Heap Sort

Group : Basic research

จัดทำโดย

นาย ธนรัตน์ แซ่เฮีย 663380035-4

นาย กฤษณพงษ์ สงครามสงค์ 663380251-8

นาย ธนธร บุญเต็ม 663380267-3

นาย ภควัต ชูโชติรส 663380287-7

นาย ภิณัฐคณิน พิสิษฐกุล 663380528-1

นาย สรชัช พัฒนภิญญ์ 663380535-4

เสนอ

ผศ. ดร.สายยัญ สายยศ

รายงานนี้เป็นส่วนหนึ่งของรายวิชาโครงสร้างข้อมูลและขั้นตอนวิธี sec.2

วิทยาลัยการคอมพิวเตอร์ สาขาวิชาปัญญาประดิษฐ์

ภาคเรียนที่ 1 ปีการศึกษา 2567

บทนำ (Introduction)

การเรียงลำดับข้อมูล (Sorting) เป็นกระบวนการที่สำคัญในวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ เพราะการเรียงลำดับช่วยให้เราสามารถค้นหาและประมวลผลข้อมูลได้ง่ายและมีประสิทธิภาพมากขึ้น นอกจากนี้ยังใช้ในการแก้ปัญหาหลายประเภทในชีวิตประจำวัน เช่น การค้นหาข้อมูลในฐานข้อมูล การจัดลำดับความสำคัญของงานในระบบจัดการทรัพยากร และการวิเคราะห์ข้อมูลจำนวนมาก อัลกอริธึมการเรียงลำดับ (Sorting Algorithms) เป็นหัวข้อที่มีความหลากหลาย และมีวิธีการที่แตกต่างกันในการจัดเรียงข้อมูล อัลกอริธึมการเรียงลำดับที่มีชื่อเสียงได้แก่ Bubble Sort, Merge Sort, Quick Sort และ Heap Sort แต่ละอัลกอริธึมมีข้อดีและข้อเสียที่ต่างกันไป ขึ้นอยู่กับประเภทของข้อมูลและความซับซ้อนของปัญหา

Heap Sort เป็นหนึ่งในอัลกอริธึมการเรียงลำดับที่มีประสิทธิภาพสูง ซึ่งถูกพัฒนาขึ้นโดยอาศัยโครงสร้างข้อมูลแบบ Heap อัลกอริธึมนี้มีจุดเด่นที่สามารถจัดการกับข้อมูลขนาดใหญ่ได้ดี และมีเวลาในการทำงานแบบ O(n log n) ในทุกกรณี ทั้งในกรณีที่ดีที่สุด (best case) และกรณีที่แย่ที่สุด (worst case) ซึ่งเป็นข้อดีที่ทำให้ Heap Sort แตกต่างจากอัลกอริธึมอื่นๆ เช่น Quick Sort ที่เวลาการทำงานอาจมีการเปลี่ยนแปลงตามลักษณะของข้อมูล Heap Sort ใช้โครงสร้างข้อมูลแบบ binary heap ในการจัดการข้อมูล ซึ่งช่วยให้สามารถคำนวณและจัดเรียงข้อมูลได้อย่างรวดเร็วและมีประสิทธิภาพ

ความสำคัญของการเรียนรู้ Heap Sort ไม่เพียงแต่อยู่ที่การทำความเข้าใจวิธีการจัดเรียงข้อมูล แต่ยังช่วยเสริมความเข้าใจในโครงสร้างข้อมูลแบบ Heap ซึ่งมีประโยชน์อย่างมากในงานประมวลผลที่ต้องจัดลำดับความสำคัญของข้อมูล Heap Sort มีการประยุกต์ใช้งานที่หลากหลาย ตั้งแต่การสร้างระบบการจัดลำดับความสำคัญ (priority queue) การจัดการทรัพยากรในระบบปฏิบัติการ การค้นหาข้อมูลที่มีขนาดใหญ่ (external sorting) ไปจนถึงงานที่ต้องการความรวดเร็วในการประมวลผลข้อมูลจำนวนมาก

รายงานนี้มีวัตถุประสงค์เพื่อศึกษากระบวนการทำงานและการใช้งานของ Heap Sort โดยจะอธิบายรายละเอียดของขั้นตอนการทำงานและหลักการที่เกี่ยวข้อง รวมถึงการวิเคราะห์ข้อดีข้อเสียของอัลกอริธึม เพื่อให้ผู้อ่านสามารถเข้าใจการทำงานและสามารถนำไปประยุกต์ใช้งานได้ในสถานการณ์ต่างๆ

แนวคิดและพื้นฐานของ Heap (Concept and Basics of Heap)

Heap เป็นโครงสร้างข้อมูลที่มีลักษณะเป็นต้นไม้ (tree) ที่จัดเรียงข้อมูลตามคุณสมบัติที่เฉพาะเจาะจง Heap ถูกนำมาใช้ในหลายอัลกอริธึมโดยเฉพาะ Heap Sort ซึ่ง Heap สามารถแบ่งออกเป็นสองประเภทหลักๆ คือ Max Heap และ Min Heap โดยมีคุณสมบัติที่ทำให้ Heap เหมาะสมกับการใช้งานในการจัดเรียงข้อมูลหรือการจัดลำดับความสำคัญของงานในระบบคอมพิวเตอร์

