SORTING ALGORITHMS AND COMPLEXITY ANALYSIS

(อ้างอิงจาก สมชาย ประสิทธิ์จูตระกูล, การออกแบบและวิเคราะห์อัลกอริทึม)

ค่ายอบรม สอวน. 58/2 มหาวิทยาลัยเกษตรศาสตร์

หัวข้อ

- ประสิทธิภาพของขั้นตอนวิธี
- อัตราการเติบโต (order of growth)
- สัญกรณ์เชิงเส้นกำกับ (asymptotic notations)
- ขั้นตอนวิธีการเรียงลำดับ (sorting algorithms)
  - Selection sort, Insertion sort, Bubble sort
  - Merge sort, Quick sort
  - Heap sort
  - Shell sort
- Empirical analysis of algorithms

ประสิทธิภาพของขั้นตอนวิธี

- การวิเคราะห์ขั้นตอนวิธี เป็นการวิเคราะห์ประสิทธิภาพการทำงานของขั้นตอนวิธี (algorithm efficiency) ใน 2 ลักษณะ คือ
  - วิเคราะห์ประสิทธิภาพเชิงเวลา (time efficiency) – เวลาที่ใช้ในการประมวลผล
  - วิเคราะห์ประสิทธิภาพเชิงพื้นที่ (space efficiency) – พื้นที่บนหน่วยความจำที่ใช้เก็บข้อมูลระหว่างการประมวลผล

การวิเคราะห์ running time

- เวลาที่ใช้ในการทำงาน (running time) ของขั้นตอนวิธีขึ้นอยู่กับข้อมูลนำเข้า
  - การเรียงลำดับข้อมูล 1,000 จำนวน ใช้เวลามากกว่าการเรียงลำดับข้อมูล 3 จำนวน
  - วิธีการเรียงลำดับข้อมูลบางวิธีใช้เวลาแตกต่างกันแม้ว่าจำนวนข้อมูลนำเข้า 2 ชุด จะมีขนาดเท่ากัน
    - Insertion sort บนข้อมูลที่เรียงลำดับ (sorted order) แล้วกับข้อมูลที่เรียงลำดับแบบกลับหลัง (reverse order)
- ขนาดของข้อมูลนำเข้าจะขึ้นอยู่กับลักษณะของปัญหา
  - โดยทั่วไป เป็นจำนวนข้อมูลหลักที่ถูกดำเนินการ เช่น ขนาดของ array หรือ list ในปัญหาการเรียงลำดับ หรือปัญหาการค้นหาข้อมูล
  - หรืออาจหมายถึงค่าอื่น เช่น
    - ปัญหาการคูณเลข 2 จำนวน → จำนวนหลัก (หรือบิต) ของเลขแต่ละจำนวน
    - ปัญหาเกี่ยวกับกราฟ → จำนวน vertices และจำนวน edges ของกราฟ

การวิเคราะห์ running time

- Running time ของขั้นตอนวิธีเขียนอยู่ในรูปของฟังก์ชันของจำนวน input
  - ลักษณะของฟังก์ชันบอกอัตราการเติบโตของฟังก์ชันเมื่อ input มีจำนวนเพิ่มขึ้น
  - การเปรียบเทียบประสิทธิภาพของขั้นตอนวิธีพิจารณาจากอัตราการเติบโต หรือ order of growth ของฟังก์ชัน เฉพาะกรณีจำนวน input มีจำนวนสูงมากๆ

Order of growth

<u>ตัวอย่าง 1</u> พิจารณาฟังก์ชันของ running time 3 ฟังก์ชันต่อไปนี้

$$T_1(n) = t_5(n)+t_6(n-1)+t_3+t_7$$

$$T_2(n) = t_{11}(n)+(t_6+t_3)(n-1)$$

$$T_3(n) = t_{22}(n)+(t_{23}+t_{24}+t_{25})(n-1)$$

ฟังก์ชันทั้งสามเป็นฟังก์ชันเชิงเส้น นั่นคือฟังก์ชันทั้งสามมีค่าเพิ่มขึ้น k เท่า เมื่อจำนวน input เพิ่มขึ้น k เท่า

<u>ตัวอย่าง</u> ถ้า $T_1(n) = 1000 \log n$ และ $T_2(n) = n$

พบว่า $1000 \log n > n$ เมื่อ $n < 3551$ เท่านั้น

ถ้าฟังก์ชัน f(n) โตเร็วกว่า g(n) แล้วค่าของ f ต้องมากกว่า g เมื่อ $n >= n_0$

n	1000 log n
3549	3,550.11
3550	3,550.23
3551	3,550.35
3552	3,550.47
3553	3,550.60
3554	3,550.72
3555	3,550.84
3556	3,550.96
3557	3,551.08

Order of growth

- ใช้ f(n) $\prec$ g(n) แทน f(n) โตช้ากว่า g(n) โดยมีนิยาม คือ

$$f(n) \prec g(n) \text{ ก็ต่อเมื่อ } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- กรณีที่ลิมิตข้างต้นมีค่าเป็นอนันต์ หมายความว่า f(n) โตเร็วกว่า g(n)
- กรณีที่ลิมิตข้างต้นมีค่าเป็นอื่นๆ ที่ไม่ใช่ 0 หรืออนันต์ หมายความว่า f(n) โตพอกันกับ g(n)

<u>กฎของโลปิตาล</u> ถ้า f(n) และ g(n) เป็นฟังก์ชันที่หาอนุพันธ์ได้ โดย $\lim_{n\to\infty} f(n) = \infty$ และ $\lim_{n\to\infty} g(n) = \infty$ แล้ว $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

Order of growth

<u>ตัวอย่าง 2</u> จงเรียงลำดับอัตราการเติบโตของ 0.5^n, 1, log n, n, 10^n

<u>วิธีทำ</u> พบว่า $0.5^n \prec 1 \prec \log n$ เพราะ ...

และ $\lim_{n\to\infty} \frac{\log n}{n} = \lim_{n\to\infty} \frac{(1/\ln 10)(1/n)}{1} = 0$ ดังนั้น $\log n \prec n$

ทำนองเดียวกัน $10^{\log n} \prec 10^n$ ดังนั้น $n \prec 10^n$

ดังนั้น $0.5^n \prec 1 \prec \log n \prec n \prec 10^n$

Order of growth

<u>ตัวอย่าง 3</u> จงเปรียบเทียบอัตราการเติบโตของ $\ln^9 n$ กับ $n^{0.1}$

<u>วิธีทำ</u> จากกฎของโลปิตาล

$$\lim_{n\to\infty}\frac{ln^9 n}{n^{0.1}} = \lim_{n\to\infty}\frac{(9ln^8 n)(1/n)}{0.1n^{0.1-1}} = \lim_{n\to\infty}\frac{9ln^8 n}{0.1n^{0.1}} = \lim_{n\to\infty}\frac{9\cdot 8ln^7 n}{0.1^2 n^{0.1}} = \cdots$$
$$= \lim_{n\to\infty}\frac{9\cdot 8\cdots 1ln^0 n}{0.1^9 n^{0.1}} = 0$$

สรุปได้ว่า $\ln^9 n \prec n^{0.1}$

จากตัวอย่างที่ 2 สามารถแสดงให้เห็นได้อีกว่า $\log^a n \prec n^b$ สำหรับจำนวนจริง b > 0 และฐานของ log เป็นค่าใดก็ได้ที่มีค่ามากกว่า 1

(เนื่องจากการเปลี่ยนฐานของ log ทำได้โดยการคูณด้วยค่าคงตัวหนึ่ง ซึ่งไม่มีผลกับค่าของลิมิต)

$\log^a n \prec n^b$, b > 0 หมายความว่า ฟังก์ชัน polylogarithmic โตช้ากว่าฟังก์ชัน polynomial

Order of growth

<u>ตัวอย่าง 4</u> จงเปรียบเทียบอัตราการเติบโตของ n^{10} กับ 2^n

<u>วิธีทำ</u> จาก $\log^a n \prec n^b$ แสดงว่า $\lg^{10} n \prec n^2$ และเมื่อแทน lg n ด้วย n จะได้ว่า $n^{10} \prec 2^n$ (เมื่อ lg แทน log ฐาน 2)

จากตัวอย่างที่ 3 สามารถแสดงให้เห็นได้อีกว่า $n^a \prec b^n$ สำหรับจำนวนจริง $b > 1$

$n^a \prec b^n$, $b > 1$ หมายความว่า ฟังก์ชัน polynomial โตช้ากว่าฟังก์ชัน exponential

Asymptotic notations

- สัญกรณ์เชิงเส้นกำกับ (asymptotic notations) เป็นสัญลักษณ์ที่ใช้แทนพฤติกรรมของฟังก์ชันในการวิเคราะห์ประสิทธิภาพของขั้นตอนวิธี
- Asymptotic notation
  - Θ-notation (Big-Theta notation)
  - o-notation/O-notation (Small/Big-O notation)
  - ω-notation/Ω-notation (Small/Big-Omega notation)
- Asymptotic efficiency (statistics)
  - The efficiency of an estimator within the limiting value as the size of the sample increases.

Big-Theta notation

- $\Theta(g(n))$ = {f(n): มีค่าคงที่บวก c_1, c_2 และ n_0 ที่ทำให้ $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ เมื่อ $n \geq n_0$} หรือ

  $\Theta(g(n)) = \{f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0, c \neq \infty\}$

- เมื่อ $f(n) \in \Theta(g(n))$ อาจเขียนแทนด้วย $f(n) = \Theta(g(n))$

  สังเกตว่าเราจะไม่เขียน $\Theta(g(n)) = f(n)$

- เมื่อ $n \geq n_0$, g(n) เป็นฟังก์ชันที่กำหนดขอบเขตการเติบโตของ f(n) ทั้งขอบเขตบนและล่าง ในระยะของค่าคงที่ c_1 และ c_2

g(n) is an asymptotic tight bound for f (n).

Small and Big-O notation

- $o(g(n)) = \{f(n) | \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$ หรือ $o(g(n))$ เป็นเซตของฟังก์ชันที่โตช้ากว่า $g(n)$
- เมื่อ $f(n) \in O(g(n))$ หมายความว่า $f(n)$ เป็นฟังก์ชันที่โตไม่เร็วกว่า $g(n)$ หรืออาจกล่าวว่า $O(g(n)) = o(g(n)) \cup \Theta(g(n))$

- $O(g(n)) = \{f(n)$: มีค่าคงที่บวก c และ n_0 ที่ทำให้ $0 \leq f(n) \leq cg(n)$ เมื่อ $n \geq n_0\}$
- $o(g(n)) = \{f(n)$: มีค่าคงที่บวก c และ n_0 ที่ทำให้ $0 \leq f(n) < cg(n)$ เมื่อ $n \geq n_0\}$
- $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since $\Theta(g(n))$ is stronger notation than $O(g(n))$ or, we can say that $\Theta(g(n)) \subseteq O(g(n))$.

g(n) is an asymptotic upper bound for f(n).

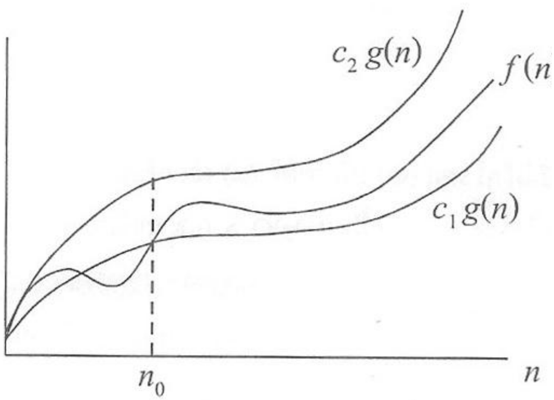
Small and Big-Omega notation

- $\omega(g(n)) = \{f(n) | \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty\}$ หรือ $\omega(g(n))$ เป็นเซตของฟังก์ชันที่โตเร็วกว่า g(n)
- เมื่อ $f(n) \in \Omega(g(n))$ หมายความว่า f(n) เป็นฟังก์ชันที่โตไม่ช้ากว่า g(n) หรืออาจกล่าวว่า $\Omega(g(n)) = \omega(g(n)) \cup \Theta(g(n))$

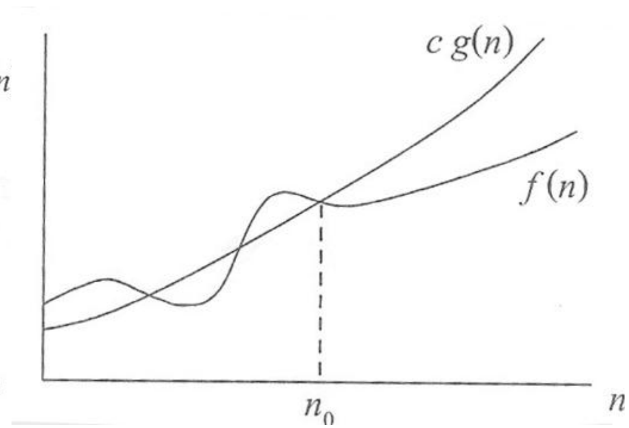
- $\Omega(g(n)) = \{f(n)$: มีค่าคงที่ c และ n_0 ที่ทำให้ $0 \le cg(n) \le f(n)$ เมื่อ $n \ge n_0\}$
- $\omega(g(n)) = \{f(n)$: มีค่าคงที่ c และ n_0 ที่ทำให้ $0 \le cg(n) < f(n)$ เมื่อ $n \ge n_0\}$
- $f(n) = \Theta(g(n))$ implies $f(n) = \Omega(g(n))$, since $\Theta(g(n))$ is stronger notation than $\Omega(g(n))$ or, we can say that $\Theta(g(n)) \subseteq \Omega(g(n))$.

g(n) is an asymptotic lower bound for f(n).

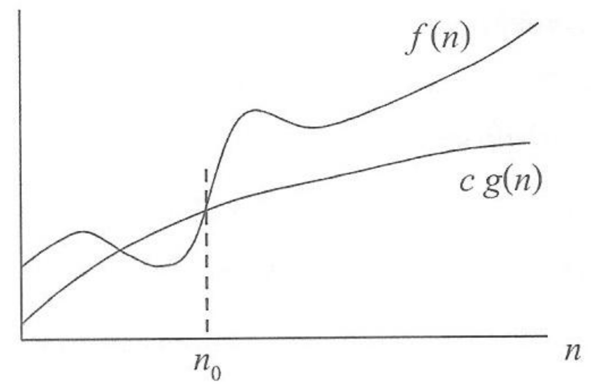
Asymptotic notations



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Asymptotic notations properties

- Transitivity
  - ถ้า $f(n) \in \Theta(g(n))$ และ $g(n) \in \Theta(h(n))$ แล้ว $f(n) \in \Theta(h(n))$
  - ถ้า $f(n) \in O(g(n))$ และ $g(n) \in O(h(n))$ แล้ว $f(n) \in O(h(n))$
  - ถ้า $f(n) \in \Omega(g(n))$ และ $g(n) \in \Omega(h(n))$ แล้ว $f(n) \in \Omega(h(n))$
  - ถ้า $f(n) \in o(g(n))$ และ $g(n) \in o(h(n))$ แล้ว $f(n) \in o(h(n))$
  - ถ้า $f(n) \in \omega(g(n))$ และ $g(n) \in \omega(h(n))$ แล้ว $f(n) \in \omega(h(n))$

Asymptotic notations properties

- Reflexivity
  - $f(n) \in \Theta(f(n))$
  - $f(n) \in O(f(n))$
  - $f(n) \in \Omega(f(n))$
- Symmetry
  - $f(n) \in \Theta(g(n))$ ก็ต่อเมื่อ $g(n) \in \Theta(f(n))$
- Transpose symmetry
  - $f(n) \in O(g(n))$ ก็ต่อเมื่อ $g(n) \in \Omega(f(n))$
  - $f(n) \in o(g(n))$ ก็ต่อเมื่อ $g(n) \in \omega(f(n))$

Asymptotic notations properties

- กำหนดให้ $f_1(n) = O(g_1(n))$ และ $f_2(n) = O(g_2(n))$ จะได้ว่า
  - $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
  - $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
  - $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
  - $f_1(n)^k = O(g_1(n)^k)$

Asymptotic notations properties

ถ้า $f_1(n) \in O(g_1(n))$ และ $f_2(n) \in O(g_2(n))$ แล้ว $f_1(n)+ f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

<u>พิสูจน์</u> สำหรับจำนวนจริงใดๆ a_1, b_1, a_2 และ b_2

ถ้า $a_1 \leq b_1$ และ $a_2 \leq b_2$ แล้ว $a_1 + a_2 \leq 2\max\{b_1, b_2\}$

จาก $f_1(n) \in O(g_1(n))$ จะได้ว่า $f_1(n) \leq c_1 g_1(n)$ สำหรับทุก $n \geq n_1$

และ $f_2(n) \in O(g_2(n))$ จะได้ว่า $f_2(n) \leq c_2 g_2(n)$ สำหรับทุก $n \geq n_2$

ให้ $c_3 = \max\{c_1, c_2\}$ และเมื่อพิจารณา $n \geq \max\{n_1, n_2\}$ แล้ว

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\leq c_3 2\max\{g_1(n), g_2(n)\} \end{aligned}$$

ดังนั้น $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

เมื่อ $c = 2c_3 = 2\max\{c_1, c_2\}$ และ $n_0 = \max\{n_1, n_2\}$.

