที่มีลูปซ้อนกันสองรอบ และอาจเป็นปัญหาหากข้อมูลมีขนาดใหญ่.

5. **O(2^n) และ O(n!) (Exponential และ Factorial):** แสดงเป็นสีชมพูเข้ม ความซับซ้อนเหล่านี้มีการเพิ่มขึ้นของการดำเนินการอย่างรวดเร็วมาก และอาจทำให้อัลกอริธึมนั้นใช้ไม่ได้กับข้อมูลขนาดใหญ่

Work

ให้เขียนโปรแกรมขึ้นมา 1 โปรแกรมและบอกว่า โปรแกรมที่เขียนขึ้นนั้น มี Big O เท่ากับเท่าใด •เขียนโปรแกรมการหาค่าผลรวมของเลขคู่และเลขคี่ของตัวเลขที่อยู่ในช่วง 1 ถึง N เมื่อ N คือค่าที่รับจากผู้ใช้ • N=10 • Sum of odd number = 25 • Sum of even number = 30 •เมื่อ N = 500000 , Time = ?? ,Big O = ?

```python
import time  # นำเข้าโมดูล time สำหรับวัดเวลา

def calculate_sums(N):
    sum_odd = 0  # ตั้งตัวแปรเริ่มต้นสำหรับผลรวมตัวเลขคี่
    sum_even = 0  # ตั้งตัวแปรเริ่มต้นสำหรับผลรวมตัวเลขคู่
    for i in range(1, N + 1):  # วนลูปจาก 1 ถึง N
        if i % 2 == 0:  # ตรวจสอบหารด้วย 2 หากเท่ากับศูนย์คือเลขคู่
            sum_even += i  # ถ้าเป็นเลขคู่, ให้เพิ่ม i เข้าไปใน sum_even
        else:
            sum_odd += i  # ถ้าไม่เป็นเลขคู่ (คือเลขคี่), ให้เพิ่ม i เข้าไปใน sum_odd
    return sum_odd, sum_even  # คืนค่าผลรวมของเลขคี่และเลขคู่

N = int(input("Enter N: "))  # รับค่า N จากผู้ใช้
start_time = time.time()  # เก็บเวลาเริ่มต้นการคำนวณ
sum_odd, sum_even = calculate_sums(N)  # คำนวณผลรวมด้วยฟังก์ชันที่กำหนด
end_time = time.time()  # เก็บเวลาสิ้นสุดการคำนวณ

# พิมพ์ผลลัพธ์ออกมา
print(f"Sum of odd numbers: {sum_odd}")
print(f"Sum of even numbers: {sum_even}")
print(f"Time taken: {end_time - start_time:.5f} seconds")  # พิมพ์เวลาที่ใช้ในการคำนวณ
```

```
PS C:\Users\ADMIN\Desktop\week5> pyth
Enter N: 50000
Sum of odd numbers: 625000000
Sum of even numbers: 625025000
Time taken: 0.00399 seconds
PS C:\Users\ADMIN\Desktop\week5>
```