ใน Max Heap โหนดแม่จะต้องมีค่ามากกว่าหรือเท่ากับลูกทั้งสองของมันเสมอ ในทางกลับกัน Min Heap จะมีโหนดแม่ที่มีค่าน้อยกว่าหรือเท่ากับลูกทั้งสองของมัน ในกรณีของ Heap Sort นั้นจะใช้ Max Heap เป็นหลัก เนื่องจากต้องการนำค่าที่มากที่สุดออกไปเรื่อยๆ เพื่อนำมาเรียงลำดับในอาเรย์ Heap ถูกจัดเก็บในรูปแบบอาเรย์ (array) และมีวิธีการคำนวณตำแหน่งของลูกซ้ายและลูกขวาจากตำแหน่งของโหนดแม่ได้อย่างมีประสิทธิภาพ

คุณสมบัติสำคัญของ Heap ที่เรียกว่า Heap Property นั้น ระบุว่าตำแหน่งของโหนดแม่ต้องมากกว่าหรือน้อยกว่าโหนดลูกทั้งสอง ขึ้นอยู่กับว่าเป็น Max Heap หรือ Min Heap ตัวอย่างเช่น ใน Max Heap โหนดแม่จะต้องมีค่ามากกว่าหรือเท่ากับลูกเสมอ ซึ่งหมายความว่าโหนดที่มีค่ามากที่สุดจะอยู่ที่ราก (root) ของ Heap เสมอ ในทางปฏิบัติ โหนดลูกใน Heap จะอยู่ในตำแหน่งที่สามารถเข้าถึงได้โดยตรงจากโหนดแม่ที่อยู่ในอาเรย์ โดยสามารถคำนวณตำแหน่งของลูกซ้ายได้จากสูตร 2*i + 1 และลูกขวาจากสูตร 2*i + 2 โดยที่ i คือ index ของโหนดแม่ในอาเรย์

Heap ถูกใช้อย่างกว้างขวางในการจัดการข้อมูลที่ต้องการการเข้าถึงหรือดึงค่าที่มากที่สุดหรือน้อยที่สุดอย่างรวดเร็ว เช่น การใช้งานใน priority queue ซึ่งจะช่วยจัดลำดับความสำคัญของงานในระบบคอมพิวเตอร์ นอกจากนี้ Heap ยังสามารถใช้ในการจัดการทรัพยากรในระบบปฏิบัติการ การจัดลำดับเหตุการณ์ในระบบเกม หรือการจัดการกับข้อมูลขนาดใหญ่ที่ไม่สามารถโหลดเข้าสู่หน่วยความจำได้พร้อมกัน (external sorting)

ในกรณีของ Max Heap ซึ่งใช้ใน Heap Sort แนวคิดหลักคือการดึงโหนดที่มีค่ามากที่สุดออกจากโครงสร้าง Heap โดยสลับตำแหน่งของรากกับโหนดสุดท้ายในอาเรย์ จากนั้นลดขนาดของ Heap และเรียกใช้ฟังก์ชันที่เรียกว่า heapify เพื่อจัดโครงสร้าง Heap ใหม่ กระบวนการนี้จะถูกทำซ้ำไปเรื่อยๆ จนกระทั่ง Heap ถูกลดขนาดลงจนหมด ทำให้ได้ข้อมูลที่เรียงลำดับอย่างสมบูรณ์ Heap จึงมีความสำคัญอย่างมากในอัลกอริธึม Heap Sort ทั้งในแง่ของการจัดการข้อมูลและการประมวลผล

ขั้นตอนการทำงานของ Heap Sort (Working of Heap Sort)

Heap Sort เป็นอัลกอริธึมการเรียงลำดับที่มีประสิทธิภาพ โดยใช้โครงสร้างข้อมูลแบบ Heap ซึ่งประกอบไปด้วยสองขั้นตอนหลักคือ การสร้าง Max Heap จากข้อมูลต้นฉบับ และการสลับค่าที่มากที่สุดไปยังตำแหน่งท้ายของอาเรย์เพื่อจัดเรียงข้อมูล ในขั้นตอนเหล่านี้ Heap Sort จะทำงานผ่านการปรับโครงสร้าง Heap ด้วยฟังก์ชันที่เรียกว่า heapify ที่ทำหน้าที่จัดเรียงโหนดต่างๆ ให้เป็น Max Heap เสมอ

1. การสร้าง Max Heap จากอาเรย์

ขั้นตอนแรกของ Heap Sort คือการเปลี่ยนอาเรย์ที่ยังไม่ได้เรียงลำดับให้กลายเป็น Max Heap ซึ่งใน Max Heap โหนดแม่จะต้องมีค่ามากกว่าหรือเท่ากับลูกทั้งสองฝั่งเสมอ กระบวนการนี้เริ่มต้นจากการเลือกโหนดที่ไม่ใช่ใบ (non-leaf node) ที่อยู่ในตำแหน่งล่างสุดของต้นไม้ โดยจะเริ่มจากโหนดที่อยู่ล่างสุดและทำการเรียกฟังก์ชัน heapify สำหรับแต่ละโหนดเพื่อให้แต่ละ subtree เป็น Max Heap