Asymptotic notations properties

- ทฤษฎีบทนี้มีความหมายว่า ประสิทธิภาพโดยรวมของขั้นตอนวิธีถูกกำหนดค่าโดยส่วนการทำงานที่มีอัตราการเติบโตที่สูงกว่า (หรือส่วนที่มีประสิทธิภาพต่ำกว่า)
- เช่น ปัญหา Identical elements arrays ประกอบด้วยการทำงาน 2 ส่วน คือ
  - การจัดเรียงข้อมูลใน array
    - มีจำนวนครั้งของการเปรียมเทียบน้อยกว่าหรือเท่ากับ $1/2n(n-1)$ ครั้ง คิดเป็น $O(n^2)$
  - การตรวจสอบความเท่ากันของสมาชิกในลำดับที่อยู่ติดกัน
    - มีจำนวนครั้งของการเปรียบเทียบน้อยกว่าหรือเท่ากับ $n-1$ คิดเป็น $O(n)$
  - สรุปประสิทธิภาพรวมของขั้นตอนวิธี คือ $O(\max\{n^2, n\}) = O(n^2)$

Limits for comparing orders of growth

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n) \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n) \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

- สองกรณีแรกมีความหมายว่า f(n) ∈ O(g(n))
- สองกรณีหลังมีความหมายว่า f(n) ∈ Ω(g(n))
- กรณีที่สองมีความหมายว่า f(n) ∈ Θ(g(n))

Limits for comparing orders of growth

- <u>ตัวอย่าง 5</u> เปรียบเทียบอัตราการเติบโตของ 1/2n(n–1) และ n^2

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}$$

ดังนั้น 1/2n(n–1) $\in \Theta(n^2)$

- <u>ตัวอย่าง 6</u> เปรียบเทียบอัตราการเติบโตของ $\log_2 n$ และ $\sqrt{n}$

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{2}}{\frac{1}{2\sqrt{\mathrm{n}}}}=2\log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n}=0$$

ดังนั้น $\log_2 n \in O(\sqrt{n})$

Asymptotic vs. real number comparison

- $f(n) = O(g(n))$ $\approx$ $a <= b$
- $f(n) = \Omega(g(n))$ $\approx$ $a >= b$
- $f(n) = \Theta(g(n))$ $\approx$ $a = b$
- $f(n) = o(g(n))$ $\approx$ $a < b$
- $f(n) = \omega(g(n))$ $\approx$ $a > b$

- เราอาจไม่สามารถเปรียบเทียบเชิง asymptotic ของสองฟังก์ชันใดๆ เหมือนกับการเปรียบเทียบจำนวนจริงใดๆ ($a<b$, $a>b$ หรือ $a=b$) อย่างใดอย่างหนึ่งได้เสมอไป

  นั่นคือ เราอาจไม่สามารถบอกอย่างใดอย่างหนึ่งว่า $f(n) = O(g(n))$ หรือ $f(n) = \Omega(g(n))$ เช่น $f(n) = n$ และ $g(n) = n^{1+\sin n}$ เนื่องจากค่าของตัวชี้กำลัง $1+\sin n$ มีค่าแกว่งไปมาระหว่าง 0 ถึง 2 ทำให้ไม่สามารถหาความสัมพันธ์เชิง asymptotic ระหว่างทั้งสองฟังก์ชันได้

Asymptotic notations examples

<u>ตัวอย่าง 7</u> จงแสดงว่า $2n^2+500n+1000 \log n = O(n^2)$

<u>วิธีทำ</u> หาค่า c และ n_0 ที่ทำให้ $2n^2+500n+1000 \log n <= cn^2$ เป็นจริง เมื่อ $n >= n_0$

ในที่นี้ ให้ $c = 1502$ และ $n_0 = 1$ จะได้ว่า

$2n^2 + 500n + 1000 \log n <= 1502n^2$ เป็นจริง เมื่อ $n >= 1$

เช่น เมื่อ $n=1$ แล้ว $2 \cdot (1)^2 + 500 \cdot (1) + 1000 \log (1) = 502 <= 1502 \cdot (1)^2$ เป็นจริง

<u>ตัวอย่าง 8</u> จงแสดงว่า $2n^2+500n+1000 \log n = O(n^{200})$

<u>วิธีทำ</u> จาก $n^2 <= n^{200} = O(n^{200})$ ดังนั้น $2n^2+500n+1000 \log n = O(n^{200})$

เรียกว่าเป็นการระบุขอบเขตบนเป็นแบบ ขอบเขตหลวม (loose bound) ซึ่งต่างจากแบบขอบเขตกระชับ (tight bound)

Asymptotic notations examples

<u>ตัวอย่าง 9</u> จงแสดงว่า $(n/2) \lg (n/2) = \Omega(n \lg n)$

<u>วิธีทำ</u> หาค่า c และ n_0 ที่ทำให้ $cn \lg n <= (n/2) \lg (n/2)$ เป็นจริง เมื่อ $n >= n_0$

เขียนใหม่เป็น $cn \lg n <= (n/2) \lg n - (n/2) \lg 2$

หารตลอดด้วย $n \lg n$ ได้ $c <= (1/2) - (1/2) \lg 2/(\lg n) = (1/2) - (1/2)/(\lg n)$

สมมติให้ $n = 4$ จะได้ $c <= (1/2) - (1/2)/(\lg 4) = 1/2 – 1/4 = 1/4$

ดังนั้น อสมการเป็นจริงเมื่อ $c = 1/4$ และ $n_0 = 4$

Asymptotic notations examples

<u>ตัวอย่าง 10</u> จงแสดงว่า $3 \lg n + \lg \lg n = O(\lg n)$

<u>วิธีทำ</u> หาค่า c และ n_0 ที่ทำให้ $3 \lg n + \lg \lg n <= c \lg n$ เป็นจริง เมื่อ $n >= n_0$

หารตลอดด้วย $n \lg n$ ได้ $3 + (\lg \lg n)/(\lg n) <= c$

สมมติให้ $n = 2$ จะได้ $3 + (\lg \lg 2)/(\lg 2) = 3 + 0 <= c$

สมมติให้ $n = 4$ จะได้ $3 + (\lg \lg 4)/(\lg 4) = 3 + 1/2 <= c$

ดังนั้น อสมการเป็นจริงเมื่อ $c = 4$ และ $n_0 = 2$

Asymptotic notations examples

<u>ตัวอย่าง 11</u> จงแสดงว่า log n! = Θ(n log n)

<u>วิธีทำ</u> จาก n! = n·(n-1) ·(n-2) ...2·1

เมื่อแทนแต่ละพจน์ด้วย n จะได้ว่า n! <= n^n จากนั้นหาค่า log จะได้ว่า

log n! <= log n^n <= n log n = O(n log n)

และเมื่อแทนแต่ละพจน์ n, (n-1), ..., (n/2) ด้วย (n/2) และแทนแต่ละพจน์ (n/2-1), (n/2-2), ..., 2, 1 ด้วย 1 จะได้ว่า n! >= $(n/2)^{n/2} \cdot 1^{n/2}$ จากนั้นหาค่า log จะได้ว่า

log n! >= log $(n/2)^{n/2}$ >= (n/2) log (n/2) = Ω(n log n)

ดังนั้น สรุปได้ว่า log n! = Θ(n log n)

Asymptotic notations examples

<u>ตัวอย่าง 12</u> จงแสดงว่า $\log_a n = \Theta(\log_b n)$ สำหรับค่าคงที่ a, b > 1

<u>วิธีทำ</u> จาก $\log_a n = (\log_b n) / (\log_b a)$

ดังนั้น สรุปได้ว่า $\log_a n = \Theta(\log_b n)$

จะเห็นได้ว่า $\log_{13} n$ กับ $\log_{100} n$ มีอัตราการเติบโตเท่ากัน คือ $\Theta(\log_b n)$ เมื่อ b > 1

การใส่ฐานของ log ในสัญกรณ์เชิงเส้นกำกับ จึงไม่มีผลใดๆ และอาจละไม่เขียนได้

Asymptotic notations examples

<u>ตัวอย่าง 13</u> จงแสดงว่า $\log n^a = \Theta(\log n)$ สำหรับค่าคงที่ $a > 0$

<u>วิธีทำ</u> จาก $\log n^a = a \log n = \Theta(\log n)$

จะเห็นได้ว่า $\log n^{100}$ กับ $\log n^{1/2}$ มีอัตราการเติบโตเท่ากัน คือ $\Theta(\log n)$

เลขชี้กำลังภายใน log จึงไม่มีความสำคัญ

Asymptotic notations examples

<u>ตัวอย่าง 14</u> จงแสดงว่า $a^{\lg n} \neq \Theta(a^{\log n})$

<u>วิธีทำ</u> จาก $a^{\log_b n} = n^{\log_b a}$ ดังนั้น $a^{\lg n} = n^{\lg a}$

เนื่องจาก lg n ไม่เท่ากับ log n ดังนั้น $n^{\lg a} \neq \Theta(n^{\log a})$ แสดงว่า $a^{\lg n} \neq \Theta(a^{\log n})$

ฐานของ log ที่เป็นเลขชี้กำลังของพจน์อื่น ไม่อาจละทิ้งไปได้

Asymptotic notations examples

<u>ตัวอย่าง 15</u> **จงแสดงว่า** $7n - 2 = O(n)$

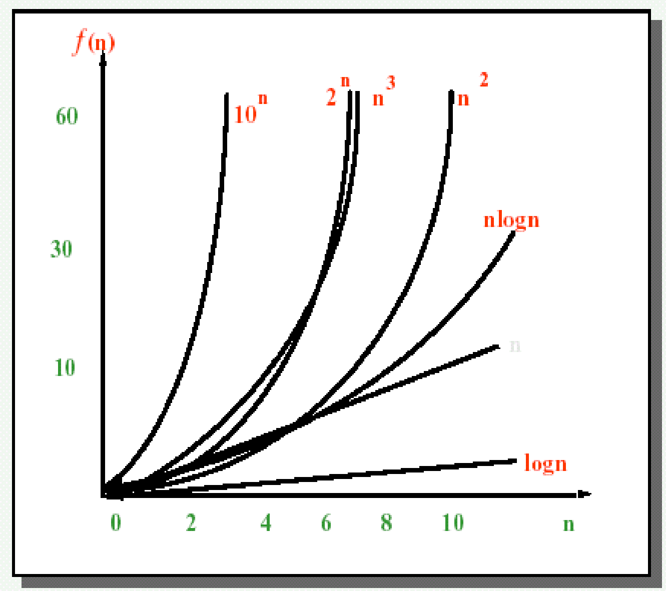
<u>ตัวอย่าง 16</u> **จงแสดงว่า** $3n^3 + 20n^2 + 5 = O(n^3)$

Big-O and growth rate

- We can use the big-O notation to rank functions according to their growth rate from best to worst
  - O(1)
  - O(log n)
  - O(n)
  - O(n log n)
  - O(n^2)
  - O(n^3)
  - O(2^n)
  - O(n!)

Complexity Classes

Common Complexity Classes (growth curves)



Asymptotic efficiency class

- Constant time: O(1)
  - An algorithm is O(1) when its running time is independent of the number of data items.
  - The algorithm runs in constant time.
- Logarithmic Time: O(log n)
  - An algorithm is O(log n) is among the best algorithms.
  - Typically a result of cutting a problem's size by a constant factor on each iteration of the algorithm.
  - When the number of elements doubles, the number of operations increases by just 1.

Asymptotic efficiency class

- Linear Time Algorithms: O(n)
  - An algorithm is O(n) when its running time is proportional to the size of the list.
  - When the number of elements doubles, the number of operations doubles.
- n log n Algorithms: O(n log n)
  - An algorithm is O(n log n) are from many divide-and-conquer algorithms.
  - A little bit slower than O(n) but it is still consider very good algorithm.

Asymptotic efficiency class

- Quadratic & Cubic : $O(n^2)$ & $O(n^3)$
  - Algorithms with running time $O(n^2)$ are quadratic.
    - typically, algorithms with two embedded loops
    - practical only for relatively small values of n.
    - whenever n doubles, the running time of the algorithm increases by a factor of 4.
  - Algorithms with running time $O(n^3)$are cubic.
    - typically, algorithms with three embedded loops
    - efficiency is generally poor; doubling the size of n increases the running time eight-fold.

Asymptotic efficiency class

- Exponential : $O(2^n)$
  - Typical for algorithms that generate all subsets of an n-element set.
  - Often the term exponential is used in a broader sense to include this and larger orders of growth as well.
  - Considered as bad algorithm.
- Factorial Time : $O(n!)$
  - Typical for algorithms that generate all permutations of an n-element set.
  - Considered as worst algorithm.
  - $O(n!)$, sometimes, reads Oh! No! Algorithm.

Algorithm efficiency

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Table 1.1 Running times for different sizes of input. "nsec" stands for nanoseconds, "μ" is one microsecond and "cent" stands for centuries.

Running time

- การวัดประสิทธิภาพเชิงเวลา (time efficiency) ทำโดยการวัด running time ของขั้นตอนวิธี ซึ่งได้แก่การนับจำนวนคำสั่งมูลฐาน (primitive operations) ที่ถูกดำเนินการ
- คำสั่งมูลฐานเป็นคำสั่งที่ทำงานเสร็จโดยไม่ขึ้นกับขนาดของข้อมูลที่ประมวลผล นั่นคือ ใช้เวลา $\Theta(1)$ หรือใช้เวลาเป็นค่าคงที่หนึ่ง

```
isPrime(p):
   if (((p-1)!+1) % p == 0)
      return true
   else
      return false
```

- จะเห็นว่าการประมวลผลนิพจน์ ((p-1)!+1) % p ใช้เวลาแปรตามค่าของ p ดังนั้น คำสั่งนี้จึงไม่จัดเป็นคำสั่งมูลฐาน

Selection sort

- เลือกข้อมูลที่มีค่าสูงสุดจากชุดข้อมูลที่ยังไม่ได้เรียงลำดับ แล้วนำไปวางในตำแหน่งที่ถูกต้อง นั่นคือ สลับตำแหน่งระหว่างข้อมูลที่มีค่าสูงสุดกับข้อมูลในตำแหน่งสุดท้ายในส่วนที่ยังไม่ได้จัดเรียง ทำเช่นนี้จนกระทั่งข้อมูลทุกตัวอยู่ในลำดับที่ถูกต้อง
- สมมติให้ข้อมูลใน array A เมื่อเริ่มต้นมีค่าเป็น <29, 10, 14, 37, 13>

[1]	[2]	[3]	[4]	[5]
29	10	14	37	13
29	10	14	13	37
13	10	14	29	37
13	10	14	29	37
10	13	14	29	37

Selection sort


```
01: SelectionSort(A, 1, n):
02:    for (i=n; i--; i>=2)
03:       j = indexMax(A, 1, i)
04:       swap(A[i], A[j])
```

ในที่นี้คำสั่งในบรรทัดที่ 3 ไม่ใช่คำสั่งมูลฐาน จึงเขียนใหม่ได้เป็น

```
01: SelectionSort(A[1..n]):
02:    for (i=n; i--; i>=2)
03:       j = 1
04:       for (k=2; k<=i; k++)
05:          if (A[j] < A[k])
06:             j = k
07:       swap(A[i], A[j])
```

ถึงแม้คำสั่ง swap จะประกอบด้วย 3 คำสั่งย่อย แต่ในที่นี้คำสั่ง swap เป็นคำสั่งมูลฐาน

Selection sort running time


```
01: SelectionSort(A, 1, n):
02:   for (i=n; i--; i>=2)
03:     j = 1
04:     for (k=2; k<=i; k++)
05:       if (A[j] < A[k])
06:         j = k
07:     swap(A[i], A[j])
```

คำสั่งบรรทัดที่ 2 ทำงาน _____ ครั้ง

คำสั่งบรรทัดที่ 3 ทำงาน _____ ครั้ง

คำสั่งบรรทัดที่ 7 ทำงาน _____ ครั้ง

คำสั่งบรรทัดที่ 4 ทำงาน _____ ครั้ง

คำสั่งบรรทัดที่ 5 ทำงาน _____ ครั้ง

คำสั่งใดใน SelectionSort จัดเป็นคำสั่งมาตรเวลา?

คำสั่งมาตรเวลาเป็นคำสั่งในขั้นตอนวิธีที่ถูกใช้ทำงานมากที่สุด เวลาการทำงานรวมของขั้นตอนวิธีจะแปรผันตามจำนวนครั้งที่คำสั่งมาตรเวลาประมวลผล

ให้ f(n) เป็นฟังก์ชันของ running time ของ Selection sort บนแถวลำดับขนาด n และ

f(n) = ...

สรุปได้ว่า f(n) = ...

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ for

- ถ้าชุดคำสั่งภายในลูปใช้เวลา t_i ในการทำงานรอบที่ i และลูปทำงาน m รอบ การทำงานรวมทั้งหมดจะเท่ากับ $t_1+ t_2+\ldots+ t_m$

```
01: SelectionSort(A, 1, n):
02:    for (i=n; i--; i>=2)
03:      j = indexMax(A, 1, i)
04:      swap(A[i], A[j])
```

- บรรทัดที่ 3 ใช้เวลา $\Theta(i)$ และบรรทัดที่ 4 ใช้เวลา $\Theta(1)$ ดังนั้น ชุดคำสั่งในลูปใช้เวลา $\Theta(i)+ \Theta(1) = \Theta(i)$ และลูปทำงานทั้งหมด n-1 รอบ ดังนั้นเวลาการทำงานรวมเท่ากับ

$$\sum_{i=2}^{n} \Theta(i) = \Theta\left(\sum_{i=2}^{n} i\right) = \Theta\left(\frac{n(n+1)}{2} - 1\right) = \Theta(n^2)$$

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ for

<u>ตัวอย่าง 17</u> **จงวิเคราะห์ขั้นตอนวิธีต่อไปนี้**

```
01: for (i=1; i++; i<=m)
02:    for (j=1; j++; j>=m)
03:       t += A[i, j]
```

บรรทัดที่ 3 ใช้เวลา $\Theta(1)$ ดังนั้นเวลาการทำงานรวมเท่ากับ

$$\sum_{i=1}^{m}\sum_{j=1}^{m}\Theta(1) = \sum_{i=1}^{m}\Theta\left(\sum_{j=1}^{m}1\right) = \sum_{i=1}^{m}\Theta(m) = \Theta\sum_{i=1}^{m}m = \Theta(m^2)$$

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ for

<u>ตัวอย่าง 18</u> จงวิเคราะห์ขั้นตอนวิธีต่อไปนี้

```
for (i=1; i++; i<=m)
    for (j=1; j++; j>=i)
        t += A[i, j]
```

ลูปชั้นในใช้เวลา Θ(i) ดังนั้นเวลาการทำงานรวมเท่ากับ

$$\sum_{i=1}^{m} \Theta(i) = \Theta \sum_{i=1}^{m} i = \Theta\left(\frac{m(m+1)}{2}\right) = \Theta(m^2)$$

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ for

<u>ตัวอย่าง 19</u> จงวิเคราะห์ขั้นตอนวิธีต่อไปนี้

```
01: for (i=2; i++; i<=m-1)
02:    for (j=3; j++; j>=i)
03:       t += A[i, j]
```

สังเกตว่าตัวแปร i และ j มีค่าเริ่มต้นที่ต่างไปจากตัวอย่างที่ผ่านมา แต่การใช้สัญกรณ์เชิงเส้นกำกับช่วยให้ละสิ่งเหล่านี้ได้ บรรทัดที่ 3 ใช้เวลา $\Theta(1)$ ดังนั้นเวลาการทำงานรวมเท่ากับ

$$\sum_{i=2}^{m-1}\sum_{j=3}^{i}\Theta(1)=\Theta\left(\sum_{i=2}^{m-1} i\right)=\Theta(m^2+\Theta(m))=\Theta(m^2)$$

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ for

<u>ตัวอย่าง 20</u> จงวิเคราะห์ขั้นตอนวิธีต่อไปนี้

```
01: oneFunc(G(V, E)):
02:    c = 0
03:    for (each vertex u in v)
04:       for (each vertex w adjacent to u)
05:          c++
06:    return c
```

บรรทัดที่ 5 เป็นคำสั่งมาตรเวลา ที่ทำงานเท่ากับผลบวกของดีกรีของทุกๆ จุดในกราฟ หรือเท่ากับสองเท่าของเส้นเชื่อม ดังนั้นจึงใช้เวลาการทำงานรวมเท่ากับ $\Theta(|E|)$

แต่หากทุกจุดไม่เชื่อมกับจุดอื่นเลย บรรทัดที่ 5 จะไม่ถูกประมวลผลเลย กรณีนี้คำสั่งมาตรเวลาจะอยู่ในบรรทัดที่ 3 ดังนั้นขั้นตอนวิธีนี้ใช้เวลารวมเท่ากับ $O(|V|+|E|)$

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ while

<u>ตัวอย่าง 21</u> **จงวิเคราะห์ขั้นตอนวิธีต่อไปนี้**

```
01: while (n >= 0)
02:    n=⌊n/2⌋
```

