SCCS 321

Data Structure & Algorithm*

โครงสร้างข้อมูล และ อัลกอริทึม　ダーター ストラクチャー と アルゴリソム。

by:

Prologue.

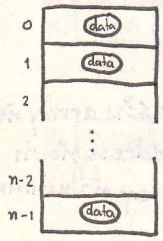
Data Structure ฯ เป็นวิชาที่สอนเกี่ยวกับการ เขียนโปรแกรม + จัดการกับ ข้อมูล ให้ มี "ประสิทธิภาพ" มากที่สุด ซึ่งในที่นี้จะใช้ภาษา Java ... (ลืมแล้ว!? ไปทวนมาซะ:) (เคยเรียนแล้วน่ะ:ว่าอย่าลืม) เป็นตัว main ในการเขียน...

"ประสิทธิภาพ"... ที่ผ่านมาเวลาเราเขียนโปรแกรมกัน เราสนแต่ ผลลัพท์ ที่ออกมา ชิมิ? แต่สังเกตรึป่าว่า บางคน ในโปรแกรมหนึ่ง เขียนไป 60 บรรทัด แต่บางคนใช้แค่ 20 บรรทัด, บางคนใช้ loop ซ้อนกัน 3-4 ชั้นตบท้ายด้วย if อีกมากมาย แต่บางคนใช้ loop ตัวเดียวก็จบไรเงี้ย... แต่ในวิชานี้ เราต้อง ทั้งเขียนโปรแกรมให้ได้ พร้อมกับใช้บรรทัดน้อยลง, โปรแกรม ทำงานเร็วขึ้น, ใช้หน่วยความจำน้อยลง (การประกาศ variable มากไปก็ทำให้โปรแกรมกิน RAM น:)

"Algorithm"... คือ "การคิดแบบเป็นลำดับ" ต้องรู้ก่อนน:ว่า com.ฯ มันคิดได้ ทีละ:อย่าง แต่ที่มันคิดได้เร็วกว่าคนก็แค่ มันคิดเร็ว, การคิดเป็นลำดับเช่น 5+7*2 ... ถ้าเป็นคนคิด ก็จะเอา 7*2 ก่อนแล้วค่อย +5 ได้ 19 แต่ถ้า com.ฯ คิด มันต้องหาก่อนว่า operater ไหน ต้องทำก่อน แล้วก็ดูว่าตัวเลขที่ขนาบข้าง คือเลขอะไรบ้าง แล้วไปจอง memory เก็บคำตอบของ operater นั้นก่อน แล้วค่อย คำนวณ ได้ผลลัพท์ ก็เอาไปเก็บไว้ในช่อง memory ที่จองไว้ แล้วค่อยเอาไปคิดกับ +5 ... เหนื่อย!! แต่ที่เราเห็นว่า com.ฯ มันคิดได้ดีกว่าเรา เพราะมันทำขั้นตอนได้เร็ว! แต่สังเกตว่ามันทำได้ ทีละ:อย่างเท่านั้น algorithm ก็เหมือนกับการสั่ง com.ฯ ให้ทำงานเป็นขั้นๆ นั่นแหละ:

Array.

- สิ่งที่ใช้เก็บ data ได้อย่างมี "ประสิทธิภาพ" ตัวนึงก็คือ array... เวลาคิดถึง array ให้นึกถึง memory ที่ถูกจองพื้นที่ใช้งานเรียงติดกัน n ช่อง (ปกติเวลาเราประกาศ var. (declare) โปรแกรมจะ:จองพื้นที่ให้แบบโดดไปที่ address โน้นที กลับมาทางนี้ที แต่สำหรับ array ช่องพวกนั้นจะเป็นช่องที่มี address ของ mem. ต่อกัน)



index เป็นตัวบอกว่าเราอ้างอิงถึงช่องเก็บของช่องไหน... ถ้า array n ช่อง จะมี index ตั้งแต่ 0 ถึง n-1 ※ใน Java... ถ้าเราอ้างถึง index ตั้งแต่ช่องที่ n เป็นต้นไปจะเกิด Exception น:

length คือความยาว หรือก็คือจำนวนช่องทั้งหมดของ array ตัวนั้น ※ใน Java... เราสามารถขอ length ของ array "A" ได้โดย A.length, ไม่มี () ต่อจากคำว่า length น: ไม่ใช่ method!

Declaring & Reference

การประกาศ var. แบบ array (พูดอีกแล้ว ขี้ไปขี้มา — —*) โดยใช้ Java ทำได้โดย

จำนวนช่อง ของ array

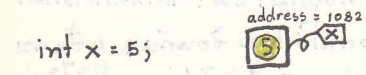
```
int[] name = new int[length];
```

- `int[] name` → สร้าง var. ชื่อ name ซึ่งระบุว่าจะให้เก็บ int ที่เป็น array
- `new int[length]` → สร้างช่องเก็บ (ของ array) จำนวน length ช่อง (สังเกต : มีคำสั่ง "new" แสดงว่ามีการสร้างอะไรบางอย่าง)

- การทำงานของ [=] มันจะทำทางขวาก่อน แล้วโยนค่าของทางขวาให้ทางซ้าย
  - สร้างช่องarray
  - name เก็บค่าช่องที่เพิ่งสร้างเมื่อกี้

อธิบายเรื่อง Reference กันหน่อย

ปกติเวลาเราสร้าง variable มาเก็บค่า ตัวcompiler จะเป็นตัวจัดการเรื่องการจองพื้นที่ใน memory ให้โดยที่เราไม่ต้องทำเอง

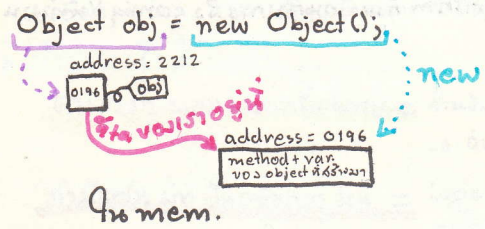


◁ เช่นถ้าเราประกาศ int มาตัวนึงให้ชื่อว่า "x"... compiler จะไปจองพื้นที่ใน mem. ให้ 32 bit (ขนาดของ int ใน Java) ซึ่งจะจองที่ address ไหนก็ไม่รู้ (ในตัวอย่างที่ address 1082) compiler จะทำให้เองว่าถ้าเรามีการพูดถึง x อีกให้ไปยุ่งกับ address 1082... เมื่อเราบอกว่าให้ x มีค่า 5 มันก็จะไป set ค่าใน address 1082 ให้เท่ากับ 5 เก็ทมั๊ยเนี่ย..?

ตัวแปรแบบนี้ (ที่เก็บ data ของมันไว้ที่ address ของตัวมันเองเลย) เราเรียกว่า **Primitive data type** หรือก็คือตัวแปรแบบ basic ๆ มาตรฐานมากมาย (พิมพ์แล้วเป็นสีน้ำเงิน, ใน EditPlus นะ:)

? ที่นี้... คงคิดว่า var. มันก็ต้องเก็บ data ของมันไว้ที่ตัวมันสิ ไม่งั้นจะเก็บที่ไหน

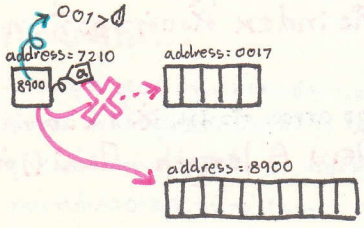
▷ ก็มันมี var. ประเภท Reference ที่ data ของตัวมันเองไม่ได้เก็บอยู่ที่ตัวมัน



◁ ส่วนมาก Reference มักจะใช้กับ obj. (obj. มีรายละเอียดของ method และ var. เยอะเกินไปที่จะเก็บไว้ที่ตัวมันเอง)... การสั่ง new คือมันจะไปสร้าง Object ที่ตำแหน่งหนึ่งใน mem. เสร็จที่นี่ เราก็สร้าง var. ชื่อ "obj." ซึ่งเก็บค่าเป็น object ที่เพิ่งสร้างไป... แต่มันเอา Object ที่เพิ่งสร้าง (เพิ่ง new ไป) มาเก็บไว้ที่ตัวมันไม่ได้ มันเลยเก็บ "ที่อยู่" ก็คือ address แทน

ที่นี้... ลองสังเกตดูว่าเวลาเราสร้าง array เราต้องสั่ง new เหมือนกัน ก็แปลว่า array (เฉพาะใน Java) เป็น var. แบบ reference

```
① int[] a = new int[5];
② a = new int[10];
```



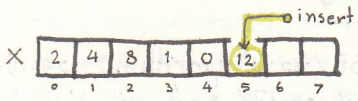
◁ ข้อดีของ array แบบ obj. (reference) ก็คือ เราสร้าง array ตัวใหม่ได้โดยการ new array ตัวใหม่ขึ้นมา... address ตัวเก่าก็จะถูกโยนทิ้งออกไป แล้วเก็บ address ของ array ตัวใหม่แทน

Note Java จะทำการล้าง mem. ให้ system เมื่อ obj. (หรือ array) ตัวนั้นไม่ถูก reference จาก var. ตัวไหนเลย... เช่นในตัวอย่าง เรา สั่ง a ให้เท่ากับ array ตัวใหม่ mem. ที่ 0017 ก็จะถูกล้างทิ้ง!

Insert

การ insert data ตัวใหม่เข้าไปใน array มี 2 แบบ คือ insert เข้าไปที่ตัวสุดท้าย กับ insert เข้าไปตรงกลางของแถว array (ตรงที่มี data อยู่แล้ว)...

※ การ insert ไม่ต้องการการเปรียบเทียบ (comparisons)

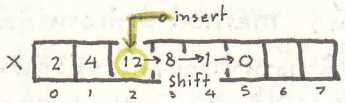


insert ตัวท้าย

→ ขั้นยัดเข้าที่แหน่ง (index) ท้ายสุด... จบ ง่ายๆ

- move : 1

insert new data



insert ตรงกลาง

→ ก่อนจะยัดได้ เราต้องเลื่อน data ตัวอื่นให้เขยิบไป 1 index แล้วค่อย แทรก data ตัวใหม่เข้าไป

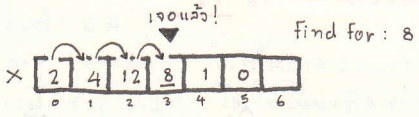
- move : $1-n+1$ ← แทรก data 1 ที่ — insert new data

จำนวน data ใน array ตอนแรก — index ที่จะแทรก data ใหม่เข้าไป — เลื่อน data 1 index

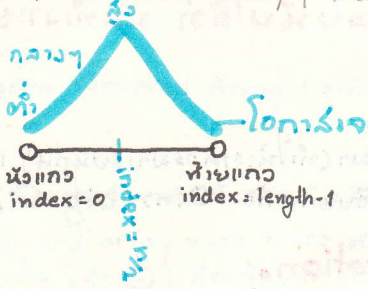
Find

การ find data ที่อยู่ใน array ก็เพื่อต้องการรู้ว่า data ตัวนั้นอยู่ที่ index ที่เท่าไหร่ (ถ้าไม่เจอ return -1) ซึ่งก็มี 2 แบบ อีกแหะ: คือ find ใน array ที่ยังไม่เรียงกับเรียงแล้ว (Unordered, Ordered array)

※ สำหรับ Ordered array เราจะใช้ 'algorithm' ชื่อว่า binary search (เคยเขียนแล้ว เลยจะไม่พูดซ้ำ) กลับไปอ่าน binary ฯ link เสียก็ขี้เกียจใหม่!