การสร้าง Max Heap จะทำให้ข้อมูลทั้งหมดในอาเรย์ถูกจัดเรียงในลักษณะที่โหนด root (index 0) มีค่ามากที่สุด เมื่อสร้าง Max Heap เสร็จสิ้น ค่าใหญ่สุดจะถูกย้ายไปที่ตำแหน่งรากของต้นไม้ นี่คือพื้นฐานในการดำเนินการเรียงลำดับของ Heap Sort

2. การสลับค่าโหนด root ไปยังตำแหน่งท้ายของอาเรย์

หลังจากที่สร้าง Max Heap สำเร็จแล้ว โหนด root ซึ่งมีค่ามากที่สุดใน Heap จะถูกสลับกับโหนดที่อยู่ท้ายสุดของอาเรย์ การสลับนี้ทำให้ค่าที่ใหญ่ที่สุดถูกย้ายออกจาก Heap และถูกนำไปเก็บในตำแหน่งที่ถูกต้องในอาเรย์ที่ถูกเรียงลำดับ

หลังจากสลับค่าระหว่างโหนด root และโหนดสุดท้ายเสร็จแล้ว ขนาดของ Heap จะถูกลดลงไปทีละ 1 เนื่องจากค่าที่ถูกสลับไปนั้นเป็นค่าที่ถูกเรียงลำดับแล้ว และไม่ต้องนำมาเข้ากระบวนการอีก ใน Heap ที่เหลือเราจะมีข้อมูลที่ยังไม่ได้เรียงลำดับอยู่ และต้องการให้ค่าใหญ่ที่สุดใน Heap ปรากฏที่โหนด root อีกครั้ง

3. การเรียกใช้ heapify เพื่อจัดโครงสร้าง heap ใหม่

เมื่อมีการสลับโหนด root กับโหนดสุดท้ายแล้ว โครงสร้างของ Heap อาจเสียสมดุลไป ดังนั้นเราจึงต้องเรียกใช้ฟังก์ชัน heapify เพื่อจัดโครงสร้าง Heap ใหม่ให้เป็น Max Heap กระบวนการนี้จะเริ่มต้นจากโหนด root ที่ถูกสลับไป และตรวจสอบโหนดลูกซ้ายและลูกขวา หากโหนดลูกใดมีค่ามากกว่าโหนดแม่ จะทำการสลับค่ากัน และเรียกใช้ heapify ซ้ำไปเรื่อยๆ จนกว่า Heap จะกลับมาเป็น Max Heap อีกครั้ง

กระบวนการนี้จะถูกทำซ้ำไปเรื่อยๆ จนกระทั่งข้อมูลทั้งหมดในอาเรย์ถูกเรียงลำดับอย่างสมบูรณ์ การทำงานของ Heap Sort นั้นสามารถสรุปได้ง่ายๆ ว่าเป็นการนำค่าใหญ่ที่สุดออกจาก Heap แล้วจัดเรียงให้ถูก

ตำแหน่งทีละตัว ผ่านการสร้าง Max Heap และการสลับค่า โดยทุกครั้งที่นำค่าออกจาก Heap เราจะปรับโครงสร้าง Heap ใหม่ด้วย heapify เพื่อให้มั่นใจว่าโครงสร้างของ Heap ยังถูกต้อง

โค้ดและการอธิบายโค้ด (Code and Code Explanation)

ในส่วนนี้เราจะทำความเข้าใจอัลกอริธึม Heap Sort ผ่านตัวอย่างโค้ดในภาษา Python โดยมีการอธิบายการทำงานของแต่ละบรรทัดในโค้ดอย่างละเอียด ซึ่งครอบคลุมทั้งฟังก์ชัน heapify ที่ใช้ในการจัดการโครงสร้าง heap และฟังก์ชัน heap_sort ที่ทำหน้าที่จัดเรียงข้อมูล

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

    print(f"i-> {i}  large_idx-> {largest}  arr-> {arr}")

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    print('-----------------------------------------------')
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

อธิบายโค้ด

1. ฟังก์ชัน heapify ฟังก์ชันนี้เป็นหัวใจสำคัญในการจัดการโครงสร้างของ Heap โดยจะตรวจสอบว่าค่าในโหนดแม่ (parent node) มีค่ามากกว่าหรือน้อยกว่าโหนดลูก (child nodes) หรือไม่

1.1 บรรทัดแรก largest = i กำหนดให้โหนดแม่ (index i) เป็นค่าที่มากที่สุดเบื้องต้น

1.2 left = 2 * i + 1 และ right = 2 * i + 2 ใช้คำนวณ index ของโหนดลูกซ้ายและลูกขวา ซึ่งเป็นคุณสมบัติของ heap ที่โหนดลูกสามารถหาจากตำแหน่งนี้ในโครงสร้างต้นไม้แบบ binary heap

1.3 เงื่อนไข if left < n and arr[left] > arr[largest] ใช้ตรวจสอบว่าโหนดลูกซ้ายมีค่ามากกว่าโหนดแม่หรือไม่ หากมีให้กำหนดค่า largest เป็นโหนดลูกซ้าย