สังเกตพบว่าในแต่ละรอบค่าตัวแปร n **ลดลงทีละครึ่งหนึ่ง ทำให้ในรอบที่** k **ย่อมมีค่า** $\frac{n}{2^k}$ **และหยุดการวนซ้ำเมื่อ** n **มีค่า** 1 **และเป็นรอบที่** k **ที่มีค่าเท่ากับ** lg n **ดังนั้นเวลาการทำงานรวมเท่ากับ** $\Theta(\lg n) = \Theta(\log n)$

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ while

<u>ตัวอย่าง 22</u> **จงวิเคราะห์ขั้นตอนวิธีต่อไปนี้**

01: i=1, j=n

02: while (i < j)

03: i = i+3

04: j = j-5

จากบรรทัดที่ 3 **และ** 4 **พบว่าในแต่ละรอบค่าของ** i **และ** j **จะขยับเข้าหากันทีละ** 8 **ทำให้จำนวนรอบของการวนซ้ำจึงมีค่าอย่างมาก** n/8 **รอบ ดังนั้นเวลาการทำงานรวมเท่ากับ** Θ(n)

หาค่าน้อยที่สุดในต้นไม้ทวิภาค

<u>ตัวอย่าง 23</u> จงวิเคราะห์ขั้นตอนวิธี FindMin ในต้นไม้ทวิภาค

```
01: BinaryNode FindMin(BST T):
02: BinaryNode p
03: p = T.root
04: if (p != NULL)
05:    while (p.left != NULL)
06:       p = p.left
07: return p
```

ขนาดของข้อมูลได้แก่ขนาดของต้นไม้ทวิภาค T ซึ่งแทนด้วยจำนวนโหนดในต้นไม้ T ในที่นี้สมมติให้มีค่า n

ในกรณีที่ต้นไม้ทวิภาคไม่สมดุลที่สุดโหนดต่างๆ ในต้นไม้ T จะเรียงกันเป็นแนวยาวไปด้านซ้ายของ parent node ทำให้การค้นหาโหนดที่มีค่าน้อยที่สุดต้องเดินผ่านทุกโหนด (บรรทัดที่ 6) ในต้นไม้ ดังนั้นเวลาในการทำงานรวมเท่ากับ O(n)

Insertion Sort

- ลักษณะการทำงานเปรียบเทียบได้กับการหยิบไพ่จากกองมาเรียงในมือ เมื่อเริ่มต้นจำนวนไพ่ในมือเป็นศูนย์ จากนั้นให้หยิบไพ่จากกองทีละ 1 ใบ มาใส่ในลำดับที่ถูกต้อง ทำเช่นนี้จนจำนวนไพ่ในกองเป็นศูนย์ และพบว่าไพ่ทั้งหมดในมือจัดเรียงในลำดับที่ถูกต้อง
- สมมติให้ข้อมูลใน array A เมื่อเริ่มต้นมีค่าเป็น <5, 2, 4, 6, 1, 3>

Insertion sort

[1]	[2]	[3]	[4]	[5]	[6]
5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

```
InsertionSort(A, 1, n):
for (j=2; j++; j<=n)
   hold = A[j]
   i = j-1
   while (i > 0 and A[i] > hold)
      A[i+1] = A[i]
      i = i-1
   A[i+1] = hold
```

การวิเคราะห์การทำงานคำสั่งแบบวนซ้ำ while

<u>ตัวอย่าง 24</u> จงวิเคราะห์ขั้นตอนวิธี Insertion sort

```
01: InsertionSort(A, 1, n):
02: for (j=2; j++; j<=n)
03:    hold = A[j]
04:    i = j-1
05:    while (i > 0 and A[i] > hold)
06:       A[i+1] = A[i]
07:       i = i-1
08:    A[i+1] = hold
```

ลูปในบรรทัดที่ 5 จะหยุดทำงานในสองกรณี คือ เมื่อ i <= 0 หรือ A[i] <= hold
ในกรณีแรก จากบรรทัดที่ 7 พบว่าค่า i ลดลงทีละ 1 แสดงว่าจำนวนรอบการทำงานเป็น O(i) ส่วนในกรณีที่สองขึ้นกับลักษณะของข้อมูล
จากบรรทัดที่ 2 และ 4 พบว่าในแต่ละรอบของลูป for คำสั่ง while ทำงาน O(j) ส่วนคำสั่งในบรรทัดที่ 3, 4 และ 8 ทำงาน $\Theta(1)$ โดย $\Theta(1)$ + O(j) = O(j)

ดังนั้นเวลาในการทำงานรวมเท่ากับ

$$\sum_{j=2}^{n} O(j) = O(n^2)$$

การทำงานแบบ recursive

- ขั้นตอนวิธีที่ใช้เทคนิค recursive ในการแก้ปัญหา ประกอบด้วย 2 ส่วน คือ คือ ส่วนแรก ส่วนที่เรียกตัวเองซ้ำ (recursive call) ด้วยขนาดของปัญหาที่เล็กลง และลู่เข้าสู่ขนาดของปัญหาที่เล็กพอที่จะแก้ปัญหาด้วยวิธีปกติ ซึ่งจัดเป็นส่วนที่สอง
- ใช้ความสัมพันธ์แบบ recurrence T(n) ในการบอกเวลาในการประมวลผล ซึ่งเกิดจากเวลาในการทำงานของทั้งสองส่วนข้างต้น
- หาคำตอบของ T(n) ในรูปของสัญกรณ์เชิงเส้นกำกับ

Selection sort (recursive)

<u>ตัวอย่าง 25</u> จงวิเคราะห์ขั้นตอนวิธี Selection sort (recursive)

```
01: SelectionSort(A, 1, n):
02:    if (n <= 1)
03:       return
04:    j = indexMax(A, 1, n)
05:    swap(A[n], A[j])
06:    SelectionSort(A, 1, n-1)
```

บรรทัดที่ 4 เป็นคำสั่งมาตรเวลาของส่วน non-recursive ซึ่งใช้เวลา $\Theta(n)$
บรรทัดที่ 6 เป็นการทำงานแบบ recursive ดังนั้นเวลาการทำงานรวมเท่ากับ

$T(n) = T(n-1) + \Theta(n)$ สำหรับ $n>1$ และ
$T(1) = \Theta(1)$

Selection sort (recursive)

หาคำตอบของ T(n) ด้วยวิธีการแทนค่าแบบย้อนกลับ (backward substitution) ได้เป็น

$T(n) = T(n-1) + \Theta(n)$

$= T(n-2) + \Theta(n-1) + \Theta(n)$

- $T(n-1) = T(n-2) + \Theta(n-1)$

$= \ldots$

$= T(1) + \Theta(2) + \ldots + \Theta(n-1) + \Theta(n)$

$= \sum_{i=1}^{n} \Theta(i)$

$= \sum_{i=1}^{n} \Theta(i) = \Theta \sum_{i=1}^{n} i = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$

การค้นหาแบบทวิภาค (Binary search)

<u>ตัวอย่าง 26</u> จงวิเคราะห์ขั้นตอนวิธี Binary search

```
BibarySearch(A, lb, ub, key):
  if (lb > ub)
    return -1
  m = ⌊(lb+ub)/2⌋
  if (key = A[m])
    return m
  else if (key < A[m])
    return BinarySearch(A, lb, m-1, key)
  else
    return BinarySearch(A, m+1, ub, key)
```

- หาไม่พบ
- แบ่งครึ่ง array

การค้นหาแบบทวิภาค (Binary search)

บรรทัดที่ 2-6 ใช้เวลา $\Theta(1)$ และบรรทัดที่ 7-10 ใช้เวลา $\Theta(1)+T(n/2)$ ดังนั้นเวลาการทำงานรวมเป็นดังนี้

$T(n) <= T(n/2) + \Theta(1)$ สำหรับ $n>1$ และ $T(1) = \Theta(1)$

ใช้เครื่องหมาย <= เนื่องจากบรรทัดที่ 7-10 ถูกประมวลผลเมื่อเงื่อนไขบรรทัดที่ 5 เป็นเท็จ

$$
\begin{aligned}
T(n) &<= T(n/2) + \Theta(1) \\
&<= T(n/2^2) + \Theta(1) + \Theta(1) \\
&\ldots \\
&<= T(n/2^k) + \sum_{i=1}^{k} \Theta(1) \\
&<= \Theta(1) + \sum_{i=1}^{\lg n} \Theta(1) \\
&<= \Theta(\log n) = O(\log n)
\end{aligned}
$$

- $T(n/2) = T(n/2^2) + \Theta(1)$

ประเภทของการวิเคราะห์

- เมื่อพิจารณาขั้นตอนวิธีหาค่ามากที่สุด (หรือหาค่าน้อยที่สุด) ของข้อมูลใน array ที่มีขนาด n เราพบว่าเวลาในการทำงานรวมเท่ากับ $\Theta(n)$ โดยการเรียงลำดับของข้อมูลใน array ไม่ส่งผลต่อเวลาการทำงานดังกล่าว
- เมื่อข้อมูลมีขนาดเพิ่มขึ้นก็จะใช้เวลาในการทำงานเพิ่มขึ้นเป็นเชิงเส้น
- สำหรับการเรียงลำดับแบบ Insertion sort การเรียงลำดับขอข้อมูลใน array ตั้งต้นที่แตกต่างกัน ได้แก่ random order, reverse order หรือ sorted order มีผลกับเวลาในการทำงานด้วย

ประเภทของการวิเคราะห์


```
01: InsertionSort(A, 1, n):
02: for (j=2; j++; j<=n)
03:    hold = A[j]
04:    i = j-1
05:    while (i > 0 and A[i] > hold)
06:       A[i+1] = A[i]
07:       i = i-1
08:    A[i+1] = hold
```

- เมื่อพิจารณาในบรรทัดที่ 5 ซึ่งจะหยุดทำงานในสองกรณี คือ (1) เมื่อ i <= 0 หรือ (2) เมื่อ A[i] <= hold ซึ่งในกรณีที่สองนี้ขึ้นอยู่กับลักษณะของข้อมูล โดยอาจเป็น
  - Sorted order หรือ
  - Reverse order

ประเภทของการวิเคราะห์

- ให้ I_n เป็นเซตของลักษณะข้อมูลนำเข้าทุกๆ แบบ
- ให้ T(n, i) เป็นเวลาการทำงานของขั้นตอนวิธีที่มีข้อมูลขนาด n ในแบบที่ i เมื่อ $i \in I_n$
- ประเภทการวิเคราะห์ขั้นตอนวิธีมี 3 ประเภท ประกอบด้วย
  - การวิเคราะห์กรณีเลวสุด (worst case analysis) คิดเฉพาะลักษณะข้อมูลที่ทำให้ใช้เวลาในการทำงานมากที่สุด คือ

    $$T_{worst}(n) = \max(T(n, i)) \text{ เมื่อ } i \in I_n$$

  - การวิเคราะห์กรณีเฉลี่ย (average case analysis) บางครั้งลักษณะของข้อมูลที่ทำให้เกิดการทำงานแบบเลวสุดมีโอกาสเกิดขึ้นน้อยมากๆ และเราทราบความน่าจะเป็นที่ลักษณะข้อมูลแบบต่างๆ จะเกิดขึ้นมีค่าเท่ากับ p(i) เราจะได้ว่า

    $$T_{avg}(n) = \sum p(i) \cdot T(n, i) \text{ เมื่อ } i \in I_n$$

    (ปัญหาคือ เรามักไม่ทราบค่า p(i) ดังนั้น จึงมักใช้ $p(i) = 1/|I_n|$)

ประเภทของการวิเคราะห์

- ประเภทการวิเคราะห์ขั้นตอนวิธีมี 3 ประเภท (ต่อ)
  - การวิเคราะห์กรณีถัวเฉลี่ย (amortized analysis) การทำงานกับข้อมูลเดียวกันในแต่ละครั้งอาจใช้เวลาเร็วบ้าง ช้าบ้าง ดังนั้นการวิเคราะห์กรณีถัวเฉลี่ยจะวิเคราะห์โดยพิจารณาลักษณะเหล่านี้ด้วย

- สำหรับการวิเคราะห์กรณีที่ดีที่สุด (best case analysis) ไม่นำมาใช้เปรียบเทียบประสิทธิภาพของขั้นตอนวิธี เนื่องจากค่าดังกล่าวไม่เป็นประโยชน์มากนัก หรืออาจเป็นเพียงกรณีเดียวของลักษณะข้อมูลที่เป็นไปได้ทั้งหมด เช่น
  - ปัญหาการค้นหาข้อมูลแบบลำดับ (สมมติให้ข้อมูลมีค่าไม่ซ้ำกัน) การพบข้อมูลในตำแหน่งแรก
  - ปัญหา Selection sort ที่มีการตรวจสอบสถานะการเรียงลำดับของข้อมูลก่อน โดยถ้าเรียงลำดับอยู่แล้วก็ไม่ต้องทำ selection sort

Sequential search

<u>ตัวอย่าง 27</u> จงวิเคราะห์ขั้นตอนวิธี Sequential search ในกรณีเลวสุดและกรณีเฉลี่ย

```
01: SequentialSearch(A, n, key):
02: i = n
03: while (i>0 and A[i] ≠ key)
04:    i--
05: return i
```

- บรรทัดที่ 3 เป็นคำสั่งมาตรเวลา ซึ่งทำงานจำนวนครั้งมากที่สุดเท่ากับ n+1 ในกรณีที่พบในตำแหน่งสุดท้ายหรือไม่พบเลย ดังนั้น $T_{worst}(n) = \Theta(n)$

Sequential search

- สำหรับกรณีเฉลี่ย
  - สมมติให้ p เป็นความน่าจะเป็นของการหาค่า key พบโดย 0 ≤ p ≤ 1
  - สมมติให้ความน่าจะเป็นที่จะหา key พบในตำแหน่งที่ i ที่ค่าเท่ากับ p/n สำหรับทุก i

ดังนั้น

$$T_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n(1-p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + n] + n(1-p)$$
$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1-p)$$
$$= \frac{p(n+1)}{2} + n(1-p) = \frac{p}{2} + \left(1 - \frac{p}{2}\right)n$$
$$= \Theta(n)$$

Binary search

<u>ตัวอย่าง 28</u> จงวิเคราะห์ขั้นตอนวิธี Binary search-1 ในกรณีเลวสุดและกรณีเฉลี่ย

```
01: BinarySearch-1(A, n, key):
02: i=1; j=n
03: while (i <= j)
04:    m = ⌊(i+j)/2⌋
05:    if (key = A[m])
06:       return m
07:    else if (key < A[m])
08:       j = m-1
09:    else
10:       i = m+1
```

- บรรทัดที่ 4 เป็นคำสั่งมาตรเวลา ซึ่งพบว่าค่าของ (j-i+1) มีค่าเริ่มต้นจาก n และลดลงทีละครึ่ง
- กรณีเลวสุด จะลดลงจน i > j จึงสรุปได้ว่า $T_{worst}(n) = \Theta(\log n)$

Empirical analysis of algorithm

- บางขั้นตอนวิธีอาจมีความยุ่งยากในการวิเคราะห์ด้วยวิธีการทางคณิตศาสตร์
- เราอาจใช้วิธีการวิเคราะห์เชิงประจักษ์ (empirical analysis) ซึ่งได้แก่การ implement โปรแกรมและวัดประสิทธิภาพที่ได้จากการประมวลผลบนคอมพิวเตอร์

- เหตุผลในการทำ empirical analysis
  - เพื่อตรวจสอบความแม่นยำของการคำนวณประสิทธิภาพ
  - เพื่อเปรียบเทียบประสิทธิภาพของขั้นตอนวิธีต่างๆ ของปัญหาเดียวกัน
  - เพื่อเปรียบเทียบประสิทธิภาพของการ implement ขั้นตอนวิธีเดียวกันในรูปแบบต่างๆ
  - เพื่อสร้างความเชื่อมั่นในประสิทธิภาพของโปรแกรมบนคอมพิวเตอร์เครื่องใดเครื่องหนึ่ง
  - ...

Empirical analysis of algorithm

- วิธีการวัดประสิทธิภาพของโปรแกรม
  - แทรก counter เพื่อนับจำนวนครั้งการประมวลผลคำสั่งมาตรเวลา
  - จับเวลาการทำงานของโปรแกรม โดยต้องตระหนักว่า
    - เวลาของระบบอาจไม่แน่นอน แม้จะเป็นข้อมูลนำเข้าชุดเดียวกัน
    - เวลาอาจสั้นมาก จนมีค่าเป็น 0
    - ในการประมวลผลแบบ multitasking system เวลาที่วัดได้อาจรวมเวลาที่ CPU ประมวลผลโปรแกรมอื่นในขณะเดียวกัน

Empirical analysis of algorithm

- เลือกข้อมูลนำเข้าที่มักพบเห็นเป็นส่วนใหญ่ (typical inputs)
  - ในบางปัญหามีเซตข้อมูลสำหรับใช้เทียบประสิทธิภาพ (benchmarking)
  - ในกรณีที่ผู้เขียนโปรแกรมเป็นผู้สร้างชุดข้อมูลนำเข้า ให้คำนึงถึง
    - ขนาดของข้อมูลทดสอบ (sample size)
      - เริ่มต้นจากชุดข้อมูลทดสอบขนาดเล็ก
      - ทดสอบกับข้อมูลที่มีขนาดเท่ากันแต่มีลักษณะที่แตกต่างกัน
    - พิสัยของชุดข้อมูล
    - กระบวนการในการสร้างชุดข้อมูลอัตโนมัติ
      - มีรูปแบบที่แน่นอน เช่น 1000, 2000, ..., 10000
      - แบบสุ่ม

Mathematical vs. empirical analysis

- การวิเคราะห์ทางคณิตศาสตร์ (mathematical analysis)
  - ไม่ขึ้นกับข้อมูลนำเข้าชุดใดชุดหนึ่งโดยเฉพาะ
  - มีข้อจำกัด โดยเฉพาะอย่างยิ่งในการวิเคราะห์กรณีเฉลี่ย
- การวิเคราะห์เชิงประจักษ์ (empirical analysis)
  - ใช้ได้กับขั้นตอนวิธีใดๆ
  - ผลลัพธ์ที่ได้ขึ้นอยู่กับชุดข้อมูลนำเข้าชุดใดชุดหนึ่ง และคอมพิวเตอร์ที่ใช้ประมวลผล

Bubble sort

- เปรียบเทียบค่าระหว่างข้อมูลในลำดับถัดกันทีละคู่ สลับตำแหน่งข้อมูลในกรณีที่อยู่ในลำดับที่ไม่ถูกต้อง ดังนั้นเมื่อเปรียบเทียบข้อมูลครบทุกคู่ ข้อมูลที่มีค่าสูงสุดจะอยู่ในตำแหน่งที่ถูกต้อง ทำเช่นนี้จนกระทั่งข้อมูลทุกตัวอยู่ในตำแหน่งที่ถูกต้อง

```
BubbleSort(A, n):
for (pass=1; pass++; pass<=n-1)
   for (j=1; j++; j<=n-pass)
      if (A[j] > A[j+1])
         swap(A[j], A[j+1])
```