- move : none!
- comparisons : $n/2$ (avg.) โดยเฉลี่ยนะ

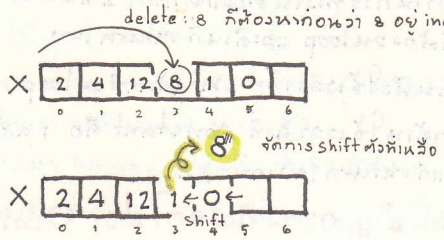


การ find เราก็ต้องไล่หาไปทีละตัว จนกว่าจะเจอ โดยส่วนใหญ่แล้ว เรามักจะเจอแถวๆ $n/2$ (ตรงกลางของ array)

find แค่ต้องการ compare ว่าที่ index นั้นเท่ากับตัวที่เราหาอยู่รึเปล่า จึงไม่ต้องมีการขยับ data ที่อยู่ใน array เลย (แค่ขอ data มา check)

Delete

การ delete มันจะตรงกันข้ามกับ insert ∴ มี 2 แบบ คือ data ที่จะ delete อยู่ตัวสุดท้าย กับ อยู่ตรงกลาง แต่ก่อนอื่น... ถ้าต้องการ delete data เราก็ต้องหาก่อนว่า data ตัวนั้นอยู่ที่ index ไหน ซึ่งก็คือ Find



∴ การ delete ต้องใช้ทั้ง find และการ shift

- move : $1-n$ (จำนวน data ตอนแรก, ที่แหน่ง delete) — น้อยกว่า insert เพราะ shift อย่างเดียว
- comparisons : $n/2$ (avg.)

Asymptotic analysis

เป็นการ "analyse" (วิเคราะห์) method ว่าทำงานได้ดีขนาดไหน...

▷ การเขียน method ขึ้นมาตัวหนึ่งให้ทำงานบางอย่าง อาจมีวิธีเขียนได้หลายวิธี (หลาย algorithm) เวลาเขียนเราต้องเลือก algorithm ที่ดีที่สุดมาใช้ เช่น method sort array ก็มีหลายวิธี sort เช่น bubble, insert, redix ซึ่งมีความเร็วในการทำงานต่างกัน แต่ให้ผลเหมือนกันคือ sort array

```
public int test(int n)
{  int i, sum=0;
   for(i=0; i<n; i++)
      sum+=i;
   return sum;
}
```

วนลูป n ครั้ง

- method นี้รับค่า n มา... ถ้าจะนับว่า method ตัวนี้ ทำงานเร็วแค่ไหนก็คือ นับว่ามี statement ที่ต้องทำงานกี่ครั้ง
  - declare i กับ sum ... 1 ครั้ง
  - วนloop n ครั้ง บวกค่า sum เพิ่ม ... n ครั้ง
  - return ... 1 ครั้ง

∴ method ใช้เวลาทำงาน 2+n

แต่เวลาเราคิดจริงๆ... method นึงมันไม่ได้มี statement น้อยๆ แบบนี้เท่าไหร่ เราเลยจะนับแค่ statement **สำคัญ** ...โดยไม่สนค่าคงที่ (constant) method ข้างบนเราเลยนับได้ว่า มันทำงานใช้เวลาประมาณ n.

Symbol

ได้มีการคิด สัญลักษณ์ขึ้นมา (ทำให้เราต้องมาเรียนกัน) เพื่อบอกว่า method ใช้เวลาทำงานแบบไหน และใช้เวลานานขนาดไหน... ที่เรียนกันคือ O(n) (big O) แต่จะแถมอีก 2 ตัวที่ควรรู้ Ω กับ Θ

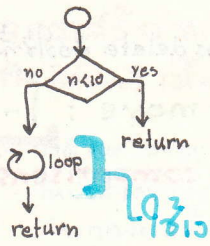
O(n) big O notation.

เราใช้ O(n) ก็ต่อเมื่อต้องการบอกว่า method นี้ ทำงานในกรณีช้าสุดเป็น n. (worst case!) เช่น... method ข้างบนนั้น จำเป็นต้องวนloop n ครั้ง ทุกรอบ อย่างน้อยมันต้องใช้เวลา n ซึ่งทำงานจบ O(n) ใช้เวลา n (ตัด constant ทิ้ง สนเฉพาะตัว variable)

method: o1

```
public int mto1(int n)
{  int i, sum=0;
   if(n<10) return sum;
   else
      for(i=0; i<n; i++)
         sum+=i;
   return sum;
}
```

- ทีนี้... method มีการใช้ if-else เลือกเส้นทาง ทำการทำงานของโปรแกรม จะเห็นว่า เวลาในการทำงานต้องแยกเป็น 2 แบบคือ ด้านที่ต้องวนloop และด้านที่ return เลย



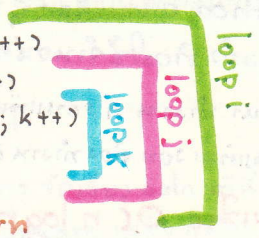
ใช้เวลา n

คือ ด้านนึงใช้เวลาเป็น n (ด้านที่วนloop)
อีกด้านใช้เวลาคงที่ constant คือ 1 ครั้ง (แค่ return เลยเองไง)...

∴ อย่างที่บอกไปว่า O(n) คือกรณีช้าสุด... เราจะนับว่า method ตัวนี้ทำงานเป็น n ไม่ใช่ 1 big O จึงเท่ากับ O(n) ไม่ใช่ O(1)

method: 02

```
public int mt02(int n)
{
    int i, j, k, sum=0;
    for(i=0; i<(n/2); i++)
        for(j=0; j<5; j++)
            for(k=0; k<n; k++)
                sum++;
    return sum;
}
```



- เจอ loop ซ้อนๆ กับแบบนี้ให้เอาจำนวนครั้งที่แต่ละ loop ต้องวนมา × (คูณ) กัน
  - loop k ใช้เวลา n ครั้ง
  - loop j ใช้เวลา 5 ครั้ง
  - loop i ใช้เวลา $n/2$ ครั้ง

∴ loop j วนทำ k 5 ครั้ง = $5k = 5 \cdot n$ ครั้ง
∴ loop i วนทำ j $n/2$ ครั้ง = $n/2 \cdot (5n) = \frac{5}{2} \cdot n$ ครั้ง

∴ ใช้เวลาทั้งหมด $\frac{5}{2}n^2$ ครั้ง แต่การคิด big O ต้องตัด constant กับ co-efficient (สัมประสิทธิ์) ทิ้งให้หมด big O คือ $O(n^2)$ (ตัด $\frac{5}{2}$ ที่เป็น co-efficient ทิ้ง)

method: 03

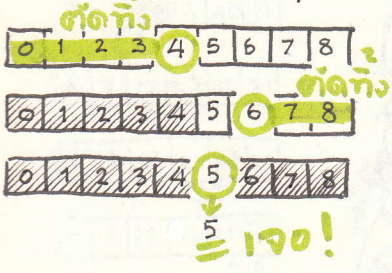
ใช้เวลา $2n^2 + 500n + \log n$
n ที่ลังมากที่สุด!

- เราจะหาว่า big O จาก method นี้ (ซึ่งที่ต้องว่าคิดมาแล้วว่าใช้เวลาเท่านั้น) ถ้าเป็นสมการยาวๆ ให้ดู term ที่ n มีกำลังมากที่สุด... เอา term นั้นมาตอบเป็น big O

∴ big O เป็น $O(n^2)$

method: 04

binary search. ...เป็นการ search หาของใน Ordered Array (array ที่ sort มาแล้ว) โดยมีวิธีคือไล่หาเป็นช่วงไปเรื่อยๆ ไม่ใช่ไล่หาทีละตัว



การใช้ binary search จะหาที่ "ตัวกลาง" หรือ index ที่อยู่กลางของช่วงที่หา เช่น ถ้าจะหา ⑤ เราจะมี array ขนาด 9 ช่อง ตัวกลางก็คือ index ที่ 4 ... ปรากฏว่า index ที่ 4 (คือ ④) มีค่าน้อยกว่าตัวที่เราหาอยู่ ก็แปลว่า index ที่ 0 ถึง 3 ต้องน้อยกว่าชัวร์ๆ เพราะ array นี้เรียงอยู่แล้ว... เราก็ไปสนใจครึ่งหลังของ array แทน ซึ่งก็ทำเหมือนเดิมคือ check ตัวกลางของช่วงนั้นแล้วถ้ายังหาไม่เจออีกก็ดูว่าเราควรจะสนด้านซ้าย หรือขวา ของช่วงนั้น ทำแบบนี้ไปเรื่อยๆ จนกว่าจะเจอตัวที่เราจะหา การใช้ binary search จะช่วยให้เราประหยัดเวลาในการหาได้มากมาย เช่น ถ้า array 1000 ช่อง การหาครั้งแรกก็ทำให้เราตัดช่องที่ไม่ต้องหาไปได้แล้วประมาณ 500 ช่อง!

ฉันใด big O จะเป็นอย่างไร?

worst case ของมันก็คือ "เจอตัวสุดท้าย!" หมายความว่าเราตัดช่วงที่ไม่ใช่ออกจนหมดๆ กว่าจะเจอตัวที่หา เช่นตัวอย่างข้างบนที่เราหา ⑤

- การหา binary search ถ้าเริ่มด้วย n รอบ 2 จะเหลือ $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}$ ... จนเป็น 1 (ตัวสุดท้าย)
- ลองคิดกลับดู... เริ่มที่ 1, รอบต่อไปเป็น 2, 4, 8, 16, ... n ก็คือ $2^x \approx n$ (โดยประมาณ)
- เราจะหาค่า x จาก $2^x \approx n$ เราก็เปลี่ยน expo → log [$b^x = n \rightarrow x = \log_b n$] จะได้ $x = \log_2 n$

∴ จะได้ว่า worst cast คือประมาณ $\log_2 n$ (แต่เขานิยมที่จะละ "base" ของ log เอาไว้)

∴ big O เป็น $O(\log n)$

note: $O(1) < O(\log n) < O(n) < O(n^{7/6}) < O(n \log n) < O(n^{3/2}) < O(n^2) \ldots$
$\ldots < O(n^3) < O(2^n)$

method : 05

```
for(i=1; i<n; i=i*2)
    for(j=0; j<n; j++)
        ...do something...
```

<ดู for ตัวนอกดีๆ นะ: ตอน update ค่า มันไม่ใช่ i++ แต่เป็น i=i*2

▶ คล้ายๆ กับ method 04... for i มันเพิ่มค่าทีละเท่าตัว อย่างที่เคยคิดไปแล้ว ต้องใช้จำนวนครั้ง คือ log n ...เท็คนะ:

∴ for j นั้นไม่มีปัญหา ทำงาน n รอบแน่นอน สรุปว่า for i ที่ต้องทำงาน log n รอบ ในแต่ละรอบ ต้องทำงานอีก n จะได้ว่า

∴ big O จะเป็น O(n log n)

Ω , Θ .

(ของแถม) ยังมี notation อีก 2 ตัวที่ควรรู้คือ Ω (omega) และ Θ (theta)
ที่ผ่านมาเราใช้ big O บอกว่า method นั้นทำงานแบบ "worst case" เป็นเท่าไหร่
- สำหรับ Ω เราใช้เพื่อบอก "best case" หรือกรณีที่ใช้เวลาน้อยที่สุดนั่นแหละ:
- สำหรับ Θ เราใช้เพื่อบอกว่า method นี้มี "ขอบบน กับ ขอบล่าง ที่เท่ากัน" พูดง่ายๆ คือไม่ว่า n จะมีค่าเป็นเท่าไหร่ มันก็ใช้เวลาตายตัว!

ตัวอย่างเช่น ใน method 01 จะมี Ω(1) และไม่มี Θ เพราะ: ขอบบน - ขอบล่าง ของการทำงานไม่เท่ากัน
※ คิด Θ ง่ายๆ ถ้า O(n) กับ Ω(n) เท่ากัน จะมี Θ(n)

---

basic Sorting.