1.4 ในทำนองเดียวกัน เงื่อนไข if right < n and arr[right] > arr[largest] ตรวจสอบโหนดลูกขวาและกำหนดให้ largest เป็นโหนดลูกขวาหากมีค่ามากกว่าโหนดแม่

1.5 หากพบว่าโหนดลูกมีค่ามากกว่าโหนดแม่ ฟังก์ชันจะสลับค่าระหว่างโหนดแม่กับโหนดลูกที่มีค่ามากที่สุด (arr[i], arr[largest] = arr[largest], arr[i])

1.6 จากนั้นเรียกฟังก์ชัน heapify ซ้ำเพื่อจัดการกับโครงสร้างของ subtree นั้นๆ

2. ฟังก์ชัน heap_sort ฟังก์ชันนี้ทำหน้าที่เรียงลำดับข้อมูลโดยเริ่มจากการสร้าง Max Heap และการสลับโหนด

2.1 บรรทัด n = len(arr) เก็บขนาดของอาเรย์ที่ต้องการเรียงลำดับ

2.2 วนซ้ำ for แรก (for i in range(n // 2 - 1, -1, -1):) เริ่มต้นจากโหนดที่ไม่ใช่ใบ (non-leaf node) แล้วทำการ heapify เพื่อสร้าง Max Heap จากอาเรย์ต้นฉบับ

2.3 หลังจาก Max Heap ถูกสร้างเสร็จ วง for ที่สอง (for i in range(n - 1, 0, -1):) ทำหน้าที่สลับค่าโหนด root (ค่ามากที่สุด) กับโหนดสุดท้ายใน heap ที่ยังไม่ถูกจัดเรียง แล้วเรียก heapify กับ heap ที่เหลือเพื่อลดขนาดลงทีละ 1 และจัดโครงสร้าง heap ใหม่

3. การคำนวณ Index การคำนวณตำแหน่งของลูกซ้ายและลูกขวาใน Heap ทำได้ง่ายๆ ด้วยสูตร left = 2*i + 1 และ right = 2*i + 2 โดยที่ i คือ index ของโหนดแม่ สูตรนี้จะช่วยให้สามารถเดินทางจากโหนดแม่ไปยังโหนดลูกในโครงสร้าง heap ที่ถูกจัดเก็บเป็นอาเรย์ได้อย่างง่ายดาย

4. การสร้าง Max Heap การสร้าง Max Heap เป็นกระบวนการที่เริ่มต้นจากโหนดที่ไม่ใช่ใบในอาเรย์ต้นฉบับ โดยทำการ heapify ตั้งแต่โหนดล่างสุดขึ้นมาจนถึงโหนด root เพื่อให้แน่ใจว่าโครงสร้าง heap นั้นเป็น Max Heap ที่โหนดแม่มีค่ามากกว่าหรือเท่ากับโหนดลูกเสมอ

ตัวอย่างการทำงาน (Examples of Heap Sort)

เพื่อให้เข้าใจถึงกระบวนการทำงานของ Heap Sort ได้อย่างชัดเจน เราจะยกตัวอย่างอาเรย์ตัวเลข [4, 3, 2, 1, 6, 5] แล้วอธิบายทีละขั้นตอน จนกระทั่งอาเรย์ถูกเรียงลำดับเรียบร้อย การทำงานของ Heap Sort เริ่มต้นจากการสร้าง Max Heap จากอาเรย์ต้นฉบับ จากนั้นจึงนำค่าใหญ่ที่สุด (ซึ่งจะอยู่ที่โหนด root) ไปสลับกับค่าในตำแหน่งท้ายสุดของ heap และลดขนาดของ heap ลง กระบวนการนี้จะทำซ้ำจนกระทั่ง heap ถูกจัดเรียงสมบูรณ์

ขั้นตอนที่ 1 การสร้าง Max Heap เริ่มต้นจากอาเรย์ต้นฉบับ [4, 3, 2, 1, 6, 5] เราจะทำการสร้าง Max Heap เพื่อให้โหนดที่มีค่ามากที่สุดอยู่ที่ตำแหน่ง root โดยเราจะเริ่ม heapify จากโหนดที่ไม่ใช่ใบ (non-leaf node) ดังนี้

1.1 โหนดที่ไม่ใช่ใบเริ่มจาก index n // 2 - 1 = 2 ซึ่งค่าของโหนดคือ 2 เราจะตรวจสอบโหนดลูก (5) และสลับค่าเนื่องจาก 5 > 2 ทำให้ได้อาเรย์ใหม่เป็น [4, 3, 5, 1, 6, 2]

1.2 ต่อมา heapify ที่โหนด index 1 (3) พบว่าลูกขวาคือ 6 มีค่ามากกว่า จึงสลับ 3 กับ 6 ทำให้อาเรย์เป็น [4, 6, 5, 1, 3, 2]

1.3 สุดท้าย heapify โหนด root (4) พบว่าค่าลูกซ้ายคือ 6 มากกว่า จึงสลับ 4 กับ 6 ทำให้ได้ Max Heap ที่สมบูรณ์คือ [6, 4, 5, 1, 3, 2]