- การทำงานในแต่ละ pass จะมีผลให้ข้อมูลที่มีค่ามากที่สุดเคลื่อนไปอยู่ในตำแหน่งที่ถูกต้อง และเมื่อขึ้น pass ใหม่ จำนวนข้อมูลที่ยังไม่ได้เรียงลำดับจะลดลง 1 จำนวน

Bubble sort

สมมติให้ข้อมูลใน array A เมื่อเริ่มต้นมีค่าเป็น <25, 57, 48, 37, 12, 92, 86, 33>

"Pass 1"

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	57	48	37	12	92	86	33
25	57	48	37	12	92	86	33
25	48	57	37	12	92	86	33
25	48	37	57	12	92	86	33

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	48	37	12	57	92	86	33
25	48	37	12	57	92	86	33
25	48	37	12	57	86	92	33
25	48	37	12	57	86	33	<u>92</u>

Bubble sort

"Pass 2"

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	48	37	12	57	86	33	92
25	48	37	12	57	86	33	92
25	37	48	12	57	86	33	92
25	37	12	48	57	86	33	92

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	37	12	48	57	86	33	92
25	37	12	48	57	86	33	92
25	37	12	48	57	33	<u>86</u>	<u>92</u>

Bubble sort

"Pass 3"

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	37	12	48	57	33	86	92
25	37	12	48	57	33	86	92
25	12	37	48	57	33	86	92
25	12	37	48	57	33	86	92
25	12	37	48	57	33	86	92
25	12	37	48	33	57	86	92

"Pass 4"

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	12	37	48	33	57	86	92
12	25	37	48	33	57	86	92
12	25	37	48	33	57	86	92
12	25	37	48	33	57	86	92
12	25	37	33	48	57	86	92

Bubble sort

"Pass 5"

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
12	25	37	33	48	57	86	92
12	25	37	33	48	57	86	92
12	25	37	33	48	57	86	92
12	25	33	37	48	57	86	92

"Pass 6"

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
12	25	33	37	48	57	86	92
12	25	33	37	48	57	86	92
12	25	33	37	48	57	86	92

Bubble sort

"Pass 7"

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
12	25	33	37	48	57	86	92
12	25	33	37	48	57	86	92

- จากตัวอย่างพบว่าการทำงานใน pass ที่ 6 ไม่มีการสลับที่เกิดขึ้น แสดงว่าข้อมูลทุกจำนวนอยู่ในตำแหน่งที่ถูกต้องแล้ว จึงไม่จำเป็นต้องมีการทำงานใน pass ถัดไป
- ดังนั้น เราสามารถปรับปรุงการทำงานของ Bubble sort ให้มีการตรวจสอบการสลับที่ข้อมูลในแต่ละ pass และให้การทำงานหยุดลงก่อนที่จะทำงานครบทุก pass ในกรณีที่ pass ที่ผ่านมาไม่มีการสลับที่ข้อมูล

Bubble sort


```
BubbleSort(A, n):
exchange = true
pass = 1
while (pass <= n-1 and exchange)
    exchange = false
    for (i=1; i++; i<=n-pass)
        if (A[i] > A[i+1])
            swap(A[j], A[j+1])
            exchange = true
    pass = pass+1
```

Bubble sort

- ข้อมูลใน array A มีขนาด n และกำหนดให้ t_{pass} เป็นจำนวนครั้งของการประมวลผล for loop ในรอบที่ pass ดังนั้น แต่ละบรรทัดประมวลผลดังนี้

  คำสั่งบรรทัดที่ 2 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 3 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 4 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 5 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 6 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 7 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 8 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 9 ทำงาน __________ ครั้ง

  คำสั่งบรรทัดที่ 10 ทำงาน __________ ครั้ง

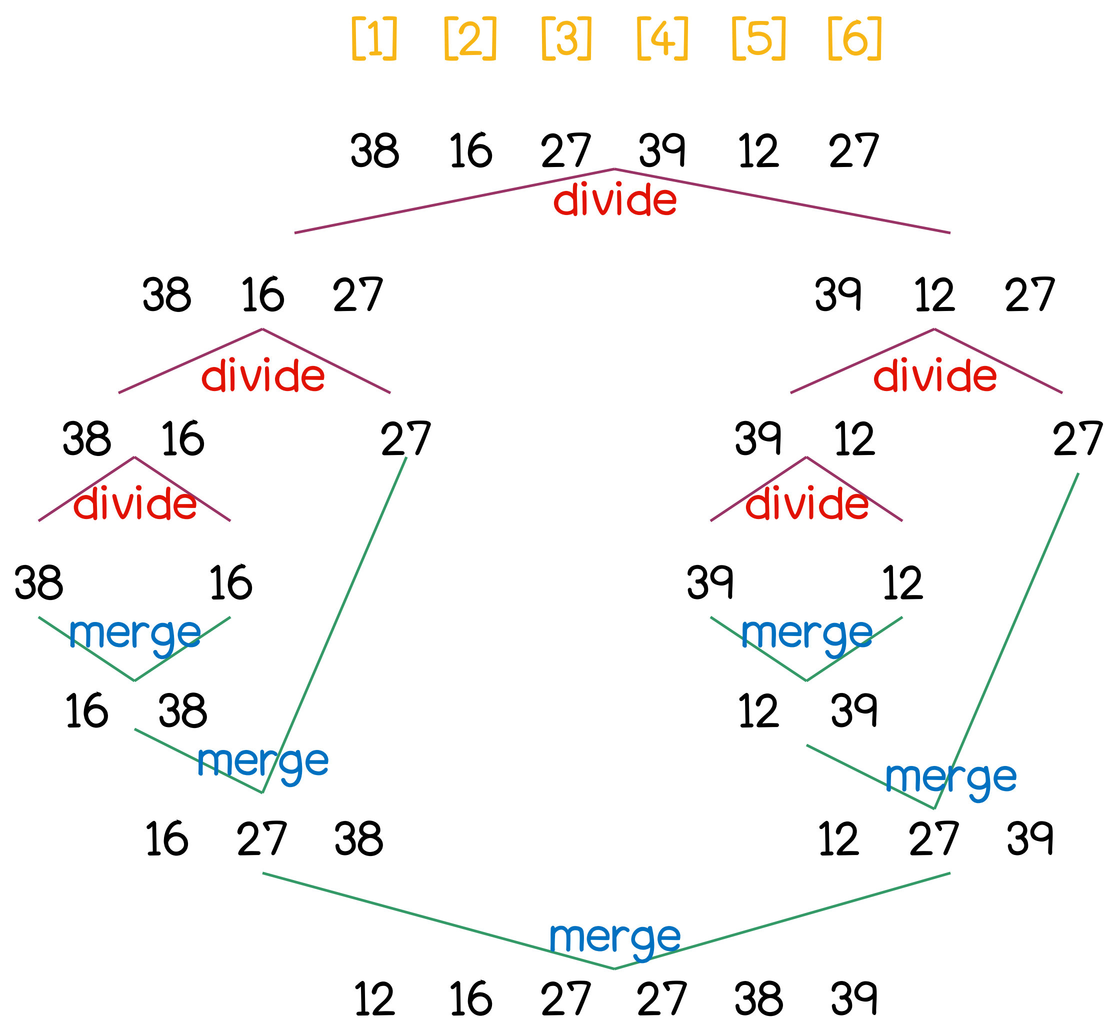
Bubble sort

- ดังนั้น เวลาการทำงานรวมเท่ากับ ……………..
- Worst case running time เกิดขึ้นเมื่อข้อมูลใน array อยู่ในลำดับแบบ reverse order
- ใน pass ที่ i มีการเปรียบเทียบทั้งหมด (n-i) ครั้ง และในกรณี worst case จะมีการ swap ข้อมูล (n-i) ครั้ง Bubble sort จึงมี data movement ในระดับ …………..
- Best case running time เกิดขึ้นเมื่อข้อมูลใน array อยู่ในลำดับแบบ sorted order และมีประสิทธิภาพการทำงานในระดับ …………..

Merge sort

- ใช้เทคนิค divide-and-conquer ในการแก้ปัญหา ซึ่งประกอบด้วย 3 ขั้นตอน คือ
  1. Divide แบ่งปัญหาออกเป็นปัญหาย่อยที่มีลักษณะเดียวกับปัญหาหลัก แต่มีขนาดเล็กกว่า
  2. Conquer แก้ปัญหาย่อยด้วยเทคนิค recursive
  3. Combine ผนวกคำตอบของแต่ละปัญหาย่อยให้เป็นคำตอบของปัญหาหลัก
- เทคนิค divide-and-conquer สำหรับการทำ merge sort ประกอบด้วย 3 ขั้นตอน คือ
  1. Divide แบ่ง array ออกเป็น 2 ส่วนเท่าๆกัน
  2. Conquer เรียงลำดับข้อมูลใน array ย่อยด้วยวิธีการ Merge sort
  3. Combine รวม array ย่อยที่เรียงลำดับแล้วเข้าด้วยกัน
- สมมติให้ข้อมูลใน array A เมื่อเริ่มต้นมีค่าเป็น <38, 16, 27, 39, 12, 27>

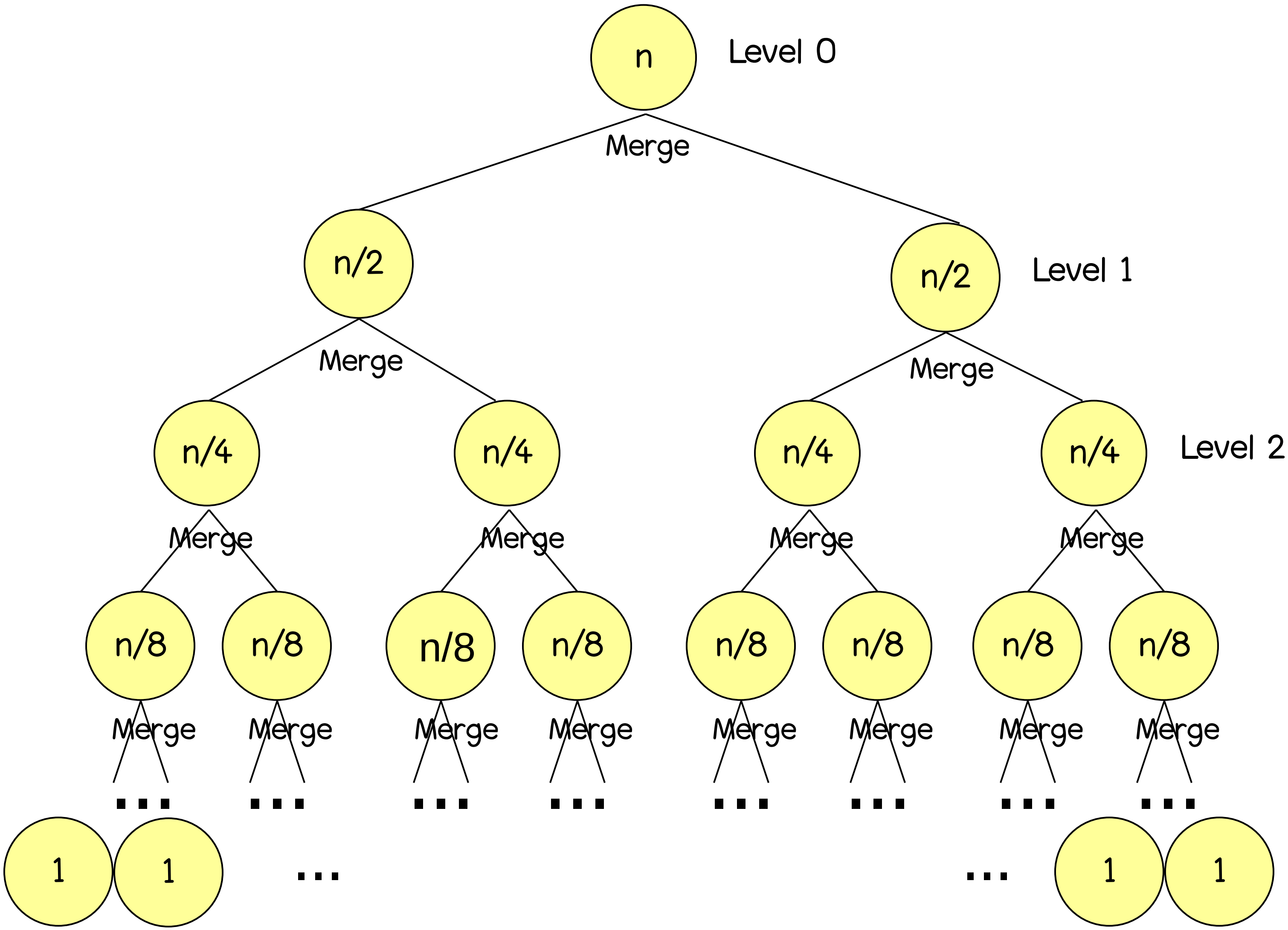
Merge sort


[1] [2] [3] [4] [5] [6]
38 16 27 39 12 27
divide
38 16 27
39 12 27
divide
divide
38 16
27
39 12
27
divide
divide
38
16
39
12
merge
merge
16 38
12 39
merge
merge
16 27 38
12 27 39
merge
12 16 27 27 38 39

Merge sort


```
MergeSort(A, lb, ub):   ▶ lb/ub is a lower/upper bound index of subarray A
if lb < ub
    j = ⌊(lb+ub) / 2⌋
    MergeSort(A, lb, j)
    MergeSort(A, j+1, ub)
    Merge(A, lb, j, ub)
```

Merge sort


n
Level 0
Merge
n/2
n/2
Level 1
Merge
Merge
n/4
n/4
n/4
n/4
Level 2
Merge
Merge
Merge
Merge
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
Merge
Merge
Merge
Merge
Merge
Merge
Merge
Merge
1
1
…
…
1
1

Merge sort

- สมมติให้การรวม array ย่อย 2 ชุดเข้าด้วยกัน จะได้ array ขนาด n ดังนั้น
  การ merge แต่ละครั้ง มีการเปรียบเทียบ ______ ครั้ง
  มีการเคลื่อนย้ายข้อมูลจาก array ย่อยทั้งสองไปยัง temporary array จำนวน ____ ครั้ง
  มีการเคลื่อนย้ายข้อมูลจาก temporary array กลับมายัง array A จำนวน ____ ครั้ง
  สรุปการรวม array ย่อยที่มีขนาดรวมกันเป็น n มีการทำงาน จำนวน ____ ครั้ง

- array A จะเริ่มต้นด้วยการ call MergeSort 2 ครั้ง และแต่ละ array ย่อย จะ call MergeSort 2 ครั้ง ดังนั้น level ทั้งหมดของการ call recursive มีค่าเป็น …………..

Merge sort

- ที่ level 0 มีการ call Merge จำนวน 1 ครั้ง มีการทำงาน 3n-1 ครั้ง
  ที่ level 1 มีการ call Merge จำนวน 2 ครั้ง มีการทำงาน 3n-2 ครั้ง
  ที่ level 2 มีการ call Merge จำนวน 4 ครั้ง มีการทำงาน 3n-4 ครั้ง
  .
  .
  ที่ level m มีการ call Merge จำนวน 2^m ครั้ง มีการทำงาน $3n-2^m$ ครั้ง
  นั่นคือ ในแต่ละ level มีการทำงานของการดำเนินการหลักเป็น O(n)
  รวมทุก level การดำเนินการหลักมีค่าเป็น lg n x O(n) = O (n log n)

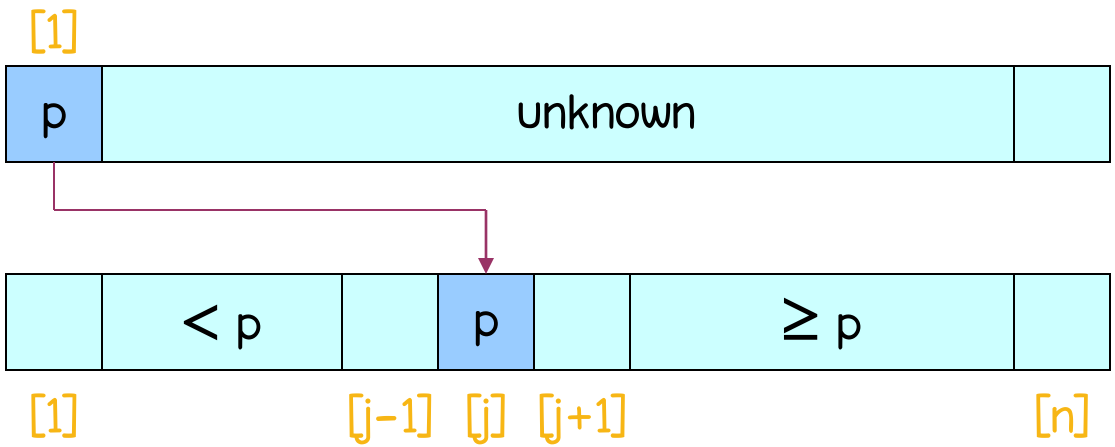
- ข้อเสียของ Merge Sort คือต้องใช้พื้นที่ภายนอก array ในระหว่างการรวม array ย่อย 2 array เข้าด้วยกัน

Quick sort

- สมมติให้ชุดข้อมูลจัดเก็บใน array A ขนาด n
  1. เลือกสมาชิกจากarray ตามเงื่อนไขที่กำหนด 1 ค่า สมมติให้มีค่าเป็น p
     เรียก p ว่า pivot element
  2. แบ่งชุดข้อมูลออกเป็น 2 ส่วน โดย
     สมาชิกใน array ย่อย A[1..j-1] มีค่าน้อยกว่า p และสมาชิกใน array ย่อย
     A[j+1..length(A)] มีค่ามากกว่าหรือเท่ากับ p
  3. ทำซ้ำขั้นตอนที่ 1-2 กับ array ย่อย A[1..j-1] และ array ย่อย A[j+1..n] จนกระทั่ง
     จำนวนสมาชิกใน array ย่อยมีขนาดเป็น 0

Quick sort

- ในที่นี้กำหนดให้ pivot element คือสมาชิกตัวแรกของ array หรือ array ย่อย



Quick sort

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
5	3	2	6	4	1	3	7
[3	3	2	4	1]	5	[6	7]
[1	2]	3	[4	3]	5	[6	7]
1	[2]	3	[4	3]	5	[6	7]
1	2	3	[4	3]	5	[6	7]