DATA-TAN～

Sorting คือการทำให้ data ใน array เรียงกัน
โดยเรียงแบบ • น้อย → มาก (Ascending)
• มาก → น้อย (Decending)

※ส่วนใหญ่จะใช้ น้อย → มาก

- โดยการ sort นั้น จะต้องมี array มาเกี่ยวข้องด้วยแน่นอน
เมื่อมี array มาเกี่ยวเราก็แยกประเภทของ array ได้โดย
1. primitive array เช่น int[] x, char[] c
2. Object array เช่น String[] str

ข้อแตกต่างระหว่าง primitive ๆ กับ Obj. .

ถ้าเรามี int x = 4; int y = 4; x กับ y ก็ถือว่ามีค่า "เท่ากัน"
ในมุมมองของ array [4|4]... มีค่าเท่ากับ [4|4]... (คือสลับที่กันได้ แล้วแยกไม่ออก)
แต่สำหรับ Obj. สมมุติว่าเรากำลัง sort array of String โดยใช้ length เป็นตัววัด

[abc|xyz]... กับ [xyz|abc]... ถึง abc และ xyz จะมี length = 3 เท่ากัน แต่มันเป็น data คนละตัวชิ้น ไม่เหมือน primitive ที่เราแยกไม่ออก

∴ ดังนั้น เมื่อเรา sort Obj. เช่น 2, 1, 3, 4, ๓, ๖ ก็ควร sort ให้เป็น 1, 2, ๖, 3, ๓, 4 ไม่ใช่ - 1, ๖, 2, ๓, 3, 4 (๖ กับ ๓ อยู่หลัง 2 กับ 3 ... ถึงมันจะมีค่าเท่ากัน แต่ก็เป็นคนละตัว) หลังจาก sort แล้ว ถ้าเจอ data ที่เท่ากัน แต่เป็นคนละตัว ตัวไหนเคยอยู่หน้าก็ควรให้อยู่หน้าต่อไป ... การ sort แบบนี้ทำให้ array ของเรามีความเสถียร!? (stable sort)

Simple Sorting.

การ sort มีหลาย algorithm มากมาย sorting แต่ละอย่างก็เหมาะกับ data อย่างนึง แต่ตอนนี้จะพูดสอง "Simple Sorting" 3 ตัว (ซึ่งเขียนค่อนข้างง่าย แต่ทำงานเร็วสู้พวก advance sorting ไม่ได้)

- Bubble Sort
- Selection Sort
- insertion Sort

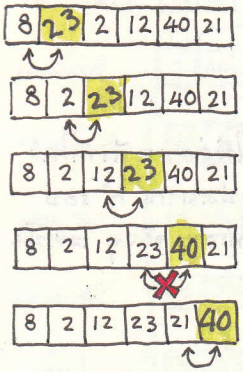
swap: การ sort ก็คือการ จัดเรียง data ทีนี้ คอม.มันสามารถ compare (เปรียบเทียบ) data ได้แค่ทีละคู่เท่านั้น เมื่อ compare เสร็จ ก็ต้อง จับสลับที่กัน ซึ่งก็คือ swap ต่างกัน ... และเพราะ swap มันใช้บ่อยมาก จึงแยกออกมาเขียนเป็น method ซะเลย!

```
private void swap(int a, int b)   <รับ "index" ของตัวที่จะ swap มา
{   int temp = array[a];          แล้วก็จัดการ swap ตาม index ซะ!
    array[a] = array[b];
    array[b] = temp;
}
```

[Bubble Sort]

เคยพูดไปครั้งนึงแล้วในสรุป c หน้า 34.

bubble เป็นการ sort แบบ basic ที่สุด ... คือใช้ for loop จับ array ทีละ 2 index แล้วไล่ compare ไปเรื่อยๆ (ถึกมากมาย)



< เช่น เปรียบเทียบคู่แรก ถ้าตัวแรกมันมากกว่าก็จะทำการ swap กัน หลังจากนั้นก็เทียบคู่ต่อไปต่อ ถ้าตัวหลังมากกว่า (แบบ 23 กับ 40) ก็จะไม่สลับ swap (อยู่เฉยๆ เลย)

- สังเกตดูว่า ในการไล่ compare ทีละคู่ 1 รอบ เราจะได้ array ช่องสุดท้ายเป็นตัวที่มากสุดเสมอ เพราะ: เราไล่ตั้งแต่ index ที่ 0 ตัวที่มีค่ามากๆ จะถูกดันไปด้านท้ายแถวเรื่อยๆ
- พูดง่ายๆ แบบ 23 ก็จะถูก swap ดันไปทางท้ายแถวไปเรื่อยๆ จนเจอตัวที่มากกว่ามัน (คือ 40) ก็จะเป็นการหยุด 23 ไว้ตรงนั้นก่อน แล้วเปลี่ยนไปดัน 40 ไปทางท้ายแถวต่อไป

! ทีนี้... การไล่ check ทีละคู่แบบนี้ แค่ "รอบเดียว" มันยังไม่ทำให้ array ตัวนี้ sort เรียบร้อย เราก็ต้องทำการวน loop ไล่ check แบบนี้อีก (สังเกตว่าการไล่ check รอบนึงจะทำให้ array เรียงเป็นระเบียบมากขึ้นทีละนิด โดยเฉพาะที่ index สุดท้ายจะเป็นตัวที่มากสุดเสมอ!)

? คำถามคือ อย่างนี้ต้องไล่ check ตั้งแต่หัวแถวถึงท้ายแถวอีกกี่รอบถึงจะชัวร์ว่า array นี้เรียงแล้ว ... ให้คิดจากการที่รอบแรกเราทำให้ index สุดท้ายถูก (ทำให้ตัวมากสุดมาอยู่ช่องสุดท้ายได้) ถ้า array มี n ช่องก็เท่ากับต้องใช้ n-1 รอบ เพราะ: วนครั้งนึงจะทำให้ช่องท้ายๆ ถูกตำแหน่งมากขึ้นเรื่อยๆ (และที่ต้อง -1 เพราะ: ในการ check ครั้งสุดท้าย จะเหลือ data ตัวเดียว ... ในเมื่อตัวอื่นมันอยู่ถูกตำแหน่งกันหมดแล้ว ถึง check ไป มันก็ต้องอยู่ตรงนี้แหละ: งั้นรอบสุดท้ายก็เลยไม่ต้องทำอะไร ชัวร์ๆ

code : bubble sort

ให้ i>1 ไม่ใช่ i>0 เพราะ: รอบสุดท้ายไม่ต้อง check

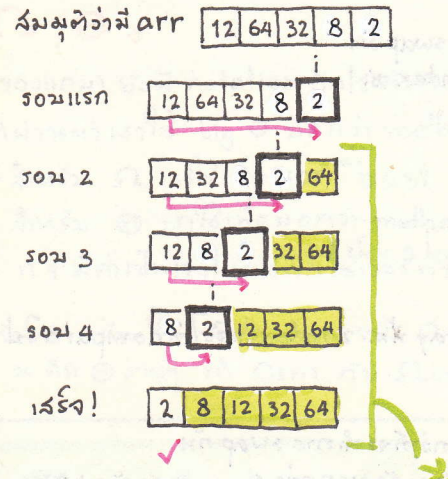
```
for(i=n-1; i>1; i--)
  for(j=0; j<i; j++)
    if(arr[j] > arr[j+1]) swap(j,j+1);
```

ไล่check ทีละคู่ ตั้งแต่ตัวแรกถึงตัวสุดท้าย

check ทั้งหมด n-1 รอบ

อธิบายเพิ่มเติม...

สมมุติว่ามี arr



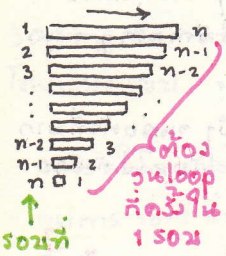
ใน code bubble เราใช้ nested for loop (for ซ้อน for) 2 ตัว โดยตัวใน (for j) ทำการไล่ check ทีละตัว แล้วจัด swap ส่วน loop ตัวนอก (for i) ทำหน้าที่คล้ายๆ เป็น pointer บอก loop ในว่ารอบนี้ให้ check ถึงตัวไหน

- ในเมื่อเรารู้แล้วว่า for j 1 รอบ ตัวสุดท้ายมันถูกต้องชัวร์ การวน check รอบต่อไปก็ check ถึงตัวรองสุดท้ายของรอบที่แล้วก็พอ

∴ หมายความว่า การ check ในแต่ละรอบ index สุดท้าย จะลดลงเรื่อยๆ เราเลยใช้ for i เป็น loop แบบ decrement

Efficiency...

bubble sort เป็น algorithm ที่ใช้หลักการง่ายที่สุดแล้ว แต่ด้านการทำงานมันไม่ค่อยจะดีเท่าไหร่ เพราะ: bubble ใช้ nested for ดังนั้น จะมีจำนวนครั้งที่ต้องทำงานประมาณ n^2 รอบ



◀ แต่ดูจากรูป... การวน loop จะน้อยลงเรื่อยๆ (รูปเป็น ▽) เวลาทำงานจริงๆ เลยเป็น $\frac{n^2}{2}$ รอบ.

∴ $O(n^2)$

※ แต่ข้อเสียของ bubble ยังมีอีกอย่างคือ... มันใช้เวลาตายตัวไม่ว่า data ใน array จะมีการเรียงยังไง แบบว่า ถ้าเราส่ง array ที่ sort แล้วไปทำ bubble อีกรอบ มันก็จะต้องไล่เปรียบเทียบถึง $O(n^2)$ เลย! (ที่เป็นแบบนี้เพราะ: loop ที่ใช้ทั้งหมดเป็นประเภท for ซึ่งทำงานตายตัว)

[Selection Sort]

Selection sort อาศัยหลักการคือ "เลือก" หรือ selection ตามชื่อมันอ่ะ คือมันจะเลือก data ที่มีค่าน้อยที่สุดมาอยู่ข้างหน้า... หลักการ selection เป็นการ sort ที่เหมือนความคิดของคนมากๆ เช่น เวลาเราจะเรียงของอะไรซักอย่าง เราก็จะเลือกหยิบของออกมาชิ้นนึง (โดยส่วนใหญ่จะหยิบชิ้นที่มีค่าไม่น้อยสุดก็มากสุด)แล้วเอาไปวางที่หัวแถวกับท้ายแถว ราวๆ นั้นแหละ

code : selection sort

```
for(i=0; i<n-1; i++)
{
  int min=i;
  for(j=i+1; j<n; j++)
  {
    if(arr[j]<arr[min]) min=j;
  }
  swap(i, min);
}
```

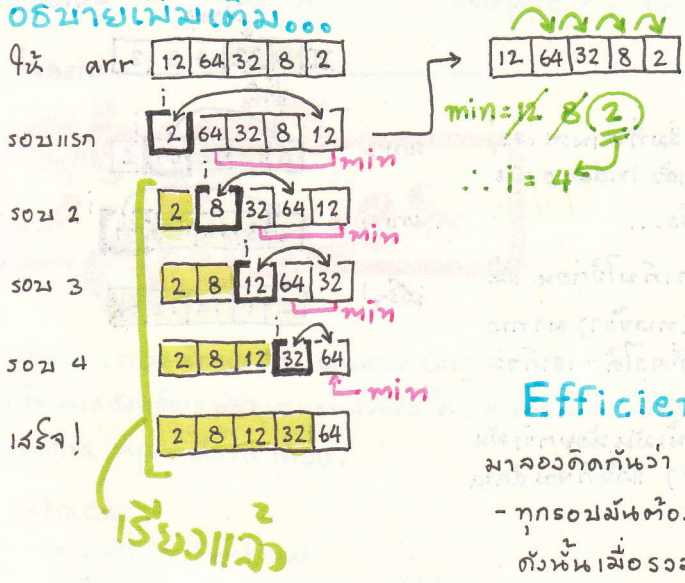
ไปใช้ทน:

หาตัว min

สลับตัวน้อยสุดกับตำแหน่งแรก

- คล้ายๆ bubble... for ตัวนอกทำหน้าที่คล้ายๆ pointer (แค่เลื่อนจาก หน้า → หลัง แทน) โดยใน 1 รอบจะทำการหาตัวที่น้อยที่สุดในช่วง index ที่เหลือ ใช้ for loop เพราะเราต้อง check ทุกตัวถึงจะรู้ว่าตัวไหนน้อยสุด
- min เอาไว้เก็บ "index" ของตัวที่น้อยที่สุด

อธิบายเพิ่มเติม...