ขั้นตอนที่ 2 สลับค่า root กับค่าท้ายอาเรย์ เมื่อสร้าง Max Heap เสร็จเรียบร้อยแล้ว เราจะสลับค่า root ซึ่งเป็นค่าที่มากที่สุด (6) กับค่าในตำแหน่งสุดท้าย (2) ทำให้อาเรย์กลายเป็น [2, 4, 5, 1, 3, 6] และลดขนาด heap ลงทีละ 1

ขั้นตอนที่ 3 heapify กับ heap ที่เหลือ หลังจากสลับค่าของ root กับตำแหน่งสุดท้ายแล้ว เราจะทำการ heapify เพื่อจัดโครงสร้าง heap ใหม่สำหรับ heap ที่เหลือ (ขนาดลดลง)

3.1 Heapify ที่ root (2) พบว่าค่าลูกขวา (5) มากที่สุด จึงสลับค่า 2 กับ 5 ทำให้อาเรย์กลายเป็น [5, 4, 2, 1, 3, 6]

3.2 สลับค่า root (5) กับตำแหน่งท้ายสุดของ heap (3) ทำให้อาเรย์กลายเป็น [3, 4, 2, 1, 5, 6]

3.3 Heapify ที่ root (3) พบว่าลูกซ้าย (4) มากที่สุด จึงสลับค่า ทำให้อาเรย์เป็น [4, 3, 2, 1, 5, 6]

ขั้นตอนที่ 4 ทำซ้ำขั้นตอน ทำขั้นตอนการสลับและ heapify ซ้ำสำหรับ heap ที่เหลือ

4.1 สลับค่า root (4) กับตำแหน่งท้ายสุด (1) ทำให้อาเรย์เป็น [1, 3, 2, 4, 5, 6]

4.2 Heapify ที่ root (1) พบว่าลูกซ้าย (3) มากกว่า จึงสลับ ทำให้อาเรย์เป็น [3, 1, 2, 4, 5, 6]

4.3 สุดท้ายสลับ 3 กับ 2 ทำให้ได้อาเรย์ [1, 2, 3, 4, 5, 6]

```
arr = [4, 3, 2, 1, 6, 5]
print("Original array:", arr, "\n")
heap_sort(arr)
print("\nSorted array:", arr)
```

[36] ✓ 0.0s

```
Original array: [4, 3, 2, 1, 6, 5]

i-> 2  large_idx-> 5  arr-> [4, 3, 2, 1, 6, 5]
i-> 5  large_idx-> 5  arr-> [4, 3, 5, 1, 6, 2]
i-> 1  large_idx-> 4  arr-> [4, 3, 5, 1, 6, 2]
i-> 4  large_idx-> 4  arr-> [4, 6, 5, 1, 3, 2]
i-> 0  large_idx-> 1  arr-> [4, 6, 5, 1, 3, 2]
i-> 1  large_idx-> 1  arr-> [6, 4, 5, 1, 3, 2]
--------------------------------------------
i-> 0  large_idx-> 2  arr-> [2, 4, 5, 1, 3, 6]
i-> 2  large_idx-> 2  arr-> [5, 4, 2, 1, 3, 6]
i-> 0  large_idx-> 1  arr-> [3, 4, 2, 1, 5, 6]
i-> 1  large_idx-> 1  arr-> [4, 3, 2, 1, 5, 6]
i-> 0  large_idx-> 1  arr-> [1, 3, 2, 4, 5, 6]
i-> 1  large_idx-> 1  arr-> [3, 1, 2, 4, 5, 6]
i-> 0  large_idx-> 0  arr-> [2, 1, 3, 4, 5, 6]
i-> 0  large_idx-> 0  arr-> [1, 2, 3, 4, 5, 6]

Sorted array: [1, 2, 3, 4, 5, 6]
```

ผลลัพธ์ที่ได้ หลังจากทำการสลับและ heapify ครบทุกขั้นตอนแล้ว อาเรย์จะถูกเรียงลำดับสมบูรณ์เป็น [1, 2, 3, 4, 5, 6] แสดงถึงการทำงานของ Heap Sort ที่ดึงค่าใหญ่ที่สุดจาก Max Heap ไปยังตำแหน่งท้ายอาเรย์เรื่อยๆ จนกระทั่งข้อมูลทั้งหมดถูกจัดเรียงเรียบร้อย Heap Sort มีประสิทธิภาพสูงในกรณีที่ข้อมูลมีขนาดใหญ่ เนื่องจากกระบวนการ heapify ใช้เวลา O(log n) และการสร้าง heap ใช้เวลา O(n log n) ซึ่งทำให้ Heap Sort มีประสิทธิภาพที่มั่นคง

การวิเคราะห์ประสิทธิภาพ (Performance Analysis)

Heap Sort เป็นอัลกอริธึมการเรียงลำดับที่มีประสิทธิภาพสูง ซึ่งสามารถทำงานได้อย่างมีเสถียรภาพในทุกกรณี เนื่องจากใช้โครงสร้างข้อมูลแบบ Heap ในการจัดการค่าของข้อมูล โดยเฉพาะใน Max Heap ซึ่งทำให้ค่าใหญ่ที่สุดถูกดึงออกไปทีละตัวและจัดเรียงอย่างเหมาะสม กระบวนการทำงานของ Heap Sort ในแต่ละขั้นตอนสามารถวิเคราะห์ได้จากเวลาในการประมวลผลในกรณีต่างๆ ดังนี้