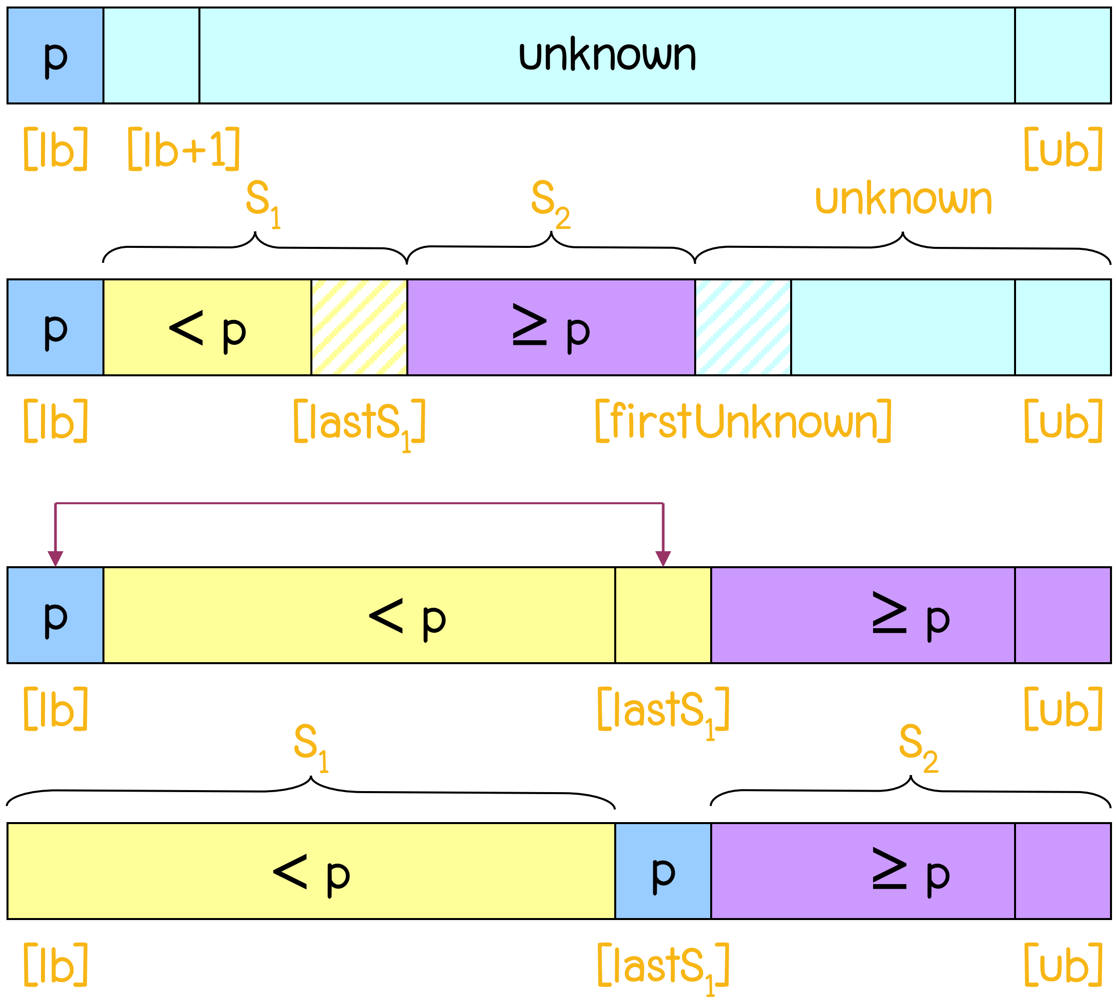
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
1	2	3	[3]	4	5	[6	7]
1	2	3	3	4	5	[6	7]
1	2	3	3	4	5	6	[7]
1	2	3	3	4	5	6	7

Quick sort


```
QuickSort(A, lb, ub):
if lb < ub
  j = Partition(A, lb, ub)
   QuickSort(A, lb, j-1)
   QuickSort(A, j+1, ub)
```

- lb/ub is a lower/upper bound index of subarray A

Quick sort


p
unknown
[lb]
[lb+1]
[ub]
S1
S2
unknown
p
< p
≥ p
[lb]
[lastS1]
[firstUnknown]
[ub]
p
< p
≥ p
[lb]
[lastS1]
[ub]
S1
S2
< p
p
≥ p
[lb]
[lastS1]
[ub]

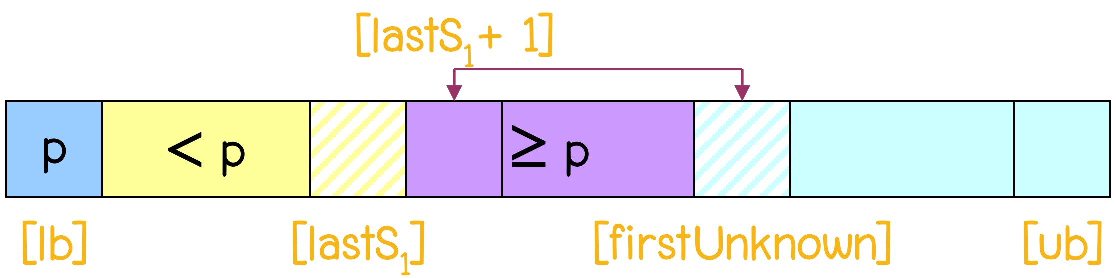
Quick sort


```
Partition(A, lb, ub):
p = A[lb]
LastS_1 = lb
firtstUnknown = lb + 1
while (firstUnknown ≤ ub)
   if A[firstUnknown] <  p)
      move A[firstUnknown] into S_1
   else
      move A[firstUnknown] into S_2
swap(A[lb], A[lastS_1])
return lastS_1
```

- lb/ub is a lower/upper bound index of subarray A
- S_1 has 1 element, S_2 is empty

Quick sort

How to move A[firstUnknown] into S_1?



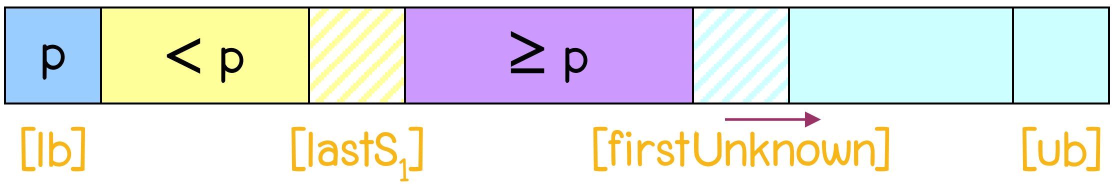
01: swap(A[firstUnknown], A[lastS$_1$+ 1])

02: LastS$_1$ = LastS$_1$+ 1

03: firtstUnknown = firstUnknown + 1

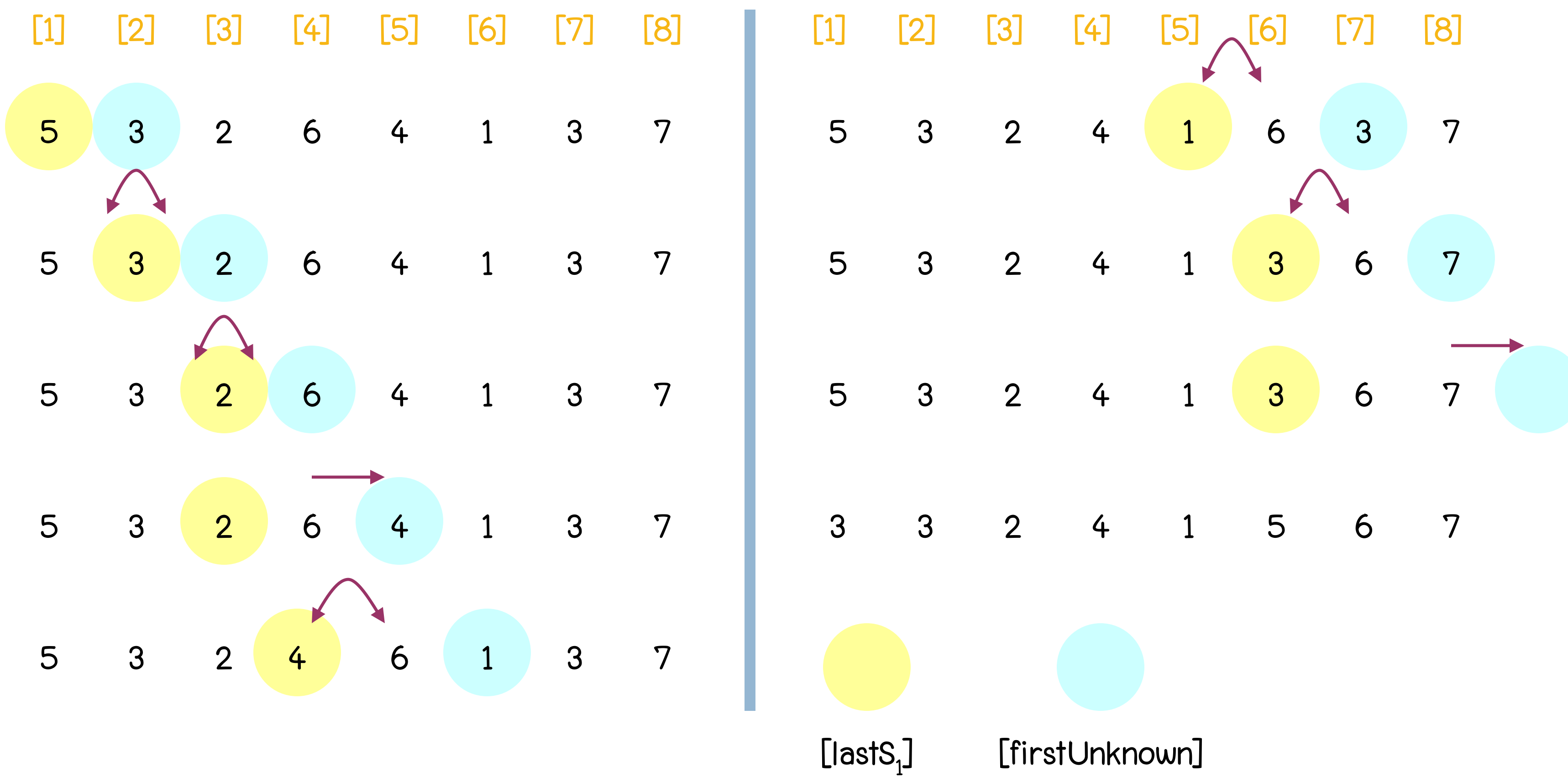
Quick sort

- How to move A[firstUnknown] into S_2?



01: firtstUnknown = firstUnknown + 1

Quick sort

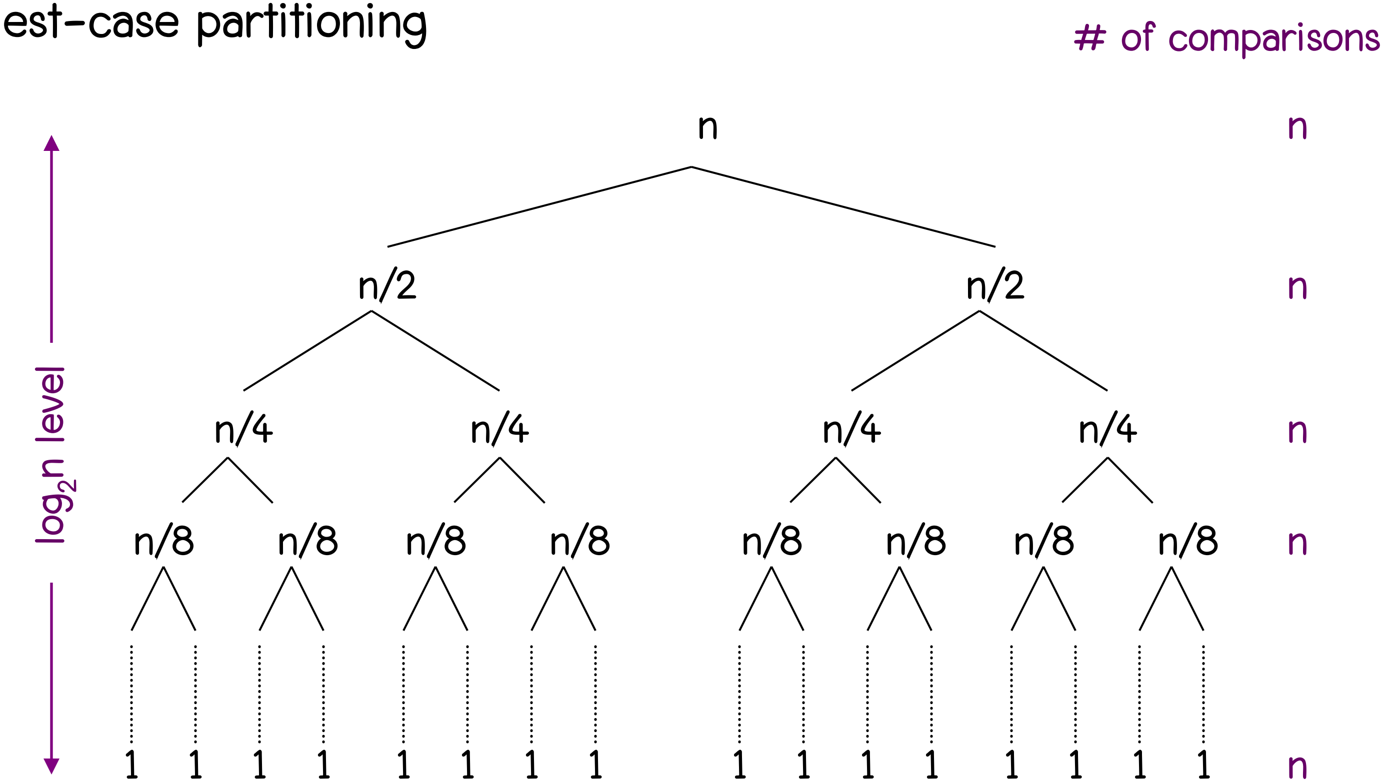

[1] [2] [3] [4] [5] [6] [7] [8]
5 3 2 6 4 1 3 7
5 3 2 6 4 1 3 7
5 3 2 6 4 1 3 7
5 3 2 6 4 1 3 7
5 3 2 4 6 1 3 7
[1] [2] [3] [4] [5] [6] [7] [8]
5 3 2 4 1 6 3 7
5 3 2 4 1 3 6 7
5 3 2 4 1 3 6 7
3 3 2 4 1 5 6 7
[lastS1]
[firstUnknown]

Quick sort

- การเลือกค่า pivot ที่เหมาะสม
  - Quick sort จะมีประสิทธิภาพ ถ้า array ย่อยที่ได้จากแบ่ง array แต่ละครั้งมีขนาดที่ใกล้เคียงกันมากที่สุด
  - ดังนั้น ค่า pivot ที่เหมาะสม จึงควรจะมีค่าใกล้เคียงกับค่ามัธยฐาน (median) ของ array หรือ array ย่อย
  - นอกจากนี้ ค่าของ pivot อาจไม่ใช่สมาชิกของ array หรือ array ย่อยก็ได้

Quick sort

Best-case partitioning



Quick sort

สมมติ array A มีขนาด n และ $n = 2^m$ (ดังนั้น $m = \log_2 n$)

กรณี best-case partitioning ซึ่งจะแบ่ง array ออกเป็น 2 ส่วนเท่าๆ กัน

	จำนวน array ย่อย	ขนาดของ array ย่อย	จำนวนครั้งของการเปรียบเทียบ
รอบที่ 1	1	n	$\approx n$
รอบที่ 2	2	n/2	$\approx n/2$
รอบที่ 3	4	n/4	$\approx n/4$
รอบที่ m	2^m	1	$\approx n/2^m$

จำนวนครั้งของการเปรียบเทียบรวม

$= n + 2(n/2) + 4(n/4) + \ldots + 2^m(n/2^m)$

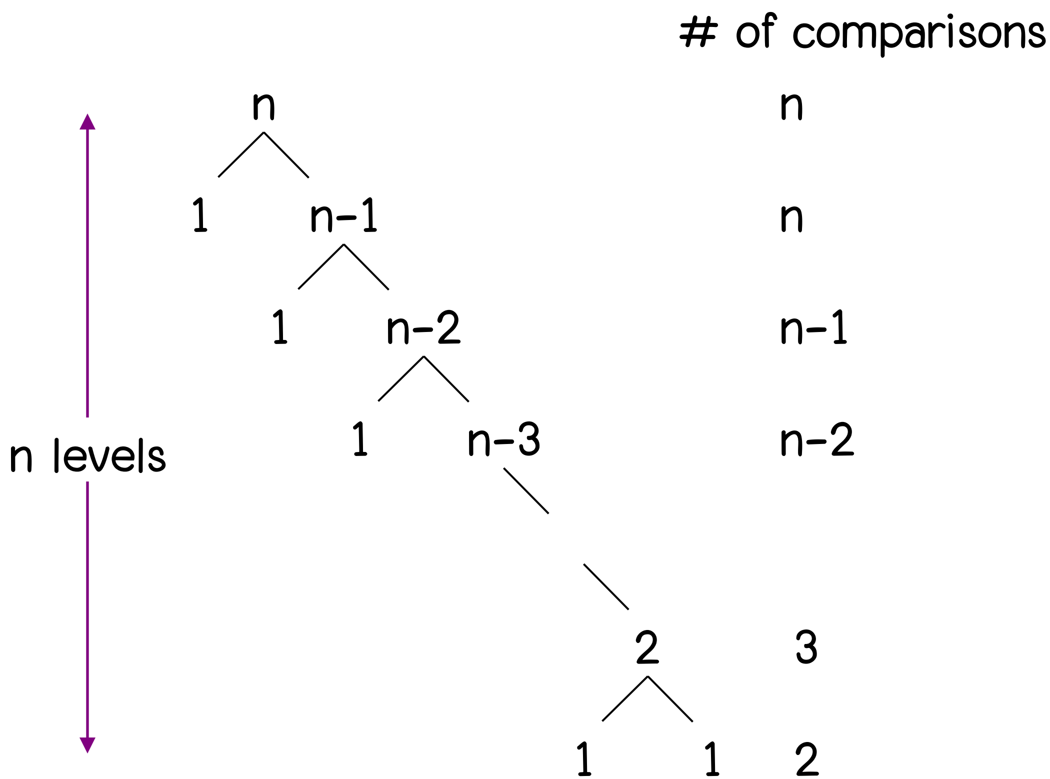
$= n + n + n + \ldots + n$ ▶ ทั้งหมด m terms

$= O(mn) = O(n \log n)$

ในกรณีที่ n <> 2^m Quick sort จะมีประสิทธิภาพประมาณ 1.38 $n\log_2 n$

Quick sort

Worst-case Partitioning ▶ A[lb] อยู่ในตำแหน่งที่ถูกต้องอยู่แล้ว



Quick sort

สมมติ array A มีขนาด n กรณี worst-case partitioning ซึ่งจะแบ่ง array ออกเป็น 2 ส่วน คือ array ย่อยที่มีขนาดเป็น 1 และ array ย่อยที่มีขนาด n-1

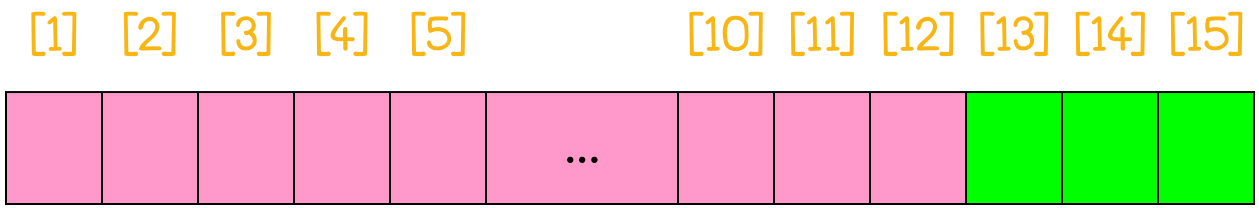
	จำนวน array ย่อย	ขนาดของ array ย่อย	จำนวนครั้งของการเปรียบเทียบ
รอบที่ 1	1	n	n-1
รอบที่ 2	1	n-1	n-2
รอบที่ 3	1	n-2	n-3
รอบที่ n	1	1	0