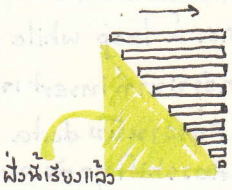
< รอบย่อย... for หาตัว min

- ตาม concept... เราต้องเริ่มต้นจากการหา min ในช่วงที่ยังไม่ได้เรียง (ด้านขวา) แล้วเอามาใส่ไว้ด้านหน้าของ array
- และที่ต้อง swap เพราะไม่งั้นเราต้อง shift ทุก index ไปทางขวา (เสียเวลา swap เร็วกว่า)

Efficiency...

มาลองคิดกันว่า selection มีข้อดีข้อเสียตรงไหน...

- ทุกรอบมันต้องวิ่งหาตัว min ทั้งแต่ i ถึงท้ายแถวของ array ดังนั้นเมื่อรวมกับ for i ตัวนอกด้วยแล้ว มันจะได้ $n^2/2$ (คล้ายๆ buble เลย)



< เมื่อเปรียบเทียบจากรูป (แสดงช่องการหา min ในแต่ละรอบ) ของเขาแคบลงเรื่อยๆ เพราะหลัง for i 1 รอบ ก็จะมีตัวถูกเพิ่มขึ้นทีละตัว

∴ $O(n^2)$

[Insertion Sort]

Insertion sort อาศัย concept ของการเอา data ไปแทรกใน index ที่มันควรจะอยู่ แนวคิดก็ใกล้เคียงกับความคิดของคนอยู่นิดหน่อย เพราะ: หลักการของมันคล้ายๆ selection sort แต่สลับการทำงานกัน คือ selection จะทำการหา index ก่อน แล้วเอามาใส่ใน ทำแหน่งที่แน่นอน (i) ส่วน Insertion จะเริ่มโดยการเอา data จากตำแหน่งที่แน่นอน (i) ไปหาที่ลง หรือ แทรก นั่นแหละ: ใน index ที่มันควรอยู่

code : Insertion Sort

```
for(i=1; i<n; i++)
{
    int temp = arr[i];      // ดึง data ออกมา
    j=i;
    while(j>0 && arr[j-1]>=temp)
    {
        arr[j] = arr[j-1];
        j--;
    }
    a[j] = temp;            // insert
}
```

type ของ arr

shift เขยิบ เรื่อยๆ จนหาที่ลงได้

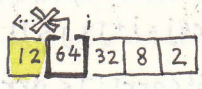
- ใน insertion เราจะเริ่มที่ index ที่ 1 ไม่เริ่มที่ 0 เพราะ: เราจะเอา data ใน index นั้น ขยับไปทางซ้าย แล้ว index 0 มันอยู่ซ้ายสุดอยู่แล้ว เลยไม่ต้องขยับไปไหนแล้วไง...
- เราจะทำการหยิบ data ที่ index นั้นออกมาเก็บไว้ก่อน แล้วทำการ shift data ใน index ก่อนหน้ามัน (ทางซ้าย) มาทางขวาเรื่อยๆ จน data ตัวที่หยิบออกมา "หาที่ลงได้" เราก็จะ insert มันลงตรงนั้นแหละ:

※ ถ้าหยิบ data ออกมาแล้ว แต่ index ก่อนหน้ามันน้อยกว่ามัน ก็จะไม่มีการเข้า while loop (ไม่มีการ shift) แล้วก็จับ data ที่หยิบออกมา ยัดกลับเข้าไปที่เดิม!

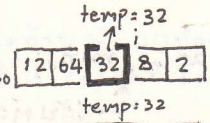
อธิบายเพิ่มเติม...

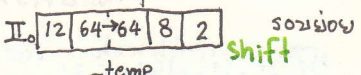
กำหนด arr | 12 | 64 | 32 | 8 | 2 |

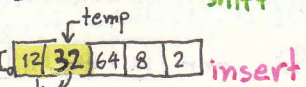
รอบแรก



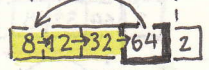
รอบ 2



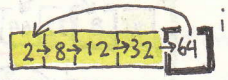




รอบ 3



รอบ 4

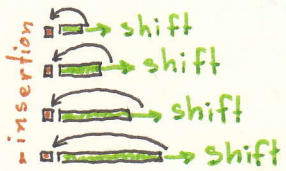


เสร็จ!

Efficiency...

ที่ผ่านมา เราเจอ nested for 2 ตัวซ้อนกัน ทำให้มี $O(n^2)$... สำหรับ insertion ต่างจาก 2 ตัวแรก คือ มันเป็น for ซ้อน while เลยต้องมาวิเคราะห์ loop while ก่อน เพราะ: มันเป็น loop ที่มีจำนวนครั้งการวน loop ในแต่ละรอบไม่ตายตัว (คือจนกว่าจะหาที่ insert เจอ อ่ะ:)

- worst case กรณีแย่สุดของ while ก็คือ เรานยิบ data ออกมา แล้วทำแหน่งที่จะเอาไป insert ก็คือ ที่หัวแถว หรือก็คือที่ index=0 (ต้อง shift ทุกตัวเลย) เมื่อเป็นแบบนั้น อุตส่าห์ใช้ while ได้ก็ไม่ต่างจาก for เลย ∴ ก็คิด big O เป็นแบบ for ซ้อนกัน



∴ $O(n^2)$

- best case เนื่องจากว่ามันมี while มันก็เลยมีโอกาสที่จะไม่เข้า loop ด้วย ก็คือกรณีที่ data ใน index ก่อนหน้ามันทั้งหมดน้อยกว่ามันอยู่แล้ว ก็คือตัวมันไม่ต้องขยับไปไหนเลย

∴ เมื่อไม่เข้า loop while เลย ก็แปลว่าทั้ง method วนแค่ loop for ตัวนอก จึงมีกรณีดีสุด (กรณีที่ทั้ง array มัน sort มาเรียบร้อยแล้ว) เป็น n ∴ $\Omega(n)$

Summary : Sorting.

bubble	Selection	Insertion
- ใช้การ check ทีละตัว แล้ว swap คู่นั้นเลย - วน nested for 2 ชั้น - ตัวมากสุดจะเขยิบไปทางขวา (ท้าย array) เรื่อยๆ $O(n^2)$ $\Omega(n^2)$ compare n^2, swap n^2	- ใช้การเทียบวนloopหาตัว min - swap ตัวmin กับ indexที่ i - เสียเวลาตอนหาตัว min เพราะต้องวน for checkทุกตัว $O(n^2)$ $\Omega(n^2)$ compare n^2, swap n	- ใช้การหยิบ data ออกมา (temp) - shift data ทางซ้ายของมันไปเรื่อยๆ จนหาที่ลงได้ (insert) - ทำงานเร็ว (กว่า 2 ตัวนั้น) เพราะมีการใช้ while $O(n^2)$ $\Omega(n)$ compare n^2, swap n^2

Simple Sorting / จบ.

Stacks & Queues.

Stack (กองซ้อน) กับ Queue (แถวลำดับ)
เป็นการจัดเรียงข้อมูลแบบหนึ่งโดยสนใจไปที่ "ลำดับ" ของการ input data เข้ามา...

Stack

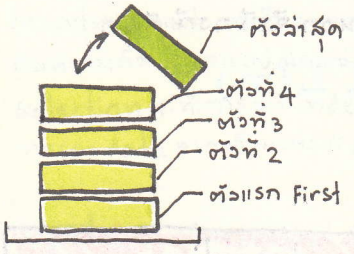
↳ Last in, First out
- สิ่งที่เข้ามาที่หลังสุดจะออกไปก่อน
- เหมือนเราวางกองหนังสือซ้อนๆ กัน เล่มที่วางสุดท้ายก็ถูกหยิบก่อน

Queue

↳ First in, First out
- เข้าก่อนออกก่อน
- เหมือนการเข้าคิวซื้อของ (อย่าบอกว่าไม่เคยเข้าแถวนะ!)

✱ ในเรื่องstack กับ Queue...เราก็ต้องมีการเก็บdataเอาไว้หลายๆ ตัว ซึ่งก็หนีไม่พ้นการใช้ array อีกแล้ว ^^!

[Stack.]



เวลาเราใช้ stack จะมี method ให้ใช้ดังนี้...

- push ใช้เพื่อ add ของเข้า stack (ใส่ตามแนวบนสุด)
- pop ใช้เพื่อ remove ของ (ตัวล่าสุด) และ return กลับไปด้วย
- peek ใช้ตลายๆ pop คือ return ตัวล่าสุดกลับไป แต่ไม่ remove
- isEmpty ใช้ check ว่าตอนนี้ stack ของเรามีอะไรใส่อยู่มั้ย
- isFull ใช้check ว่าช่องเก็บstack ตัวนี้เต็มรึยัง (เพราะใช้เป็นarray)

```
class Stack
{
  private int size;
  private int i;
  private Object[] arr;

  public Stack(int n)
  {  size=n;
     arr= new Object[n];
     i= -1;
  }
  public void push(Object in)
  {  arr[++i]=in; }

  public Object pop()
  {  return arr[i--]; }
  public Object peek()
  {  return arr[i]; }

  public boolean isEmpty()
  {  return i==-1; }
  public boolean isFull()
  {  return i== size-1; }
}
```

size เอาไว้ check ขนาดของarray

ทำหน้าที่เป็นเหมือน pointer ชี้บอกว่าตำแหน่งล่าสุดที่ add data เข้าไปคือ index ไหน

array ของ Object เอาไว้เก็บของ

constructor ทำการ set size และ new array ตอนนี้ให้ i เป็น -1 เพราะยังไม่มี data เลยซักตัว

add data เข้าไปที่ตำแหน่ง "ถัดจากตัวที่แล้ว" ※ตัวที่แล้วเก็บที่ index i

remove data โดยการเลื่อน i (ไม่ได้ลบจริงๆ ก็แค่ไม่สนใจมันแล้ว!)

return ตัวล่าสุด

ตรวจว่า Stack เราว่าง (ไม่มี data อยู่เลย) รีเปล่าโดยตรวจโดยการ check i ซึ่งทำหน้าที่เป็นpointer

check กลายๆ isEmpty... คือ array จะเต็มก็ต่อเมื่อ i ชี้มาถึงตัวสุดท้าย (index ที่ size-1)

arr[++i] คือ...?