เวลาเฉลี่ย (Average case) O(n log n) ในกรณีทั่วไป Heap Sort จะทำงานโดยการสร้าง Max Heap จากข้อมูลขนาด n ซึ่งใช้เวลาในการสร้าง heap ที่ O(n) หลังจากนั้นจะมีการสลับค่าระหว่างโหนด root และโหนดสุดท้ายของ heap ซึ่งต้องใช้เวลาสำหรับการ heapify โครงสร้างใหม่ โดยเวลาที่ใช้ในการ heapify แต่ละครั้งจะอยู่ที่ O(log n) และกระบวนการนี้จะเกิดขึ้น n ครั้ง ดังนั้น เวลาเฉลี่ยสำหรับการทำงานของ Heap Sort อยู่ที่ O(n log n) ในกรณีทั่วไป

กรณีที่ดีที่สุด (Best case) O(n log n) Heap Sort ไม่มีความแตกต่างในเวลาในการทำงานระหว่างกรณีที่ดีที่สุดและกรณีที่แย่ที่สุด เนื่องจากการทำงานของอัลกอริธึมจะต้องสร้าง Max Heap เสมอ ไม่ว่าข้อมูลที่ป้อนเข้ามาจะเรียงลำดับอยู่แล้วหรือไม่ จึงต้องใช้เวลา O(n) ในการสร้าง heap และ O(log n) ในการ heapify ทุกครั้ง ส่งผลให้เวลาในกรณีที่ดีที่สุดเป็น O(n log n) เช่นเดียวกับกรณีทั่วไป

กรณีที่แย่ที่สุด (Worst case) O(n log n) แม้ในกรณีที่ข้อมูลป้อนเข้ามาเป็นลำดับที่แย่ที่สุด (เช่น ข้อมูลเรียงลำดับจากน้อยไปมาก แต่ต้องการเรียงจากมากไปน้อย) Heap Sort ก็ยังคงใช้เวลา O(n log n) เนื่องจากทุกขั้นตอนยังคงต้องสร้าง Max Heap และทำการ heapify เพื่อรักษาโครงสร้างของ heap ให้อยู่ในรูปแบบที่ถูกต้อง กระบวนการนี้จึงทำให้ Heap Sort มีประสิทธิภาพคงที่ในทุกกรณี

การเปรียบเทียบกับอัลกอริธึมการเรียงลำดับอื่นๆ เมื่อเปรียบเทียบกับอัลกอริธึมอื่น เช่น Quick Sort และ Merge Sort จะพบว่า Heap Sort มีข้อดีที่ความเร็วคงที่ในทุกกรณี โดยที่ Quick Sort มีเวลาเฉลี่ย O(n log n) เช่นกัน แต่ในกรณีที่แย่ที่สุด Quick Sort อาจใช้เวลาถึง $O(n^2)$ หากมีการเลือก pivot ที่ไม่ดี ส่วน Merge Sort มีเวลา O(n log n) ในทุกกรณี แต่ต้องใช้หน่วยความจำเพิ่มเติม (O(n)) ในการจัดการกับข้อมูลชั่วคราว ทำให้ Heap Sort มีข้อได้เปรียบในด้านการใช้งานหน่วยความจำที่น้อยกว่า

การใช้หน่วยความจำ (Space Complexity) O(1) Heap Sort มีข้อได้เปรียบสำคัญในแง่ของการใช้หน่วยความจำ เนื่องจาก Heap Sort จัดการเรียงลำดับข้อมูลในที่เดียวกัน (in-place) โดยไม่ต้องใช้อาเรย์เพิ่มเติมในการเก็บข้อมูล ทำให้มีการใช้พื้นที่หน่วยความจำแบบคงที่หรือ O(1) ต่างจาก Merge Sort ที่ต้องการพื้นที่เพิ่มสำหรับการแบ่งและรวมอาเรย์ชั่วคราว ในขณะที่ Quick Sort อาจมีการใช้พื้นที่หน่วยความจำเพิ่มขึ้นในกรณีที่แย่ที่สุดเมื่อมีการแบ่งข้อมูลมากเกินไป

สรุป Heap Sort มีเวลาในการทำงานที่คงที่ในทุกกรณี โดยเวลาเฉลี่ยและเวลาที่แย่ที่สุดอยู่ที่ O(n log n) ซึ่งเป็นข้อได้เปรียบเมื่อเทียบกับ Quick Sort ที่อาจมีเวลา $O(n^2)$ ในบางกรณี นอกจากนี้ Heap Sort ยังใช้พื้นที่น้อยมากเพียง O(1) ทำให้เหมาะสมสำหรับการประยุกต์ใช้กับข้อมูลขนาดใหญ่ที่ต้องการการประมวลผลอย่างมีประสิทธิภาพและหน่วยความจำที่จำกัด

ข้อดี ข้อเสีย และการประยุกต์ใช้งาน (Advantages, Disadvantages, and Applications)

1.ข้อดี (Advantages)