จำนวนครั้งของการเปรียบเทียบรวม

$= (n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = O(n^2)$

Heap

Heap เป็นโครงสร้างข้อมูลเชิงเส้น ที่สามารถมองให้อยู่ในรูปของ complete binary tree ได้

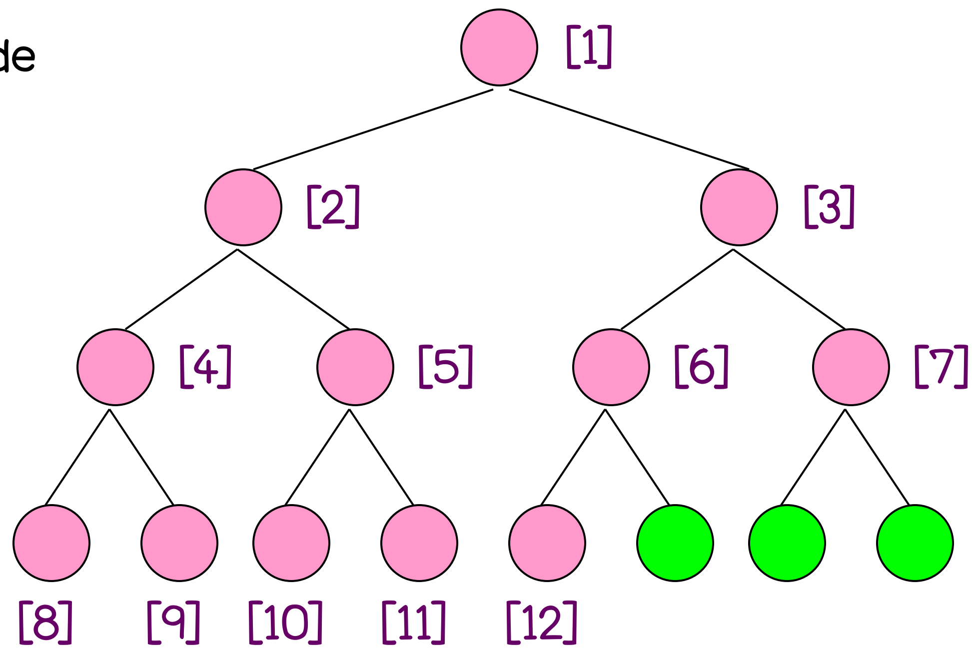


ถ้าสมมติให้ i เป็นตำแหน่งของ node ใน heap เราจะได้ว่า

parent(i) → return $\lfloor i/2 \rfloor$

left(i) → return 2i

right(i) → return 2i+1



Heap

- สมมติให้ array A ใช้เก็บข้อมูล heap และ heapSize(A) หมายถึง จำนวนสมาชิกของ heap ใน array A

  ดังนั้น ถ้า heapSize(A) ≤ length(A)

  แล้ว A[heapSize(A)+1..length(A)] ไม่เป็นส่วนหนึ่งของ heap
- สำหรับทุก node i ที่เป็นสมาชิกของ heap ยกเว้น root node, A[parent(i)] ≥ A[i]

  จากคุณสมบัติของ heap ทำให้สมาชิกที่ตำแหน่งของ root node มีค่าสูงที่สุด
- การทำงานกับ Heap ประกอบด้วย
  - BuildHeap เพื่อสร้าง heap จาก array ที่กำหนด
  - Heapify เพื่อปรับเปลี่ยนตำแหน่งของข้อมูล เพื่อให้โครงสร้างข้อมูลที่ได้จากการเพิ่ม/ลดข้อมูลจาก heap เดิม กลับมาเป็น heap อีกครั้ง
  - HeapSort เพื่อเรียงลำดับข้อมูลใน heap

Build heap


```
BuildHeap(A):
heapSize = length(A)
    ▸ A[⌊length(A)/2⌋ +1..length(A)] are all heaps with one element
for (j= ⌊length(A)/2⌋; j==1; j--)
    Heapify(A, j)
```

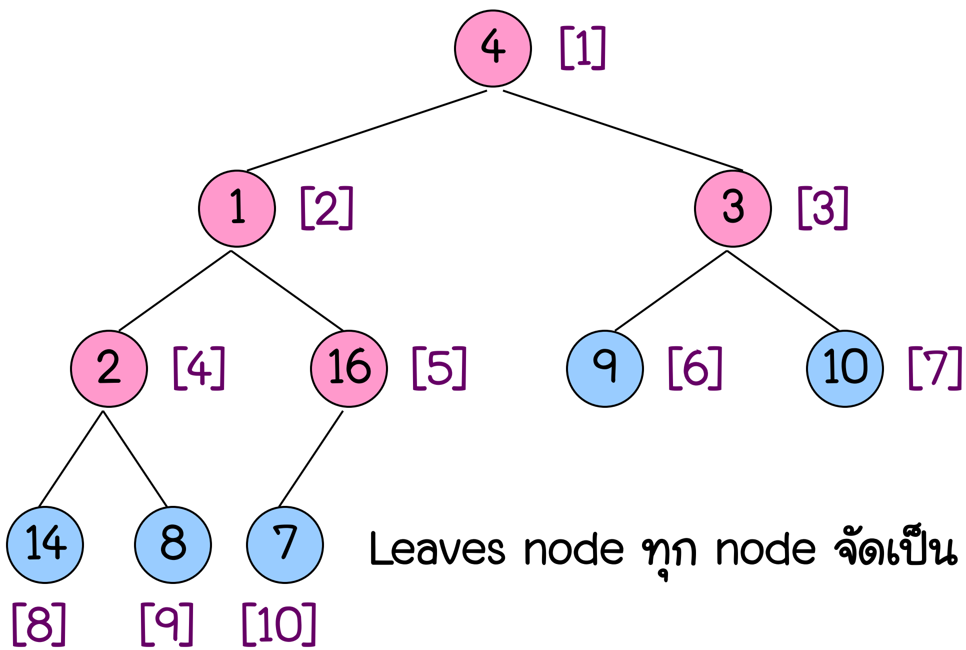
Heapify


```
01: Heapify(A, j):
02: l = left(j)
03: r = right(j)
04: if (l ≤ heapSize(A)) and (A[l] > A[j])
05:     largest = l
06: else
07:     largest = j
08: if (r ≤ heapSize(A)) and (A[r] > A[largest])
09:     largest = r
10: if (largest <> j)
11:     swap(A[j], A[largest])
12:     Heapify(A, largest)
```

- line 4-8 find the largest key
- among parent and all its child
- if parent node does not have
- the largest key, then swap

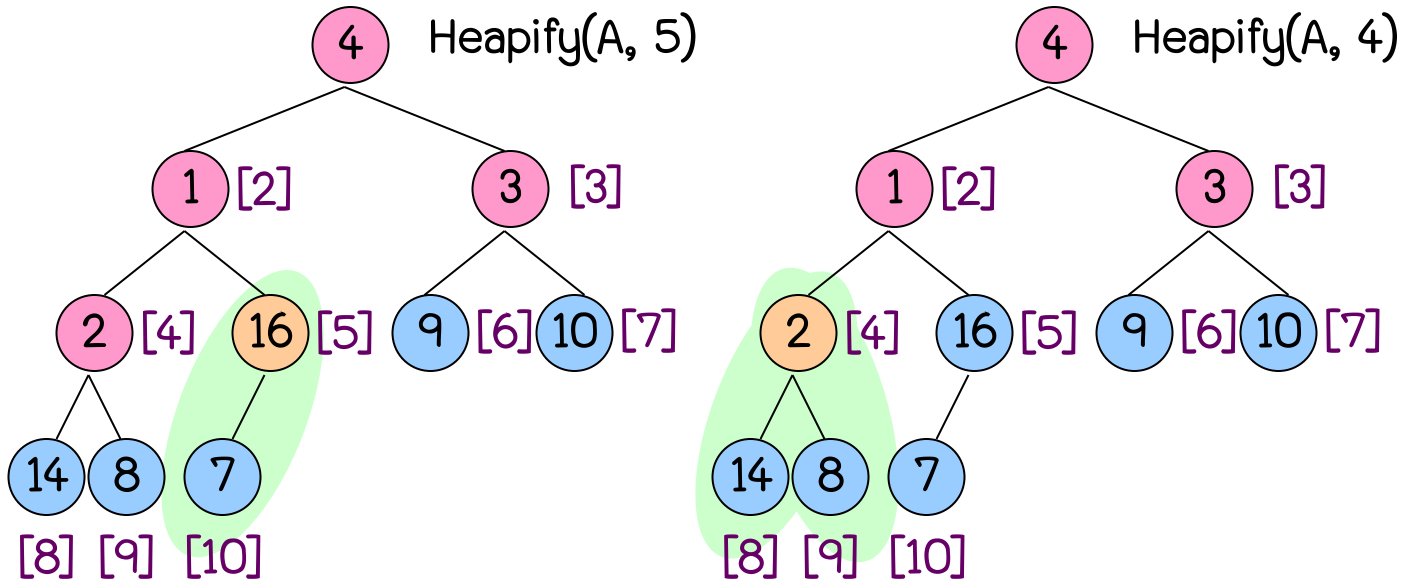
Build heap

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
4	1	3	2	16	9	10	14	8	7



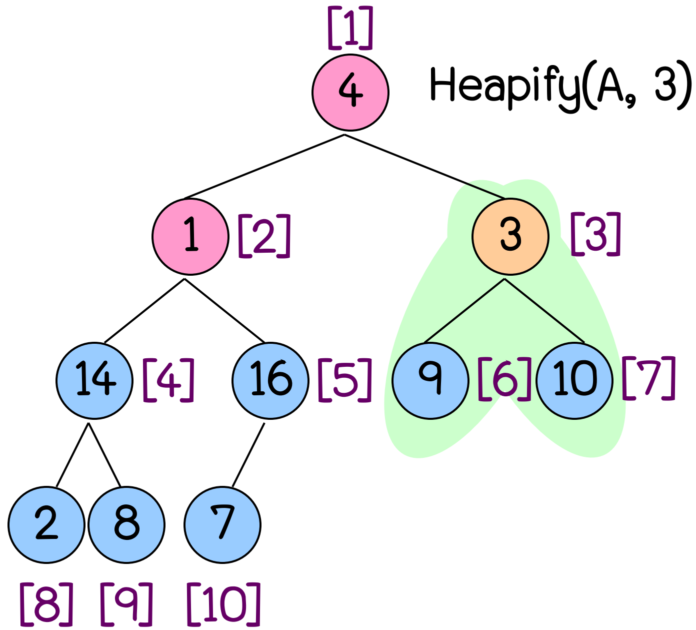
Leaves node ทุก node จัดเป็น heap ที่มีขนาด 1

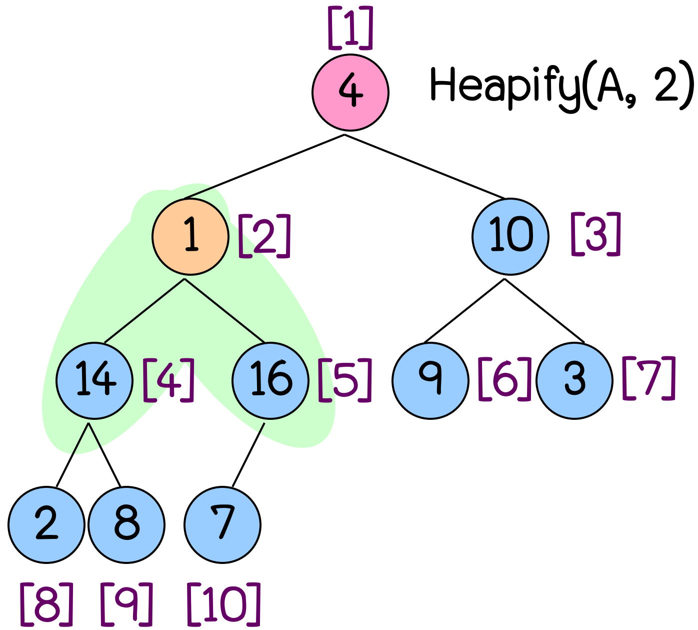
Build heap



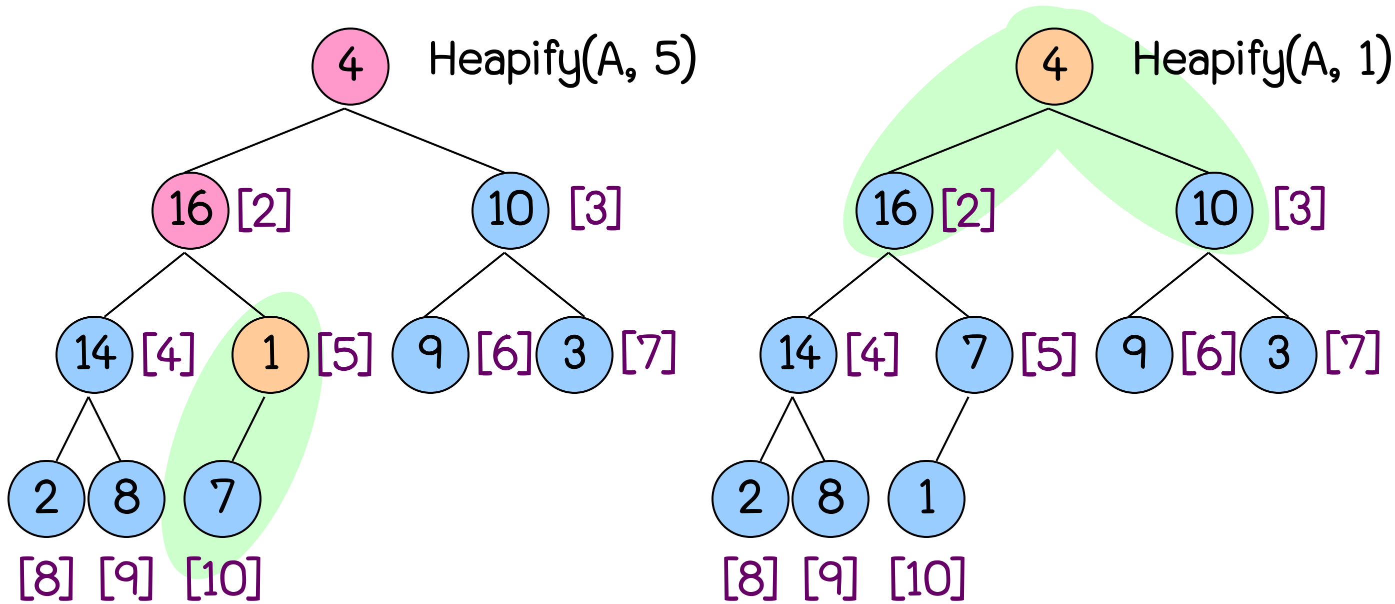
4
Heapify(A, 5)
1
[2]
3
[3]
2
[4]
16
[5]
9
[6]
10
[7]
14
8
7
[8]
[9]
[10]
4
Heapify(A, 4)
1
[2]
3
[3]
2
[4]
16
[5]
9
[6]
10
[7]
14
8
7
[8]
[9]
[10]

Build heap

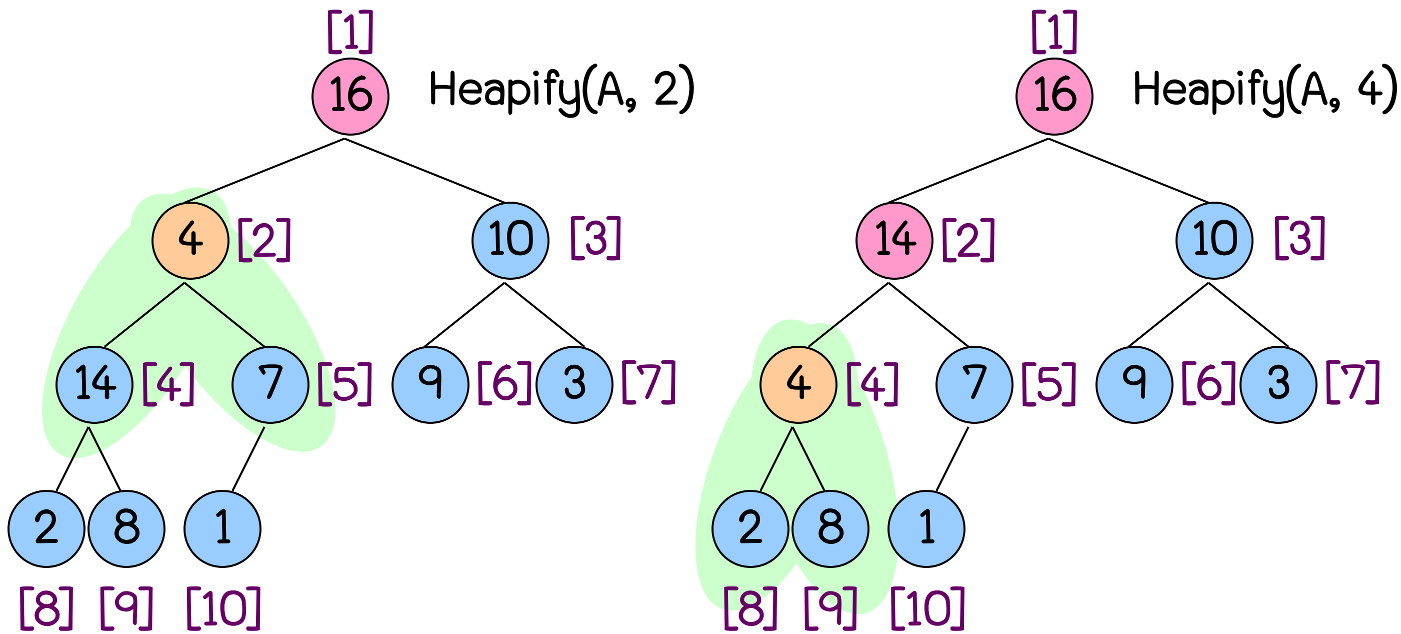

[1]
4
Heapify(A, 3)
1
[2]
3
[3]
14
[4]
16
[5]
9
[6]
10
[7]
2
8
7
[8]
[9]
[10]


[1]
4
Heapify(A, 2)
1
[2]
10
[3]
14
[4]
16
[5]
9
[6]
3
[7]
2
8
7
[8]
[9]
[10]

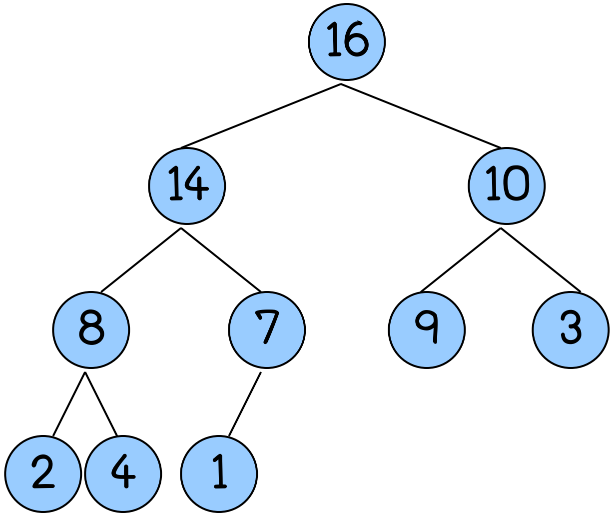
Build heap


4
Heapify(A, 5)
16
[2]
10
[3]
14
[4]
1
[5]
9
[6]
3
[7]
2
8
7
[8]
[9]
[10]
4
Heapify(A, 1)
16
[2]
10
[3]
14
[4]
7
[5]
9
[6]
3
[7]
2
8
1
[8]
[9]
[10]