++ (หรือ --) เป็นการ increment variable ตัวนั้นขึ้น 1 โดย ++ สามารถวางไว้หน้าหรือหลัง var. ก็ได้ มีค่าเหมือนกันคือ increment แต่ข้อแตกต่างคือช่วงเวลาของการ increment

- i++ หมายถึงเอาค่า i ไปใช้ก่อนแล้วค่อย increment
- ++i หมายถึง ก่อนทำอะไรทั้งหมดให้ increment ก่อนแล้วค่อยนำค่า i ที่ increment แล้วไปใช้

เช่นถ้า i = 0

↳ arr[i++] จะหมายถึงการอ้างถึง arr ตัวที่ 0 แล้วหลังจากนั้นค่อย increment i เป็น 1
↳ arr[++i] จะหมายถึงให้ increment i ให้เป็น 1 ก่อนแล้วค่อยใช้ ∴ อ้างถึง arr ตัวที่ 1

Stack ในชีวิตจริง

- **Web browser** ปุ่ม back / forword ... การกด url link ไปหน้าต่อไป หน้านั้นจะถูกเพิ่มใน stack เมื่อ กด back ก็คือจะไป pop ข้อมูลล่าสุด ซึ่งคือหน้าก่อนหน้ามันออกมา
- **undo.** การกดปุ่มย้อนกลับของโปรแกรมพวก Office , และอื่นๆ
- **Parsing (check syntax)** เช่นมีสมการเลขตัวหนึ่ง เราอยากรู้ว่าเขาใส่วงเล็บ (), {}, [] ถูกที่ หรือใส่ครบรึเปล่าก็ใช้ stack นี่แหละ

Postponing

เป็นรูปแบบการเขียนสมการเลขโดยจาก "operator" ไว้คนละตำแหน่งกัน

- Infix การเขียนแบบ op อยู่ระหว่าง number เช่น a+b
- Prefix การเขียนแบบ op อยู่ข้างหน้า number ทั้ง 2 ตัว เช่น +ab (หมายความว่า a+b)
- Postfix การเขียนแบบ op อยู่ข้างหลัง เช่น ab+ (หมายความว่า a+b)

※ เวลา computer จะคิดสมการหนึ่ง มันจะขัดรูปแบบ Infix → Postfix

How to Infix → Postfix

Number (operand)	copy เลขตัวนั้นไปออกเป็น output เลย
(	push ใส่ Stack ไปเลย
)	pop data ออกจาก Stack ไปเรื่อยๆ จนเจอ [( ] แล้วค่อยหยุด (pop ไปออก output)
Operator	- ถ้า stack ว่างอยู่ (Empty) ให้ push op ตัวนั้นใส่ stack ไป - ถ้า data ที่ push ใส่ stack ไปตัวล่าสุด เป็น [( ] ก็ให้ push op ใส่ stack ไปเลย - ถ้า data ที่ push ใส่ stack ไปตัวล่าสุด เป็น op ให้ดูว่า อยู่ระดับ สูง หรือต่ำกว่ามัน (*, /, % อยู่ระดับสูงกว่า +, -) ↳ ถ้าระดับต่ำกว่ามัน ให้ push op ตัวนั้นใส่ stack ไปเลย ↳ ถ้าระดับสูงกว่าหรือเท่ากัน ให้ pop data ตัวนั้นออกเป็น output เลย แล้วก็ให้ push op ใส่ stack ไป
No more!	pop จาก Stack ไปออก output เรื่อยๆ จนหมด

การแปลง infix เป็น postfix เราจะใช้วิธี อ่าน String มาทีละตัว (charAt(0) ถึง charAt(n)) โดยอ่านมาแล้ว จะทำอะไรก็ต้องดูว่ามันเป็น number, operator หรือ (,) ซึ่งจะทำตามตารางข้างบน

※ output คือคำตอบตอนสุดท้าย

※ stack เอาไว้เก็บพวก op และ (,) ซึ่งพวกนี้จะเป็นปัญหาในบรรทัดสุดในเรื่องนี้ ... คือ กรณีของการเจอ op มีหลายขั้นตอนมาก (จดเอาให้ได้ละกัน)

Ex. 3+(2-7/5)*2-4+((9-4)*8)

(ขอบันซีร็อก ไม่มีสีอ่านยังไงก็ละก้นหน่ะ)

	output	Stack.
3	3	
+	3	+
(	3	+(
2	32	+(
-	32	+(-
7	327	+(-
/	327	+(-/
5	3275	+(-/
)	3275/-	+
*	3275/-	+*
2	3275/-2	+*
-	3275/-2*+	-
4	3275/-2*+4	-
+	3275/-2*+4-	+
(	3275/-2*+4-	+(
(	3275/-2*+4-	+((
9	3275/-2*+4-9	+((
-	3275/-2*+4-9	+((-
4	3275/-2*+4-94	+((-
)	3275/-2*+4-94-	+(
*	3275/-2*+4-94-	+(*
8	3275/-2*+4-94-8	+(*
)	3275/-2*+4-94-8*	+
	3275/-2*+4-94-8*+	ALL

number → ออก output เลย
op → stack ว่าง (Empty) push ใส่ stack เลย
(→ ใส่ stack เลย
number → ออก output เลย
op → ตัวล่าสุดใน stack เป็น (, push op ใส่ stack.
number → ออก output เลย
op → ตัวล่าสุดใน stack เป็น op ที่ต่ำกว่า, push op.
number → ออก output เลย
) → pop stack ออก output จนเจอ (ก่อยหยุด
op → ตัวล่าสุดใน stack เป็น op ที่ต่ำกว่า, push op.
number → ออก output เลย
op → opใน stack ระดับสูง/เท่ากับมัน, pop ไปออก output
number → ออก output เลย (pop จนไม่เหลือเงื่อนไข)
op → opใน stack ระดับเท่ากัน, pop ไป output เลย!
(→ ใส่ stack เลย
(→ ใส่ stack เลย
number → ออก output เลย
op → ตัวล่าสุดใน stack เป็น (, push op ใส่ stack
number → ออก output เลย
) → pop stack ออก output จนเจอ (ก่อยหยุด
op → ตัวล่าสุดใน stack เป็น (, push op ใส่ stack.
number → ออก output เลย
) → pop stack ออก output จนเจอ (ก่อยหยุด
none! หมดแล้ว (โว้ย!) pop ทุกตัวใน stack ออก output ให้หมด!!

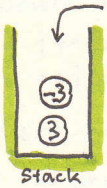
Computer คิดสมการเลขในรูปแบบ postfix.

เวลาเราเขียน code statement เราจะเขียนในรูป infix ซึ่งเวลา com. จะเอาไปคิด มันต้องแปลงให้อยู่ในรูป postfix ก่อน... เช่น

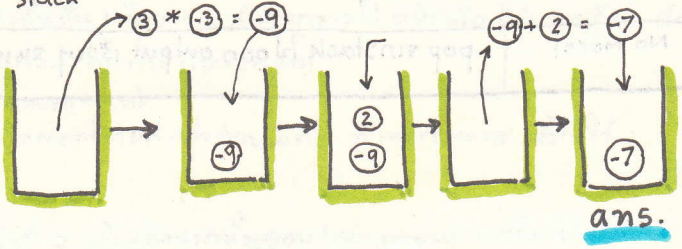
3*(2-5)+2 → 325-*2+

step1: นำ number จาก postfix ใส่ลงไปใน stack ทีละตัว...

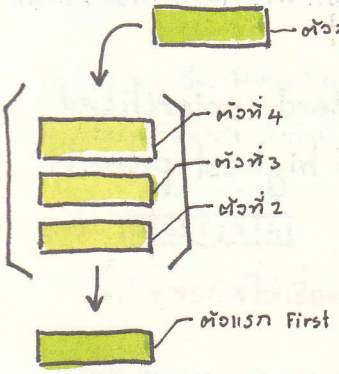
step2: เมื่อเจอ op ... ให้ pop ของใน stack ออกมา 2 ตัว แล้วทำ operation นั้นได้ result ออกมาตัวนึง



step3: push result กลับลงไปใน stack แล้วอ่าน data ตัวต่อไปใน postfix แล้วกลับไป step1 ใหม่



[Queue]



method ของ queue...

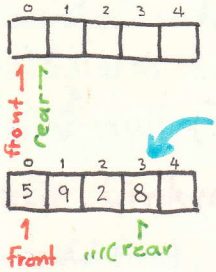
- enqueue insert data เพิ่มเข้า queue
- dequeue (เหมือน pop) ดึง data ออกจาก queue
- peek
- isEmpty, isFull } เหมือน stack

Concept การเขียน algorithm Queue

• การเขียน class queue โดยใช้ array เขียนแบบ basic ได้โดยใช้หลักคล้ายๆ stack คือมี pointer ชี้ index สุดท้าย เอาไว้ใช้ตอน enqueue ส่วนเวลา dequeue ก็ส่ง array[0] ตัวแรกกลับไป และทำการ shift data ที่เหลือเขยิบขึ้นมา 1 index

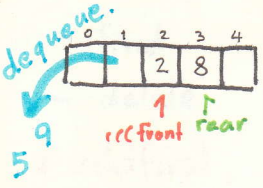
※ แต่ไม่ดีม้วง...!? เพราะถ้าเราเขียนแบบนี้ method "dequeue" ของเราจะเป็น method เดียวที่มี big O เป็น O(n) (method ทั้งใน stack และ queue มันเป็น O(1) หมดเลย!) เพราะเราต้อง shift data 'เกือบ' ทั้ง array

• เราก็เลยจะเขียนในมุมโดยมอง array เป็น "วงกลม" และมี pointer 2 ตัวชี้



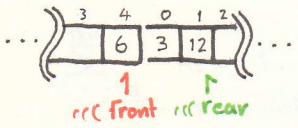
- ตอนเริ่มมี array เปล่าๆ โดยมี pointer ชี้อยู่ที่จุดเริ่ม 2 ตัว

- เมื่อ add ของเข้ามา ก็ทำเหมือน stack pointer rear (ข้างหลัง) ก็จะเป็นตัวบอกว่าต้อง add ตรงไหน แล้วก็ rear++



- เมื่อจะ dequeue ออกไป ก็จะใช้ pointer front เป็นตัวบอกว่าจะ return ตรงไหนกลับ แล้วมันก็จะ: front++

※ การที่ pointer ขยับ คือเราไม่สนใจ data ที่ตำแหน่ง front เดิมอีกแล้ว ก็เหมือนมันถูก delete ไปนั่นแหละ!



- สำหรับกรณีที่ pointer ชี้ไปสุดปลาย array แล้ว มันจะคิดเหมือนว่า วนกลับไปที่หัวแถวต่อ แบบงูกินหาง (ว้าปได้! เหมือน K-map)

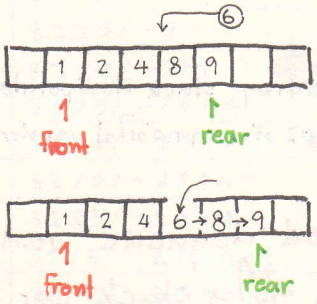
Priority queue.

คือ queue ที่มีการ **จัดคิวได้!** หรือก็คือการที่ data ตัวที่เข้ามาหลังสุด จะถูกเอาไป insert ที่ไหนก็ได้ใน array

- **Priority queue ก็คือ การทำให้ queue กลายเป็น orderd (prioritized) โดยจะมี key value เป็นตัวcheck ว่า data ตัวไหน มี highest priority**

โดยรูปแบบของ priority queue จะมีหลักๆ 2 แบบคือ...

- Ascending ใช้ key ว่า ตัวที่ smallest จะมี highest prioity
- Decending ใช้ key ว่า ตัวที่ biggest จะมี highest prioity



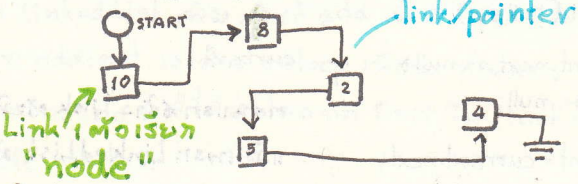
การใช้ priority จะทำให้ data ตัวหลังที่เพิ่ง add เข้ามา จะไม่เข้าไปอยู่ที่ index rear เสมอไป แต่มันจะทำที่ insert (คิดคล้ายๆ เรื่อง insertion sort อ่ะ :)

Linked List.

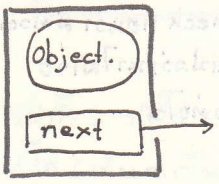
※ อ่านเพิ่มเติม สรุป Java ของเรา หน้า 58 - 60 ด้วยนะ!