1.1 มีประสิทธิภาพในการจัดเรียงข้อมูล Heap Sort เป็นอัลกอริธึมการเรียงลำดับที่มีประสิทธิภาพสูง ซึ่งสามารถจัดการกับข้อมูลที่มีขนาดใหญ่ได้อย่างมีประสิทธิภาพ เนื่องจาก Heap Sort ใช้โครงสร้างข้อมูลแบบ heap ในการจัดการกับข้อมูล และสามารถทำงานได้ในเวลา O(n log n) ในทุกกรณี ทำให้ Heap Sort เหมาะสมกับการประยุกต์ใช้งานที่ต้องการความเสถียรของเวลาในการประมวลผล ไม่ว่าจะเป็นกรณีข้อมูลที่เรียงอยู่แล้วหรือข้อมูลที่ไม่ได้เรียงก็ตาม

1.2 ไม่ต้องการพื้นที่เพิ่มเติมในการเรียงลำดับ (In-place sorting) Heap Sort เป็นอัลกอริธึมแบบ in-place หมายความว่าไม่จำเป็นต้องใช้อาเรย์หรือหน่วยความจำเพิ่มเติมในการจัดเรียงข้อมูล ข้อมูลทั้งหมดจะถูกจัดการและจัดเรียงภายในพื้นที่ที่มีอยู่แล้ว ดังนั้น Heap Sort จึงใช้หน่วยความจำอย่างมีประสิทธิภาพ โดยมีความซับซ้อนในการใช้หน่วยความจำเท่ากับ O(1) ซึ่งต่างจากอัลกอริธึมอย่าง Merge Sort ที่ต้องใช้อาเรย์เพิ่มเติม

1.3 เวลาในการทำงานคงที่ในทุกกรณี (O(n log n)) Heap Sort มีเวลาในการทำงานคงที่ในกรณีต่างๆ ทั้งกรณีที่ดีที่สุด กรณีเฉลี่ย และกรณีที่แย่ที่สุด ซึ่งอยู่ที่ O(n log n) สิ่งนี้ทำให้ Heap Sort เหมาะสมกับการประยุกต์ใช้งานที่ต้องการความเสถียรและความรวดเร็วในการจัดเรียงข้อมูลเมื่อเปรียบเทียบกับอัลกอริธึมอย่าง Quick Sort ที่อาจใช้เวลา $O(n^2)$ ในกรณีที่แย่ที่สุด

2.ข้อเสีย (Disadvantages)

2.1 ซับซ้อนกว่าอัลกอริธึมการเรียงลำดับอื่นในแง่ของการทำความเข้าใจและการนำไปใช้งาน แม้ว่า Heap Sort จะมีประสิทธิภาพสูง แต่กระบวนการทำงานของมันซับซ้อนกว่าอัลกอริธึมอื่นๆ เช่น Insertion Sort หรือ Bubble Sort เนื่องจากต้องทำความเข้าใจการจัดการโครงสร้าง heap และการทำงานของฟังก์ชัน

heapify ซึ่งมีการทำงานแบบ recursive สำหรับผู้เริ่มต้นหรือในสถานการณ์ที่ข้อมูลมีขนาดเล็ก อัลกอริธึมที่ง่ายกว่าอาจเป็นทางเลือกที่ดีกว่า

3.การประยุกต์ใช้งาน (Applications)

3.1 ใช้ในการสร้าง priority queues Heap เป็นโครงสร้างข้อมูลที่นิยมใช้ในการสร้าง priority queue ซึ่งใช้ในระบบที่ต้องจัดลำดับความสำคัญของข้อมูล เช่น ในระบบคอมพิวเตอร์ Heap Sort สามารถนำมาใช้ในการจัดการกับ priority queue โดยใช้ heap ในการเพิ่มและลบข้อมูลตามลำดับความสำคัญอย่างมีประสิทธิภาพ

3.2 ใช้ในระบบที่ต้องการการจัดการทรัพยากรที่มีลำดับความสำคัญ (เช่น CPU scheduling) ในระบบคอมพิวเตอร์ที่ต้องจัดการกับการประมวลผลหลายงาน เช่น CPU scheduling ระบบจำเป็นต้องกำหนดลำดับความสำคัญของงาน Heap Sort สามารถนำมาใช้ในการสร้างและจัดการกับลำดับความสำคัญของงานเหล่านี้ได้อย่างมีประสิทธิภาพ เช่น การจัดสรรทรัพยากร การจัดการคิวงานที่มีลำดับความสำคัญสูงหรือต่ำกว่า

3.3 ใช้ในระบบที่มีข้อมูลขนาดใหญ่และไม่สามารถจัดเรียงในหน่วยความจำทั้งหมดได้ (external sorting) ในกรณีที่ต้องจัดการกับข้อมูลขนาดใหญ่ที่ไม่สามารถโหลดเข้าสู่หน่วยความจำทั้งหมดในคราวเดียวได้ เช่น ในการจัดเรียงข้อมูลในระบบฐานข้อมูลขนาดใหญ่ Heap Sort เหมาะสมกับการใช้งานในลักษณะนี้ เนื่องจากไม่ต้องการหน่วยความจำเพิ่มเติม ทำให้สามารถจัดเรียงข้อมูลได้อย่างมีประสิทธิภาพและรวดเร็วแม้ในระบบที่มีข้อจำกัดด้านทรัพยากร