Build heap


[1]
16
Heapify(A, 2)
4
[2]
10
[3]
14
[4]
7
[5]
9
[6]
3
[7]
2
8
1
[8]
[9]
[10]
[1]
16
Heapify(A, 4)
14
[2]
10
[3]
4
[4]
7
[5]
9
[6]
3
[7]
2
8
1
[8]
[9]
[10]

Build heap



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
16	14	10	8	7	9	3	2	4	1

Root node จะมีค่าคีย์สูงที่สุด

Heap sort

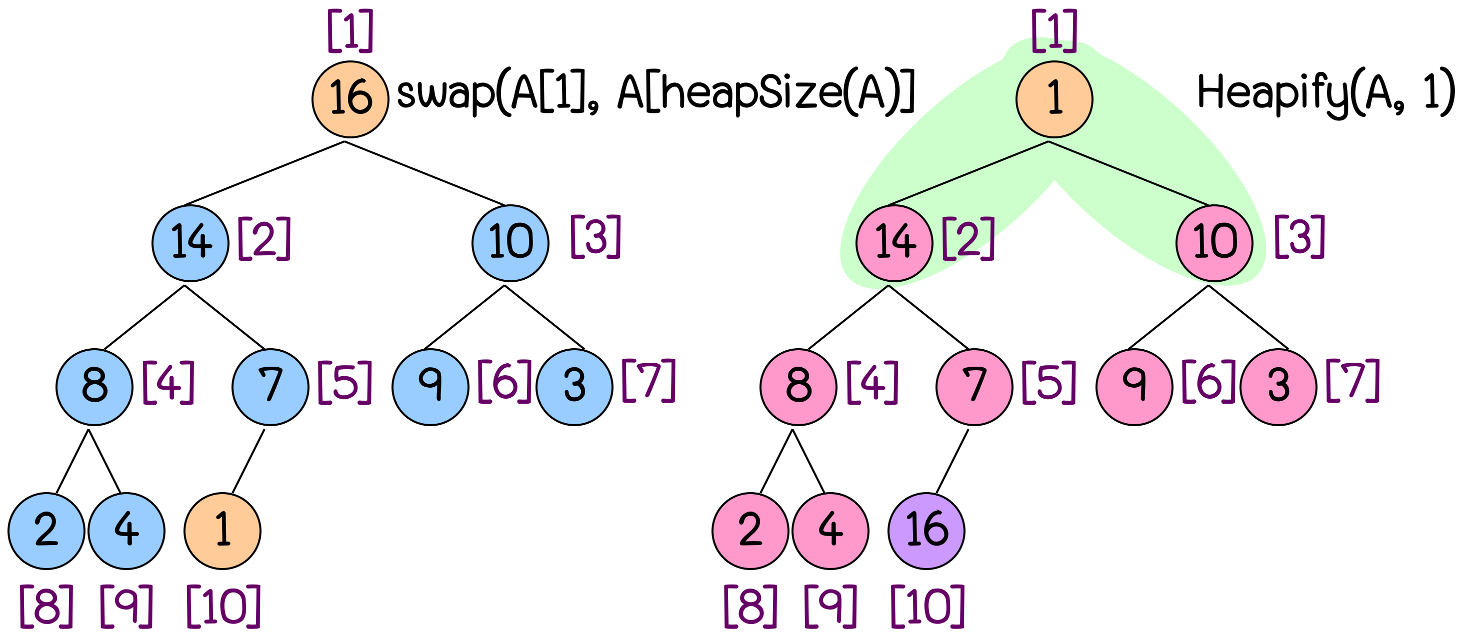
- ขั้นตอนวิธีของ Heap sort มีดังนี้
  1. BuildHeap บนข้อมูลทั้งหมดใน array ที่ต้องการเรียงลำดับ
     - root node เก็บข้อมูลที่มีค่าสูงที่สุด
  2. สลับที่ข้อมูลที่มีค่ามากที่สุดไว้ในตำแหน่งที่ถูกต้อง
     - swap(A[1], A[heapSize(A)])
     - การสลับที่มีผลให้ A[1..heapSize(A)-1] ขาดคุณสมบัติของ heap
  3. ทำ Heapify ที่ตำแหน่งแรก
     - Heapify(A, 1)

Heap sort

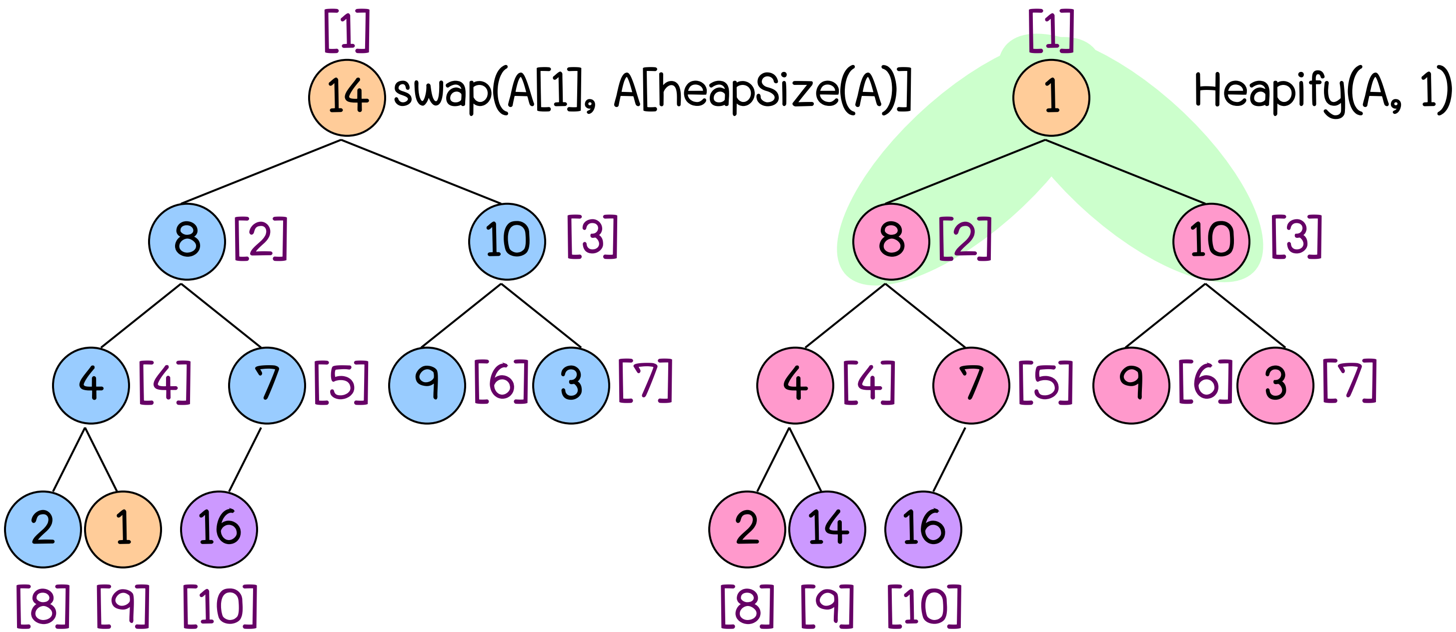

```
HeapSort(A):
BuildHeap(A)
for (j = length(A); j==2; j--)
    swap(A[1], A[j])
    heapSize = heapSize - 1
    heapify(A, 1)
```

- heapSize = length(A)

Heap sort

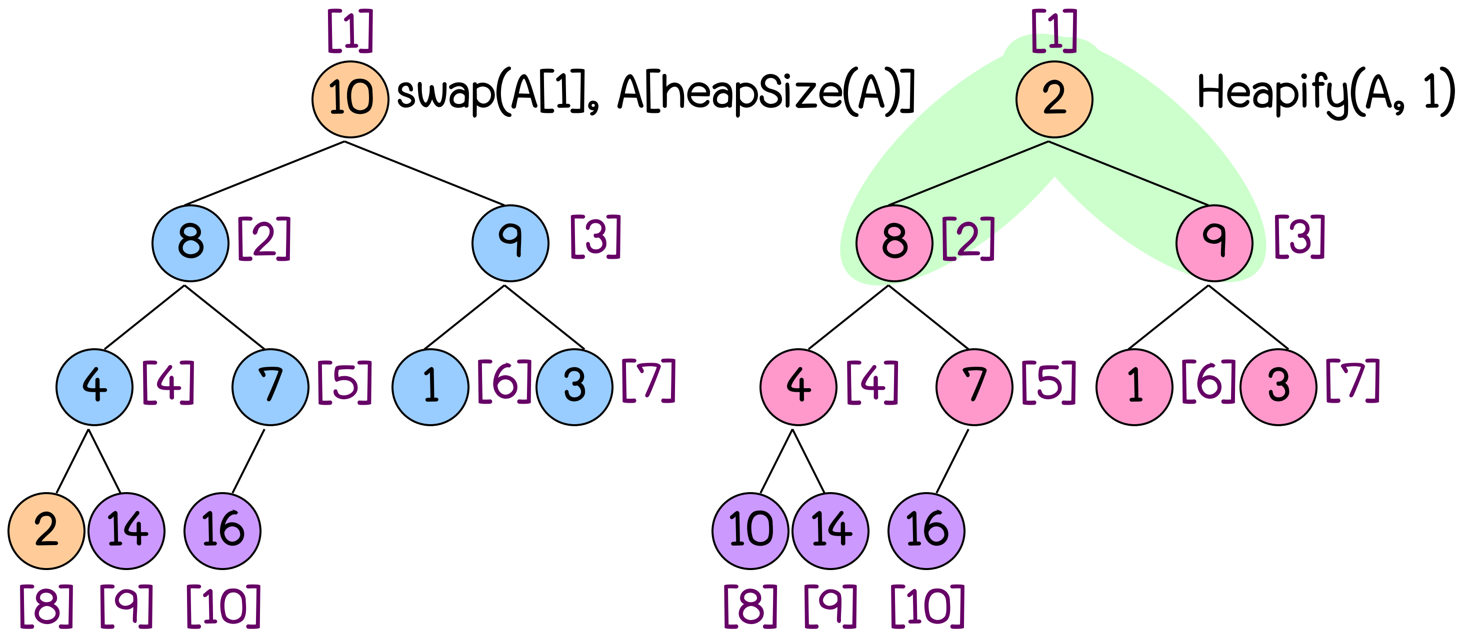

[1]
16
swap(A[1], A[heapSize(A)]
14
[2]
10
[3]
8
[4]
7
[5]
9
[6]
3
[7]
2
4
1
[8]
[9]
[10]
[1]
1
Heapify(A, 1)
14
[2]
10
[3]
8
[4]
7
[5]
9
[6]
3
[7]
2
4
16
[8]
[9]
[10]

Heap sort

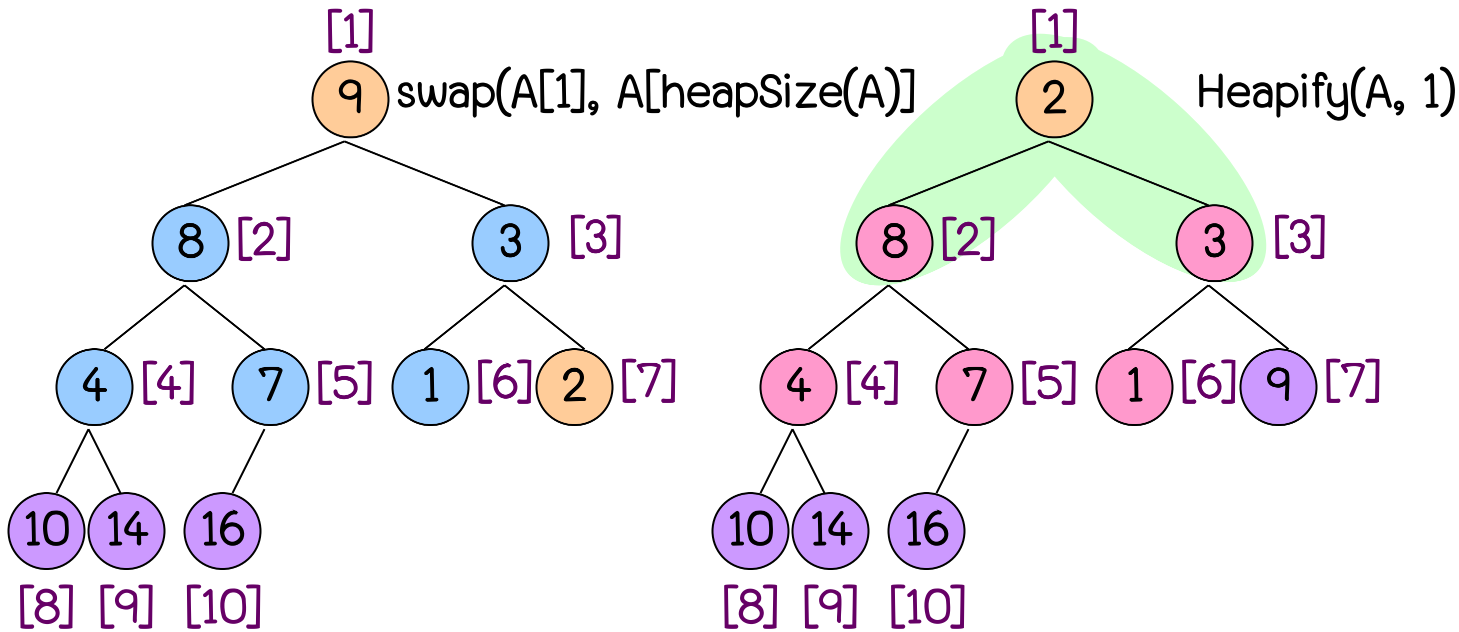


[1]
14
swap(A[1], A[heapSize(A)]]
8 [2]
10 [3]
4 [4]
7 [5]
9 [6]
3 [7]
2
1
16
[8] [9] [10]
[1]
1
Heapify(A, 1)
8 [2]
10 [3]
4 [4]
7 [5]
9 [6]
3 [7]
2
14
16
[8] [9] [10]

Heap sort

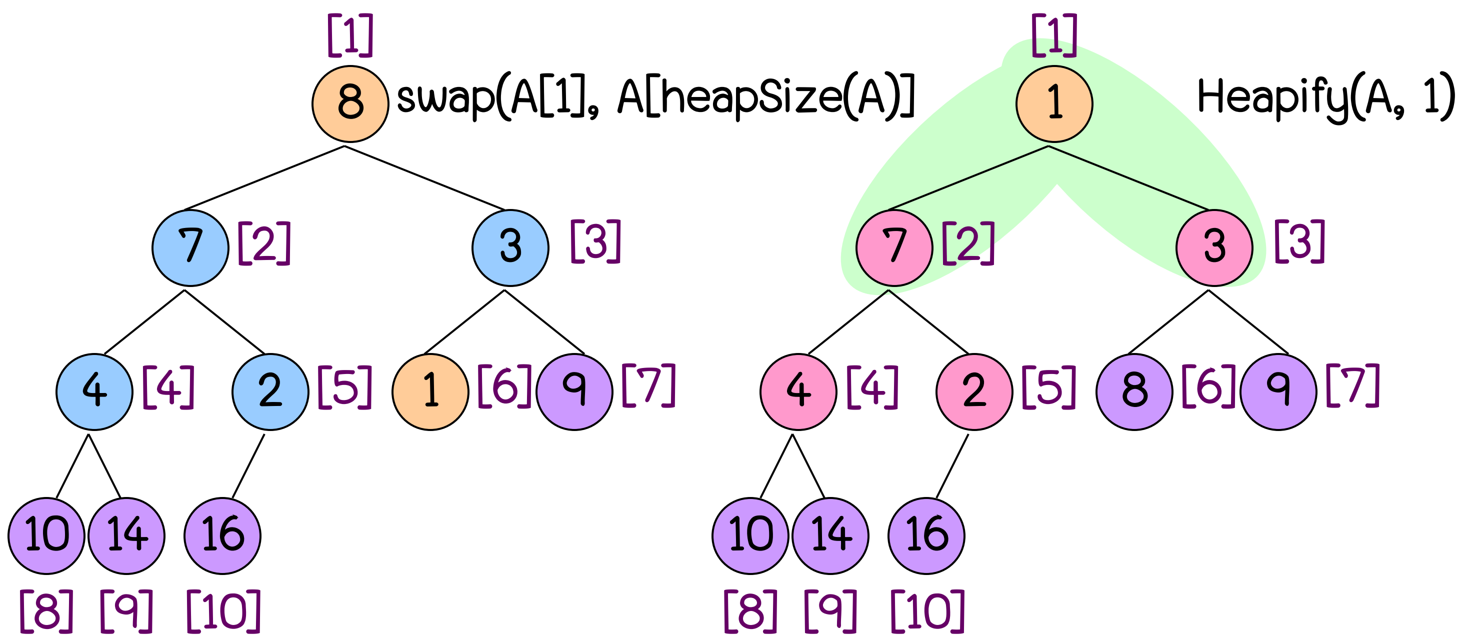

[1]
10
swap(A[1], A[heapSize(A)]
8
[2]
9
[3]
4
[4]
7
[5]
1
[6]
3
[7]
2
14
16
[8]
[9]
[10]
[1]
2
Heapify(A, 1)
8
[2]
9
[3]
4
[4]
7
[5]
1
[6]
3
[7]
10
14
16
[8]
[9]
[10]