LinkedList เป็น Data Structure แบบ Obj. ใช้สำหรับการเก็บ data ที่มีปริมาณมากมาย...
เพราะ: data ตัวต่อไป ที่เก็บใน Linked List จะไม่ได้เก็บไว้ใน memory ข้อต่อกันแบบ array

เก็บต่อกันไปเรื่อยๆ



เวลาเราใช้ array การที่เราจะใช้ data ในช่องไหน เราระบุด้วย index (เช่น บอกว่า arr[4] ก็คือบอกว่า data ตรง address ที่หัว array แล้ว +4 (แบบ assembly)) ... แต่การใช้ linkedlist data แต่ละตัวมันกระจัดกระจายกันไปใน memory (แบบ random) มันเลยต้องมีส่วนเก็บ address ว่าตัวต่อจากมันเก็บอยู่ตรงไหน!



ส่วนประกอบของ Link 1 ตัว.

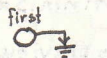
- Object data เอาไว้เก็บข้อมูล
- Link next เอาไว้เก็บ address ตัวต่อไป

LinkedList ก็คือการเอา Link แต่ละตัวมาต่อกันไปเรื่อยๆ และเนื่องจากว่า L.L. มันไม่มีทั้ง index และ size เราจึงต้องมี Link สร้างมาตัวหนึ่ง ชื่อว่า "first" เอาไว้ระบุว่า Link ตัวไหนเป็นหัวแถว ... และที่ Link ตัวสุดท้าย next ของมันก็ให้ชี้ null (ไม่มีอะไรต่อแล้ว!)

basic LinkedList.

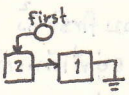
- add First, add Last
- find
- delete

• addFirst, addLast



ในตอนแรกที่สร้าง LinkedList ขึ้นมา มันยังไม่มี data เราต้องกำหนดก่อนว่าตอนนี้ first คือ null

ต่อมาเมื่อเรา add data ตัวใหม่เข้าไปที่หัวแถว เราก็สั่งให้ newData.next = first.next คือตัวใหม่ที่เข้ามาให้ชี้ตัวต่อไปของมัน เป็นตัวเดียวกับที่ first เคยชี้ แล้ว update first ให้เท่ากับ newData ตัวใหม่

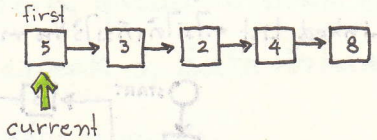


ถ้าเราจะ: addLast ... เราก็ต้องหาตัวสุดท้ายก่อน และเนื่องจากว่ามันไม่มี index เราต้องใช้การวนloop หาทีละตัว (ตัวท้ายสุดคือตัวที่ next = null) แล้วก็แทรก Link ตัวใหม่ของเราลงตรงนั้นแหละ

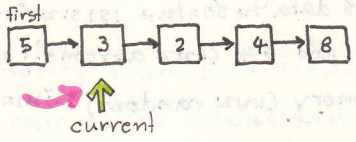
• find

ในเรื่อง LinkedList การสร้าง class Link เราอาจมี Link มี properties เพิ่มมาอย่างหนึ่งคือ id ซึ่งเอาไว้เป็น key word ว่าเราจะ find LinkedList ตัวที่มี key word ตรงกับที่หา

```
Link current = first;
while(current.id != key)
{
    if( current.next == null)
        return null;
    else
        current = current.next;
}
return current;
```



ตอนแรกสร้าง Link ตัวหนึ่งขึ้นมาเก็บว่าตอนนี้เรา find ถึงตัวไหนแล้ว เพราะ LinkedList มันไม่มี index



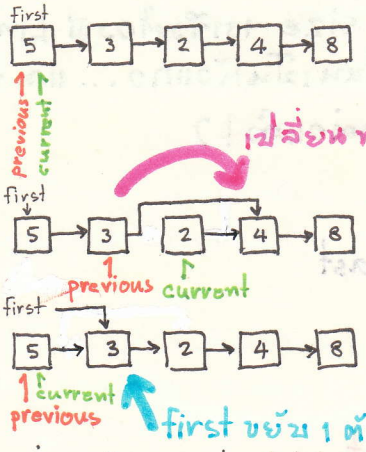
ต่อมา...เราใช้ while วนloop หา Link ตัวที่มี id ตรงกับ key ที่เราหา แล้วในการวนรอบนึงก็ต้อง check ก่อนว่ามีตัวต่อจาก current มั้ย ถ้าไม่มีแล้ว (null) ก็แปลว่าหาไม่เจอ แต่ถ้ายังมีต่อก็ขยับ current ไปชี้ตัวต่อไป

• delete

การจะลบ Link ตัวหนึ่งออกจาก LinkedList เราก็ต้องหาก่อนว่า Link ตัวนั้นอยู่ตรงไหน (ก็คือต้องทำการ find ซะก่อน)

```
Link current= first;
Link previous= first;
while(current.id!=key)
{
    if(current.next==null)
        return null;
    else
    {
        previous= current;
        current= current.next;
    }
}
if(current == first)
    first = first.next;
else
    previous.next = current.next;
return current;
```

find



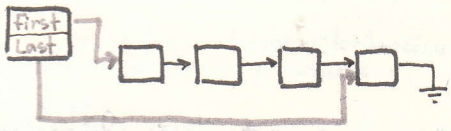
เมื่อเรา find ตัวที่จะ delete ออกจาก list เจอ เราก็จะทำการลบตัวนั้นออกจาก List โดยการเปลี่ยน next ของตัว previous ไปชี้ตัวต่อจาก current เลย (โดดข้าม current ไปเลย)

※เราต้องมี previous เป็น pointer ขยับตาม current เพราะเราต้องสั่ง set ค่า next ของตัวก่อนหน้า current ใหม่!

แต่การ delete ต้องระวังอยู่เรื่องนึงคือเมื่อเราจะลบ first ทิ้ง เพราะตอนอยู่ที่ first previous กับ current ยังอยู่ที่เดียวกันอยู่... เราเลยจะเลื่อนไปใช้ first = first.next เลย

[Note] การ delete ในเรื่อง Data Structure ส่วนใหญ่จะไม่ได้หมายถึงการ"ลบ" data ทิ้งไป แต่จะเป็นการ "เลิกสนใจมัน" ซะมากกว่า

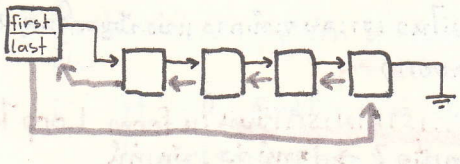
Double ended List.



เป็นรูปแบบ เขียน code Linked List ให้ node แรก หรือ first มีการ Link ไปยัง node อื่น 2 เส้นทาง คือ Link ไปยัง ตัวแรก first และ ตัวสุดท้าย Last

ข้อดี : เวลาเราจะ add data เข้า LinkedList เพิ่ม ... ถ้า add เข้าที่ first ก็ไม่มีปัญหาต่อ big O เท่าไหร่เพราะ มี O(1) แต่ถ้าเราจะ addLast เราต้อง วนloop เพื่อหา ท้ายแถว มันเสียเวลามากมาย การที่เรามีการ Link จาก หัวแถว → หางแถว จะทำให้เราโดดจาก first ไป last ได้เร็วมาก ! ซึ่งทำให้ง่ายต่อการ add และ delete ตัว Last.

Double linked List.

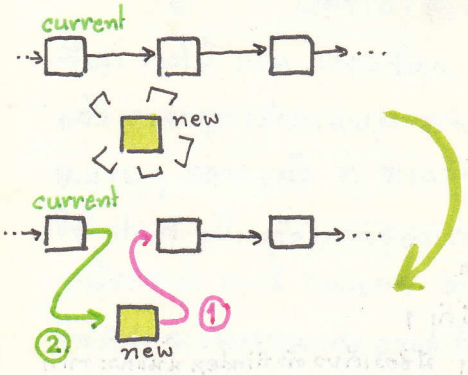


เป็นรูปแบบ เขียน Linked List ที่เติมรูปแบบมากมาย เพราะ มันสามารถ วิ่งไป วิ่งมาได้เต็มที่ ... โดยเราต้อง เพิ่ม properties ชื่อว่า "previous" เข้าไปใน class Link (next เก็บ ตัวต่อจากมัน, previous ก็เก็บตัวก่อน)

ข้อดี : ทำให้การเข้าถึง / เรียกใช้ data ใน List ของเรา ทำได้ง่าย ลื่นไหล ดี เพราะ มันวิ่งไปไหนก็ได้ (เก็บจะเป็น array อยู่แล้ว !)

ข้อเสีย : เวลา add / delete จะทำได้ ยากมาก !! ลองคิดดูดิ แค่ add กับ delete แบบ basic ธรรมดาๆ ก็เริ่มงงกันแล้ว (มี next แค่ตัวเดียวเองนะ หนึ่ง) แล้วนี่มีการเพิ่ม previous ขึ้นมาอีก code ใน method ต่างๆ ก็จะยาว + ซับซ้อนมากมาย !

การ add Link เข้าที่กลาง Linked List



ID การจะ add ตรงกลาง (สมมุติว่าจะ add "new" ต่อจาก current) ก่อนอื่นเราต้อง...

1. บอกให้ new.next = current.next เพราะ: ตำแหน่งที่มันจะแทรกคือ ต่อจาก current แปลว่า ตัวที่ เคย ต่อจาก current, เมื่อเรา add เสร็จ มันจะกลายเป็นตัวที่ต่อจาก new
2. เปลี่ยน current.next = new ให้ตัวต่อจาก current ให้เป็น new ก็จะได้ List ใหม่ที่มี new มาแทรกกลางแล้ว !

※ อ่านเพิ่มสรุป Java ของเราหน้า 52-55 นะ

"Recursion" หรือ "Recursive" คือการเขียนให้ method (หรือ function) เรียกใช้ตัวเอง !

Divide & Conquer.

เป็น concept การแก้ปัญหาแบบ แบ่งส่วน ซึ่งเป็นหลักของ Recursion เลยแหละ:

"แล้วแบ่งส่วนคือ...? ทำเมื่อ...? "

ปัญหาบางตัว อาจเป็นปัญหาที่ใหญ่มากซึ่งเราคิดทีเดียวไม่ไหว เราเลยจะทำการแบ่งปัญหาใหญ่นั่นออกเป็นปัญหาเล็กๆ หลายๆ ตัว (แบ่งไปเรื่อยๆ จนกว่าจะคิดออก)

[ขอบอก] การเขียน Recursion ส่วนใหญ่ ... เราสามารถเขียนใน form Loop ได้
> เอ๊ะ !? ... แล้วจะเขียน Recursion เพื่อ ? → โจทย์สั่ง, มันเท่

form หลักๆ ของ Recursion.

```
type  methodName(int n)
{
    if(n== กรณีเล็กสุด) return ซักอย่าง;
    else
    {
        // แบ่ง n เป็นส่วนเล็กๆ เช่น n-1
        return methodName(n-1);
    }
}
```

◀ รูปแบบทั่วไปของ Recursion คือจะมีการ check if-else ... คือถ้าเป็นกรณีที่เล็กพอที่เราจะหาคำตอบได้แล้วก็ return คำตอบเลย แต่ถ้ามันยังใหญ่อยู่เราก็ลดขนาดมัน แล้วเรียก recursion

Step การทำ recursion (ดูตัวอย่างไปด้วย)

1. ให้พยายามหาก่อนว่า solution ที่เราหาได้เลยของ problem
   เช่น - การหา factorial กรณีเล็กสุดคือ 1! และ 0! มีค่า 1
   - หาตัว min ใน array n ช่อง กรณีเล็กสุดคือ n=1, มีช่องเดียว ตัวที่ index นั้นแหละ: min
   - Tower of Hanoi กรณีเล็กสุดคือ มี 1 disk ก็ย้าย source → target

2. เตรียม solution สำหรับกรณีที่ไม่เล็กที่สุดโดย แตก problem ออกเป็น 2 ส่วน คือ ประมาณว่าตัว n ออกมาตัวนึง
   เช่น - factorial n! แตกเป็น n * (n-1)! → หาด้วย recursion
   - หาตัว min ใน array n ช่อง แตกเป็น array ช่องที่ 0 กับ [array ช่องที่ 1 ถึง n] → recursion
   - Tower of Hanoi มองว่าเป็น disk 2 ชุด (n-1) disk กับ disk ฐาน 1 disk → recursion

Ex1. factorial (basic มาก)

```
int fac(int n)
{
   if(n==0 || n==1) return 1;
   else
      return n* fac(n-1);
}
```

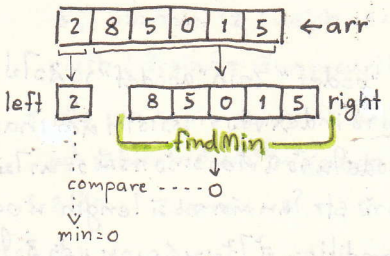
เรียก recursion ส่วนที่เล็กลง

◀ เราก็ดูก่อนว่า กรณีเล็กสุด ก็คือ n=0 กับ n=1 ซึ่ง return ตัวมันได้เลย (ans คือ 1) แต่ถ้าไม่ใช่กรณีเล็กสุด ก็ให้ return n*(n-1)! ซึ่งค่า ans ของ (n-1)! ก็เรียก recursion คิดคำตอบให้เลย!