4.สรุป Heap Sort เป็นอัลกอริธึมการเรียงลำดับที่มีประสิทธิภาพสูง ไม่เพียงแต่ทำงานได้ดีในทุกกรณีและไม่ต้องการหน่วยความจำเพิ่มเติม แต่ยังมีการประยุกต์ใช้งานในหลายสถานการณ์ โดยเฉพาะอย่างยิ่งในการจัดการลำดับความสำคัญและการประมวลผลข้อมูลขนาดใหญ่ อย่างไรก็ตาม อัลกอริธึมนี้อาจซับซ้อนกว่าอัลกอริธึมการเรียงลำดับแบบพื้นฐาน

สรุป

Heap Sort เป็นอัลกอริธึมการเรียงลำดับที่สำคัญและมีประสิทธิภาพสูง ซึ่งใช้โครงสร้างข้อมูลแบบ Heap ในการจัดเรียงข้อมูล Heap Sort มีคุณสมบัติที่โดดเด่นในด้านความเสถียรในการทำงาน เนื่องจากเวลาในการทำงานคงที่ในทุกกรณีที่ O(n log n) ไม่ว่าจะเป็นกรณีที่ดีที่สุด (Best Case), กรณีที่แย่ที่สุด (Worst Case) หรือกรณีทั่วไป (Average Case) นอกจากนี้ Heap Sort ยังเป็นอัลกอริธึมการจัดเรียงข้อมูลแบบ in-place หมายความว่าไม่ต้องการหน่วยความจำเพิ่มเติมในการจัดเรียง ทำให้เหมาะสมกับการใช้งานในกรณีที่ต้องจัดการข้อมูลขนาดใหญ่

จากการศึกษา Heap Sort เราได้เรียนรู้ถึงกระบวนการทำงานหลักสองส่วน ได้แก่ การสร้าง Max Heap และการสลับโหนด root ที่มีค่ามากที่สุดกับตำแหน่งท้ายสุดของอาเรย์ กระบวนการนี้ทำให้ข้อมูลในอาเรย์ถูกจัดเรียงอย่างมีประสิทธิภาพ Heap Sort ใช้การเรียกฟังก์ชัน `heapify` เพื่อปรับโครงสร้าง heap ใหม่ทุกครั้งหลังจากที่มีการสลับค่า ซึ่งเป็นกลไกสำคัญในการรักษาคุณสมบัติของ heap ให้คงอยู่

จุดแข็งของ Heap Sort ได้แก่ ความเสถียรในการทำงานที่ O(n log n) ในทุกกรณี ความสามารถในการจัดการข้อมูลขนาดใหญ่ได้โดยไม่ต้องการหน่วยความจำเพิ่มเติม และเป็นอัลกอริธึมที่เหมาะสมกับการประยุกต์ใช้งานในระบบที่ต้องการการจัดการทรัพยากรอย่างมีลำดับความสำคัญ เช่น priority queues หรือ CPU scheduling ในระบบปฏิบัติการ

อย่างไรก็ตาม จุดอ่อนของ Heap Sort คือความซับซ้อนในแง่ของการทำความเข้าใจและการนำไปใช้งานเมื่อเปรียบเทียบกับอัลกอริธึมอื่นๆ เช่น Insertion Sort หรือ Bubble Sort ซึ่งมีโครงสร้างที่เข้าใจง่ายกว่า นอกจากนี้ แม้ว่า Heap Sort จะมีเวลาในการทำงานคงที่ในทุกกรณี แต่การทำงานของ heapify และการจัดการโครงสร้าง heap อาจทำให้ Heap Sort ช้ากว่า Quick Sort ในกรณีที่ Quick Sort ทำงานได้ดี เช่นในกรณีที่เลือก pivot ได้ดี

การประยุกต์ใช้งานของ Heap Sort มีอยู่หลากหลาย เช่น การสร้าง priority queues, การจัดการลำดับความสำคัญในระบบ CPU scheduling และการจัดเรียงข้อมูลในระบบที่มีข้อมูลขนาดใหญ่ซึ่งไม่สามารถเก็บทั้งหมดในหน่วยความจำได้ (external sorting) ด้วยประสิทธิภาพในการจัดการข้อมูล Heap Sort จึงถูกใช้ในระบบคอมพิวเตอร์ที่ต้องการการประมวลผลข้อมูลอย่างรวดเร็วและมีประสิทธิภาพ

การแบ่งหัวข้อรายงานแบบที่นำเสนอนี้ช่วยให้เนื้อหาครอบคลุมทุกด้านที่สำคัญ ไม่ว่าจะเป็นในส่วนของทฤษฎีการทำงาน ข้อดีข้อเสีย และการประยุกต์ใช้งาน ทั้งนี้ยังช่วยให้ผู้อ่านเข้าใจการทำงานของ Heap Sort ได้อย่างชัดเจนและง่ายต่อการนำไปประยุกต์ใช้ในสถานการณ์จริง โดยเฉพาะอย่างยิ่งในระบบที่ต้องการประสิทธิภาพในการจัดการข้อมูล