Heap sort

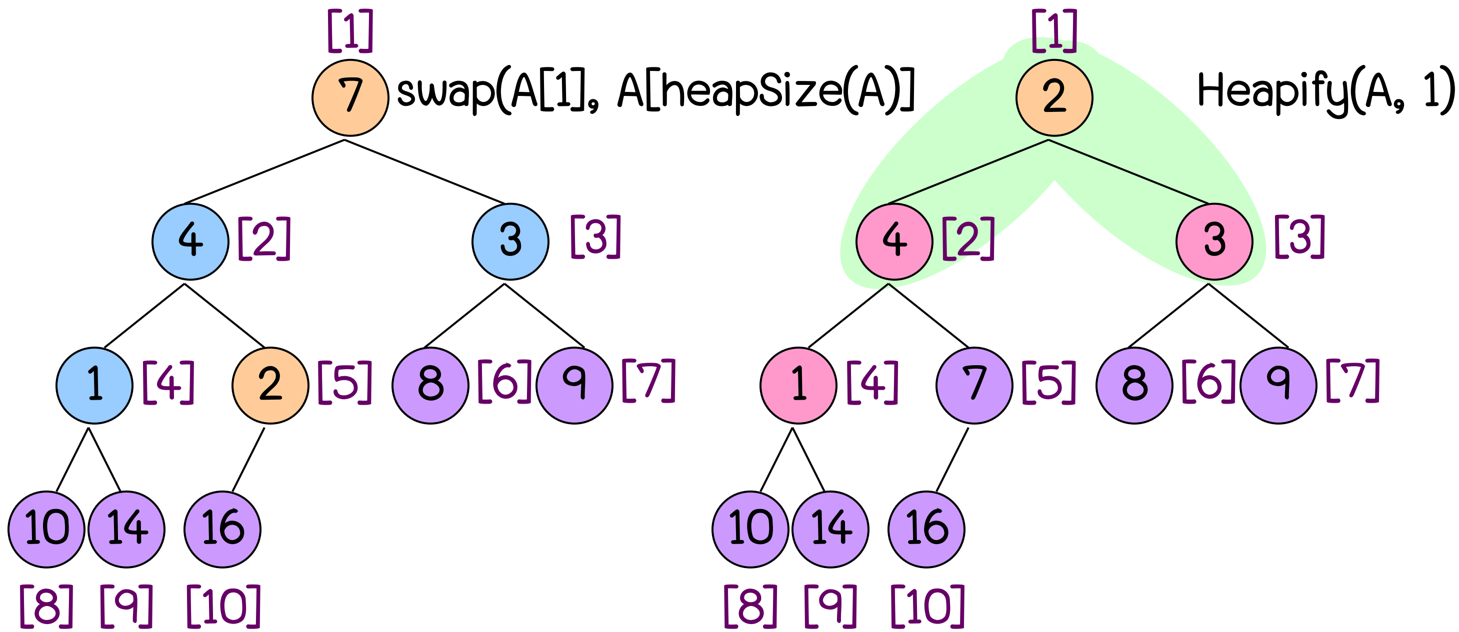


[1]
9
swap(A[1], A[heapSize(A)]
8
[2]
3
[3]
4
[4]
7
[5]
1
[6]
2
[7]
10
14
16
[8]
[9]
[10]
[1]
2
Heapify(A, 1)
8
[2]
3
[3]
4
[4]
7
[5]
1
[6]
9
[7]
10
14
16
[8]
[9]
[10]

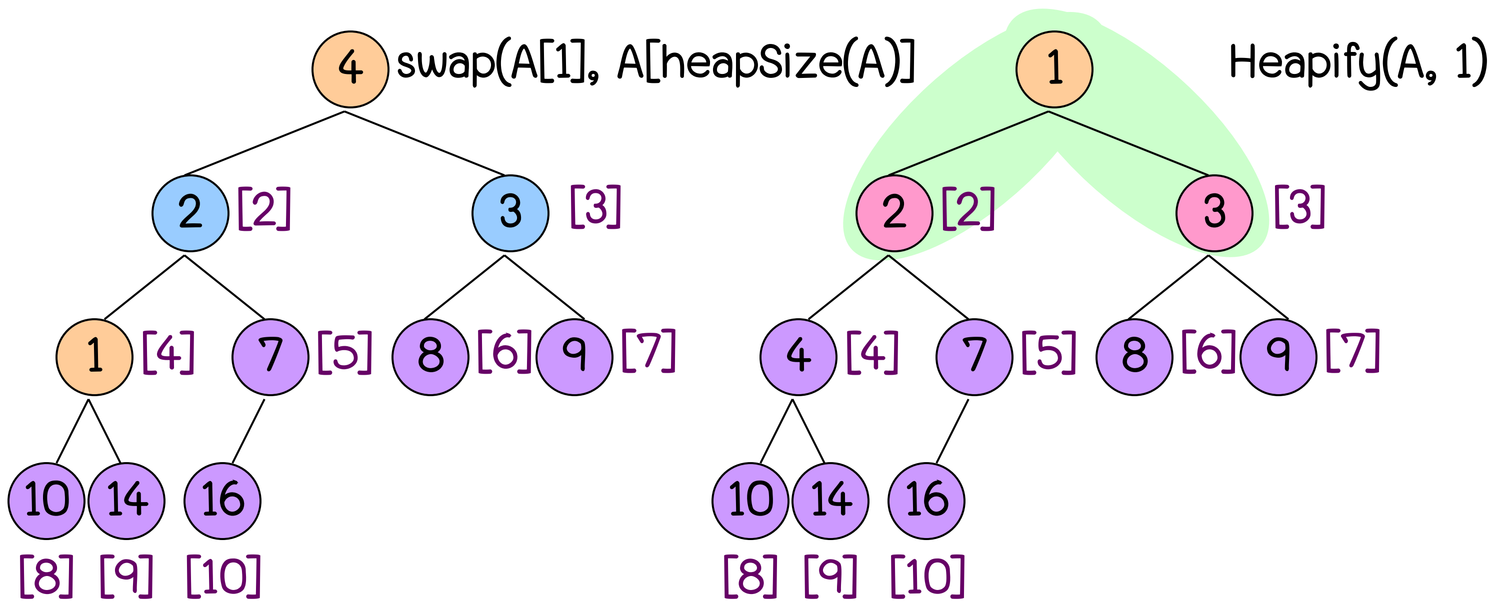
Heap sort


[1]
8
swap(A[1], A[heapSize(A)]
7
[2]
3
[3]
4
[4]
2
[5]
1
[6]
9
[7]
10
14
16
[8]
[9]
[10]
[1]
1
Heapify(A, 1)
7
[2]
3
[3]
4
[4]
2
[5]
8
[6]
9
[7]
10
14
16
[8]
[9]
[10]

Heap sort

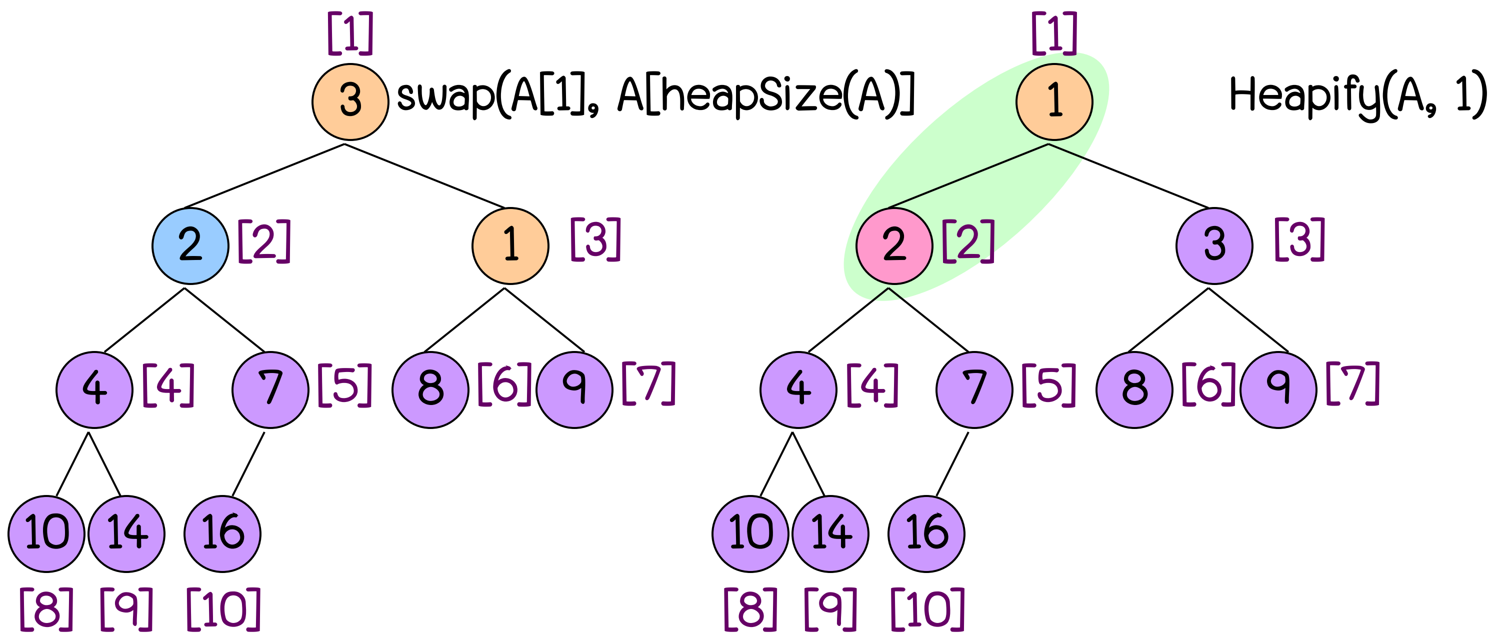

[1]
7
swap(A[1], A[heapSize(A)]
4
[2]
3
[3]
1
[4]
2
[5]
8
[6]
9
[7]
10
14
16
[8]
[9]
[10]
[1]
2
Heapify(A, 1)
4
[2]
3
[3]
1
[4]
7
[5]
8
[6]
9
[7]
10
14
16
[8]
[9]
[10]

Heap sort



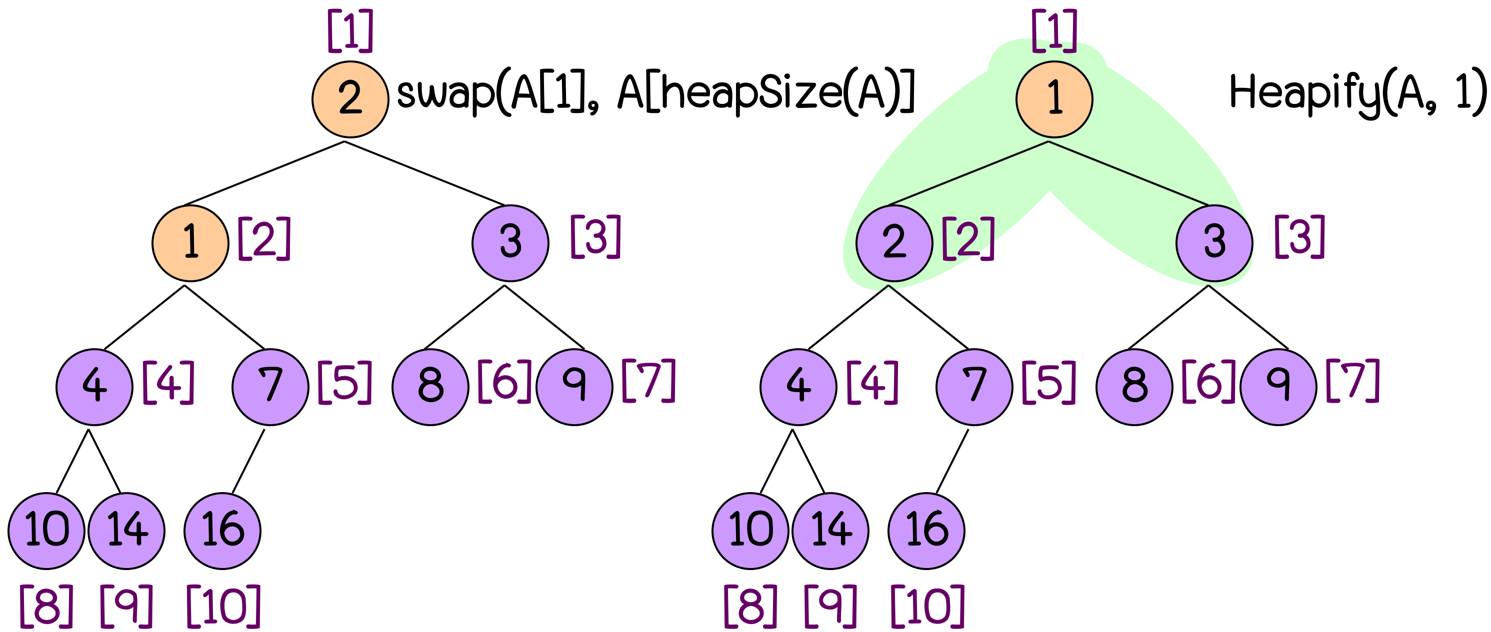
4
swap(A[1], A[heapSize(A)]
2
[2]
3
[3]
1
[4]
7
[5]
8
[6]
9
[7]
10
14
16
[8]
[9]
[10]
1
Heapify(A, 1)
2
[2]
3
[3]
4
[4]
7
[5]
8
[6]
9
[7]
10
14
16
[8]
[9]
[10]

Heap sort

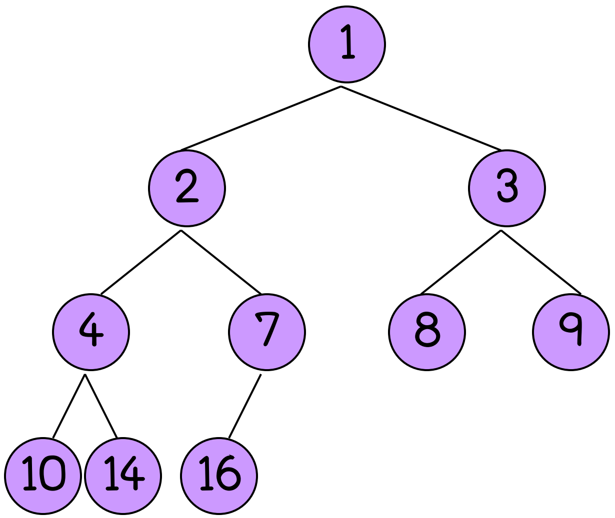


[1]
3
swap(A[1], A[heapSize(A)]
2 [2]
1 [3]
4 [4]
7 [5]
8 [6]
9 [7]
10 14 16
[8] [9] [10]
[1]
1
Heapify(A, 1)
2 [2]
3 [3]
4 [4]
7 [5]
8 [6]
9 [7]
10 14 16
[8] [9] [10]

Heap sort


[1]
2
swap(A[1], A[heapSize(A)]
1
[2]
3
[3]
4
[4]
7
[5]
8
[6]
9
[7]
10
14
16
[8]
[9]
[10]
[1]
1
Heapify(A, 1)
2
[2]
3
[3]
4
[4]
7
[5]
8
[6]
9
[7]
10
14
16
[8]
[9]
[10]

Heap sort



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
1	2	3	4	7	8	9	10	14	16

Heap sort

- เวลาที่ใช้ในการทำ Heap sort ขึ้นอยู่กับเวลาที่ใช้ในขั้นตอนการ BuildHeap และการทำ Heapify
  - เวลาที่ใช้ BuildHeap มีค่าเป็น O(n) เมื่อ n เป็นขนาดของข้อมูลทั้งหมด
  - ในแต่ละรอบการทำงาน จะต้องทำ Heapify ซึ่ง heap จะมีความสูงที่สุดไม่เกิน O(lg n) และทำงานทั้งหมด n-1 รอบ คิดเป็น
  - O(n)*O(lg n) = O(n lg n) = O(n log n)

  ดังนั้น เวลาที่ใช้ในการทำ Heap sort คิดเป็น O(n)+O(n lg n) = O(n log n)
- Heap sort มีประสิทธิภาพใกล้เคียงกับ Quick sort ในกรณีที่ลักษณะข้อมูลเริ่มต้นมีการเรียงลำดับแบบสุ่ม
- Heap sort มีประสิทธิภาพสูงกว่า worst case ของ Quick sort
- Heap sort ไม่มีประสิทธิภาพนัก เมื่อข้อมูลมีขนาดเล็ก เนื่องจากต้องใช้เวลาในขั้นตอนการสร้าง Heap และการเข้าถึงตำแหน่งของ parent และ child ในระหว่างการทำงาน

Shell sort

- Shell sort ประกอบด้วย 3 ขั้นตอน ดังนี้

1. แบ่งข้อมูลออกเป็นส่วนย่อย ข้อมูลย่อยแต่ละส่วนประกอบด้วยสมาชิกในทุกตำแหน่งที่ k ของข้อมูลเริ่มต้น และเรียก k ว่า *increment*

   สมมติ ข้อมูลจัดเก็บใน array A และกำหนดให้ k มีค่าเท่ากับ 5

   ดังนั้น array A จะถูกแบ่งออกเป็น 5 ส่วนย่อย ดังนี้

   ส่วนย่อยที่ 1 ประกอบด้วยสมาชิก A[1], A[6], A[11], ...
   ส่วนย่อยที่ 2 ประกอบด้วยสมาชิก A[2], A[7], A[12], ...
   ส่วนย่อยที่ 3 ประกอบด้วยสมาชิก A[3], A[8], A[13], ...
   ส่วนย่อยที่ 4 ประกอบด้วยสมาชิก A[4], A[9], A[14], ...
   ส่วนย่อยที่ 5 ประกอบด้วยสมาชิก A[5], A[10], A[15], ...

2. เรียงลำดับข้อมูลในแต่ละส่วนย่อย

Shell sort

3. เลือก k ใหม่ที่มีค่าลดลงจากเดิม แล้วทำซ้ำขั้นตอนที่ 1 และ 2

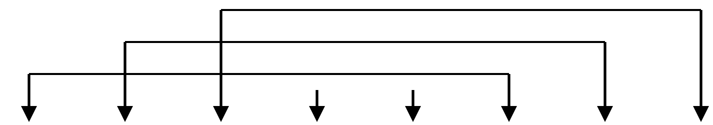
- ส่วนข้อมูลย่อยจะมีขนาดเพิ่มขึ้น
- ในที่สุด k จะมีค่าเป็น 1 และจะได้ข้อมูลที่เรียงลำดับแล้ว

สำหรับเทคนิคการเรียงลำดับข้อมูลในแต่ละส่วนข้อมูลอาจใช้ Insertion sort หรือเทคนิคอื่นๆ

Shell sort

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	57	48	37	12	92	86	33

และ k มีค่าเป็น (5, 3, 1)



<u>รอบ 1</u> k = 5

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	57	48	37	12	92	86	33

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
25	57	33	37	12	92	86	48

Shell sort

[1] [2] [3] [4] [5] [6] [7] [8] **<u>รอบ 2</u>** k = 3

25 57 33 37 12 92 86 48

[1] [2] [3] [4] [5] [6] [7] [8]

25 12 33 37 48 92 86 57

[1] [2] [3] [4] [5] [6] [7] [8] **<u>รอบ 3</u>** k = 1

25 57 33 37 12 92 86 48

[1] [2] [3] [4] [5] [6] [7] [8]

25 12 33 37 48 92 86 57

Shell sort

- ในรอบแรกๆ ค่า k มีค่าสูง ข้อมูลย่อยแต่ละส่วนมีขนาดเล็ก ทำให้ insertion sort สามารถทำงานได้อย่างรวดเร็ว
  ในรอบหลังๆ ข้อมูลย่อยแต่ละส่วนมีลำดับที่ใกล้เคียงกับลำดับที่ถูกต้องมากขึ้น ทำให้ insertion sort สามารถทำงานได้อย่างมีประสิทธิภาพมากขึ้น

  ดังนั้น shell sort จึงสามารถทำงานได้อย่างมีประสิทธิภาพ

- ลำดับของค่า k ที่เหมาะสมควรจะคุณสมบัติเป็น relative prime number
  เลขจำนวนเต็ม 2 จำนวน, a และ b จะเป็น relative prime ถ้าตัวหารร่วมมาก (great common divisor) ของ a และ b มีค่าเป็น 1 นั่นคือ gcd(a, b) = 1
  เช่น 8 และ 15 เป็น relative prime number