Ex2. find min ใน array index ที่ start ถึง end.

```
int findMin(int[] arr, int start, int end)
{
   if(start==end) return arr[start];
   else
   {   int min = findMin(arr, start+1, end);
       if(arr[start]<min) return arr[start];
       else return min;
   }
}
```

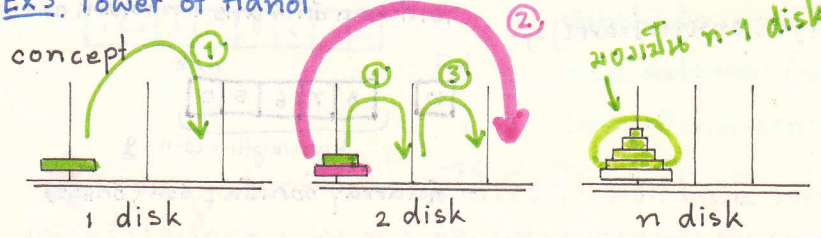
หา min ช่องที่เหลือ



◀ ตอนแรกให้ดูก่อนว่าถ้า start กับ end เท่ากัน (คือ หา min จาก array ช่องเดียว) ซึ่งแน่นอน มีอยู่ช่องเดียว ตัวนั้นแหละที่เป็น min ... กรณีใหญ่ก็คือ แตก array ออกเป็น 2 ส่วน (left กะ right ดูรูปนะ:) ส่วน right เราจะทำการ recursion (assume ว่า) ให้หาตัว min ในช่วง right (คือ 0) แล้ว return กลับมาให้เรา จากนั้นเมื่อเรารู้ค่า left (มีอยู่ตัวเดียว) กับ ตัว min ของช่วง right ก็แปลว่า ตัวที่น้อยกว่าใน 2 ตัวนี้ นั่นแหละคือ ตัว min ของ array index ที่ start ถึง end

[Tip] การที่เราเขียน method recursion แบบนี้ได้ เราต้อง "ชัวร์" ว่าเวลาเราทำการเรียก recursion แล้ว มันจะเข้าใกล้กรณีเล็กสุดที่เราวางไว้ เช่น เรื่อง fac กรณีเล็กสุดคือ 0, 1 แล้วเราเริ่มเรียกที่ n เยอะๆ แล้ว n จะถูกเรียกแบบ n-1 ผ่าน recursion แปลว่าหลังการเรียก recursion หลายๆ รอบแล้ว n จะเหลือ 0 กับ 1 ตามที่เรา set ไว้, เรื่อง find Min เราใช้ start เป็นตัวบอกขอบเขตๆ คือ กรณีเล็กสุดคือ start = end ตอนเริ่มต้น start จะน้อยกว่า end แน่นอน ตอน recursion เราใช้ start+1 เพิ่มค่าขึ้นเรื่อยๆ จนในที่สุดจะเพิ่มจนเท่า end ก็ตรงกับกรณีเล็กสุดที่เตรียมไว้!

Ex3. Tower of Hanoi

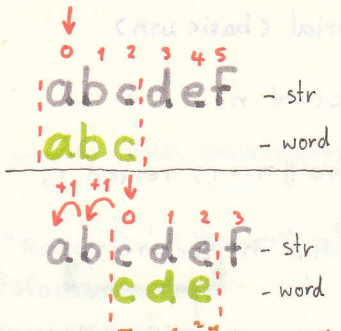


◀ ใช้หลักการว่า มอง 2 disk กับ n disk เหมือนกัน แค่ตอนที่จะย้าย step ① เราจะมองว่ามันเป็น n-1 disk ซึ่งจะเอา n-1 นี่ย้ายไปเสาที่ 2 ก็เรียก recursion อีกนั่นแหละ

※ code Hanoi แบบละเอียดกว่านี้ อ่านในสรุป Java ของเราหน้า 54-55 เลยนะ!

Ex 4. หา indexOf ใน String

```
int indexOfWord(String word, String str)
{
    if(word.length > str.length) return -1;
    if(word.equals(str.substring(0, word.length)))
        return 0;
    else
    {   int find = indexOfWord(word, str.substring(1));
        if(find == -1) return -1;
        else return find+1;
    }
}
```



note: โจทย์นี้ไม่ควรจะเขียนแบบ recursion เท่าไรนะ (แต่ให้เขียนก็เขียนได้!), คือตอนแรก... การหา indexOf ที่ง่ายที่สุดคือกรณี คำที่เราจะหาอยู่ต้นคำเลย เช่น หา abc จาก abcdef ซึ่งเรา return 0; ได้เลย... แล้วถ้าจะเรียก recursion ก็คือตัดช่วง "bcdef" (จาก "abcdef") ส่งไปหา indexOf อีก ถ้ามันเจอมันก็จะ return 0; กลับมา แต่ได้ 0 ตัวนี้มันไม่ใช่ index จริงๆ เพราะเราตัด char ข้างหน้าออกไป 1 ตัว เราเลยต้อง +1 คืนให้มัน

- การเขียน code ในข้อนี้ต้องระวังกรณีว่าหาไม่เจอด้วย (คือ word ไม่อยู่ใน str) คือเมื่อเราตัด str ไปเรื่อยๆ จะมีช่วงหนึ่งที่ str โดนตัดจนมี length น้อยกว่า word ซึ่งแน่นอน... ไม่มีทางที่ word จะอยู่ใน str แล้วเราก็วาง condition นี้ไว้บนสุดเลย แล้วสั่งให้ return -1;

Ex 5. Knapsack Problem.

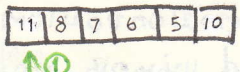
เป็นโจทย์ที่เรามี ของซึ่งมีน้ำหนักต่างกันหลายชิ้น แล้วเราต้องการรู้ว่าเมื่อเราต้องการหาน้ำหนักรวมน้ำหนักหนึ่ง ต้องหยิบชิ้นไหนบ้าง เช่นมี 11, 8, 7, 6, 5, 2 น้ำหนักรวม = 20 เราก็ต้องหยิบ 8, 7, 5

```
int[] knapsack(int[] arr, int start, int weigh)
{
    int i, j;
    if(weigh < 0 || start >= arr.length) return new int[0];

    int[] ans;
    for(i=start; i<arr.length; i++)
    {
        if(arr[i] == weigh)
        {   ans = new int[1];
            ans[0] = arr[i];
            return ans;
        }

        int[] temp = knapsack(arr, i+1, weigh-arr[i]);
        if(temp.length > 0)
        {
            ans = new int[temp.length+1];
            ans[0] = arr[i];
            for(j=0; j<temp.length; j++) ans[j+1] = arr[j];
            return ans;
        }
    }
    return new int[0];
}
```

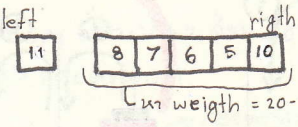
① (ส่วนที่ล้อมไว้)

note: knapscak มีการเขียน code ได้หลายวิธี... อันนี้เป็นแค่ตัวอย่าง ซึ่งใช้ array เป็นตัว return (code นี้เราคิดเองว)... อาจจะใช้ LinkedList มาช่วยแทน array ก็ได้นะ:

- method knapsack เราทำการส่ง array ที่เก็บของ (น้ำหนัก), pointer ชี้จุดเริ่มของ array, weigh ที่ต้องการ แล้วจะ return เป็น array ที่เก็บคำตอบ (ans)
- เริ่มด้วยการ check กรณีที่เป็นไปไม่ได้ก่อน คือ weigh มันติดลบ หรือ start วิ่งเลย array ไปแล้ว ก็ return array size = 0 (no ans!)

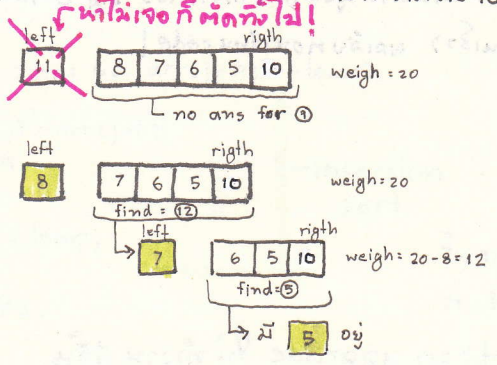


- จากนั้นเราก็วน loop หาว่าใน array เรามีเลขมีตัวที่มี "weigh" เท่ากับ weigh ที่ต้องการมั้ย ถ้ามีก็สร้าง array ans ขนาด 1 ช่อง แล้วเอาคำตอบยัดใส่ ans แล้ว return ans เลย



- ตัด array ออกเป็น 2 ส่วน (อีกแล้ว)

- ในส่วน right เรา assume ว่าเราจะหา weigh = weigh - arr[i] (เช่น weigth = 20, ถ้าเรานับ 11 ออกไปเป็นตัวแรก ก็แปลว่าใน right ถ้าเรานับของที่มี weigh = 9 ได้ เมื่อเอามารวมกัน 11 ก็จะได้ 20 พอดี, แล้วการจะหา ans ของ ช่วง right เราก็หาโดยการเรียก **recursion** โดย ขอบเขต index ที่จะหา จะเลื่อนไปทางขวา 1 (มันคือคล้ายๆ ตัด array เป็น 2 ส่วน) และ weigh ก็ลบออกด้วย น้ำหนักของ left
- แต่ถ้าเราหา weigh ใน right แล้วมัน return **array size = 0** กลับมา ... ก็แปลว่าในช่วง right ไม่มี weigh ที่เราต้องการเลย (ก็แปลว่าหาไม่ได้) → เราก็ตัดตัว left ทิ้งไปเลย (ก็มันหา ans ไม่ได้แล้วอ่ะ)



- เมื่อเราตัดตัวแรกทิ้งไป (ตัดทิ้งโดยการ i++ ใน for loop) ... รอบต่อไปเราก็ให้ left เป็น 8, แล้วก็หา 12 (weigh - 8 = 12) จากช่วง right (ซึ่งมันก็จะเรียก recursion อีก)
- เมื่อเจอ 5 ก็จะ return 5 กลับไป รวมกับ 7 แล้ว return กลับไปให้ชุดแรก ก็เอาไปรวมกับ 8 อีก ∴ ans = 8, 7, 5

Ex6. Merge Sort , Quick sort

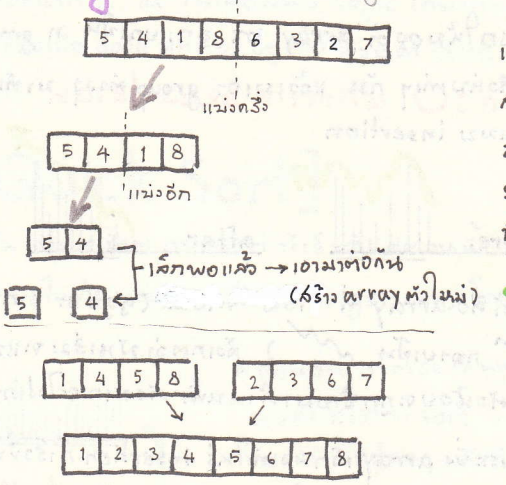
ทั้ง Merge. และ Quick ใช้หลักของการ divide and conquer โดยมี concept ว่า แบ่ง array ออกเป็น 2 ส่วน เอาทั้ง 2 ส่วนนั้นไปเรียงมาให้เสร็จ (sort ด้วยการเรียก recursion) แล้วค่อยเอาส่วนย่อยๆ ที่ sort แล้วกลับมารวมกันอีกครั้ง โดย...

Merge Sort : เอา array แบ่งครึ่งเลย (ได้ 2 array ที่มี size เท่ากัน) แต่เลขใน array ยังมั่วอยู่

Quick Sort : แบ่ง array เป็น 2 ตัว มี size ไม่เท่ากัน โดยมี pivot เป็นตัวกำหนด เดี๋ยวค่อยพูดใน advance sort!

• Merge Sort

อ่าน merge sort เพิ่มเติมในสรุป C ของเรา เรื่อง array (ประมาณหน้า 35)



เราใช้ Merge sort ด้วยการตัด array ออกเป็น 2 ส่วน ในแต่ละส่วนก็จะเอาไป recursion ตัดออกเป็นชิ้นย่อยๆ ลงไปอีก จนมันเหลือขนาดเล็กจน sort ได้ (เช่น เหลือ 2 ช่องเว้ย!) ก็จัดการ sort ชิ้นเล็กๆ พวกนั้นให้เสร็จ แล้วค่อยเอาชิ้นเล็กๆ พวกนั้นที่ sort แล้วมาต่อกันอีกที

- เอามาต่อกันโดยใช้ while loop (หรือวิธีอื่นก็ได้) (วน loop จนกว่า data จะหมด โดยในแต่ละรอบให้ทำการ check ว่าที่ index แรกของ array ย่อย 2 ตัว อันไหนที่น้อยกว่า ก็เอาไปใส่ array ใหญ่ ... ทำเหมือนหยิบ data จาก array ย่อยไปใส่ array ใหญ่ ทีละตัว (หยิบได้แค่ตัวแรก) จนหยิบหมดก็เสร็จ!)

✱ merge sort ทำงานได้เร็วกว่าพวก basic sort O(n log n) [คิดโดยมันต้องแบ่งครึ่งไปเรื่อยๆ ($\log_2 n$) และ: แบ่ง + เอามาต่อกัน (merge) ทั้งหมด n ครั้ง]

แต่ ข้อเสีย ของ merge sort คือ เปลือง memory มากมาย (ต้องสร้าง array ใหม่)

Advance Sorting.

Advance Sort.

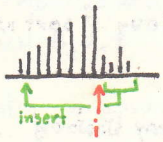
เป็น algorithm ขั้นสูง! ในการ sort ซึ่งมีประสิทธิภาพสูงมาก (ทำงานเร็ว big O เล็ก)... แต่เมื่อมัน advance มันก็ "ยาก" อยู่ ผลดี เวลา run (ทำงานเร็ว) ผลเสีย ตอนเขียน code!

- Shell Sort.
- Quick Sort.
- Radix Sort.

[Shell Sort]

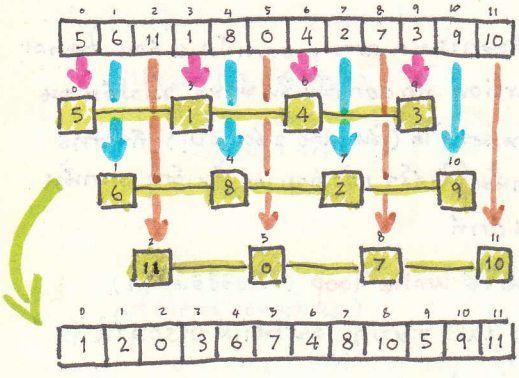
Shell sort เป็นการเอา "Insertion Sort" มา upgrade ให้ทำงานดีขึ้น

- เวลาเราทำ insertion sort... เราจะทำการหยิบ data ออกมาทีละ 1 ตัว แล้ววนloop หาที่ลง พีละ index ไปเรื่อยๆ จนเจอ ซึ่งการทำอย่างนี้ ถ้าเรามี data ที่มีค่าน้อยๆ อยู่ปลายแถวมากๆ จะทำให้การวนloop insert เสียเวลามากมาย เพราะ: มันต้องวิ่งกลับมาไกลมาก!

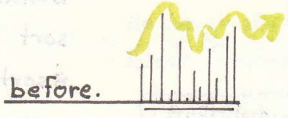


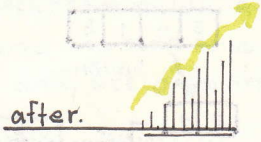
< ตัวน้อยๆ อยู่ปลายมากๆ ทำให้ insertion ทำงานนาน เพราะ: ต้องนำ data พวกนั้นย้ายที่ไปไกลมาก (สั่งหัวแถวเลย!)

concept : แบ่ง data เป็นช่วงๆ แล้วทำการ sort คล้ายๆ insertion... จัดให้รูปร่าง, ตำแหน่ง ของ data ใน array ดีขึ้นทีละนิด



ตอนแรกให้มองว่า array src ถูกแยกเป็น n group แต่ละตัวห่างเท่าๆ กัน แล้วเราเอา group ต่างๆ มาทำการ sort แบบ insertion





ผลที่ได้ คือ array ที่ "เริ่ม" จะเรียบ (ดูรูปนะ จนกลายเป็น) สังเกตว่ามันเริ่มจะแบ่งกลุ่ม ตัวน้อยจะถูกย้ายมาด้านหน้า ตัวมากจะไปเกาะๆ กันที่ด้านหลัง... วิธีนี้จะเร็ว การขับทั้ง array มาทำ insertion เพราะ: ยิ่ง array เล็กลงเท่าไหร่ insertion ก็เร็วขึ้น จากนั้น, เราก็เอา array ที่ได้มาทำการ sort โดยการแบ่งกลุ่มอีก แต่ แบ่งช่วงให้เล็กลง (นาย 3 โดยประมาณ) เช่น ในตัวอย่าง เราใช้ ช่วง h=3 ในตอนแรก การ sort รอบ 2 ก็จะได้ h=1 หรือก็คือ ทำเหมือน insertion เลย แต่การทำ insertion แบบนี้จะทำงานเร็ว เพราะ: data เกาะกลุ่มกันเยอะแล้ว (วิ่งหาที่ insert ไม่ไกล ก็ลงได้แล้ว!)

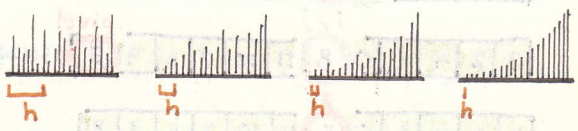
```
int i, j;
int temp, h=1;
while ( h<=arr.length/3) h=h*3+1;
while (h>0)
{
  for( i=h ; i < arr.length ; i++)
  {
    temp = arr[i]
    j=i;
    while ( j>h-1 && arr[j-h] >=temp)
    {
      arr[j] = arr[j-h];
      j-=h
    }
    arr[j] = temp;
  }
  h=(h-1)/3;
}
```

calculate h

insertion sort

การเขียน code ของ Shell Sort ส่วนใหญ่ จะใกล้เคียง insertion มากมาย แต่จะมีเพิ่ม h เข้ามา โดย h จะเป็นตัวบอก ช่วง ของ group

- ใน insertion เรา ขยับทีละ 1 index (ถ้าเทียบก็คือ h=1)
- ใน Shell เราจะขยับทีละ h ซึ่งค่า h นี่จะเปลี่ยนเรื่อยๆ (โดยประมาณ จะลดทีละ $\frac{1}{3}$)



ในการวน for sort 1 รอบ h จะเล็กลงเรื่อยๆ ตลายๆ กับว่า เริ่มด้วยการ sort นยายๆ แล้วทำละเอียดขึ้นเรื่อยๆ

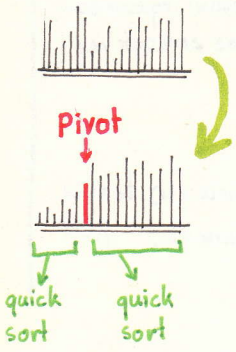
[Note] การทำ insertion sort ใน แต่ละ group มันไม่ได้ทำ group I ให้เสร็จก่อน แล้วค่อยทำ group II แล้วค่อย group III, แต่มันจะคิดทีละ n group พร้อมกันเลย คือ คิดคู่ที่ 1 ของ group I → คู่ที่ 1 ของ group II → คู่ที่ 1 ของ group III แล้ววนกลับมาคิดคู่ที่ 2 ของ group I ใหม่ (คือว่า มันจะคิดแต่ละชุดไปพร้อมๆ กัน)

Efficiency...

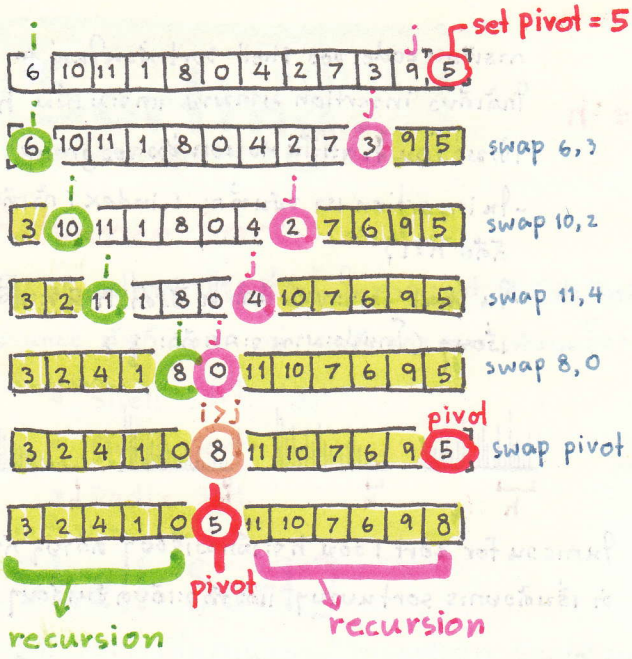
Shell sort เป็น sort ที่มีเวลาในการทำงานไม่คงที่ ขึ้นอยู่กับการ set ค่า h และการเรียง data ใน array ตอนแรก ... และ ในเมื่อมันมี Logic เดียวกับ insertion กรณี best case จะเท่ากับ $\Omega(n)$ ซึ่งจะเป็นเมื่อ data มันเรียงอยู่แล้ว ส่วนค่า average จะใช้เวลาประมาณ $n \log n$ ส่วนในกรณีแย่สุดหรือ worst case จะเท่ากับ $O(n^{3/2})$

[Quick Sort]

เป็น algorithm การ sort ที่เร็วที่สุดตอนนี้ ... ใช้ concept "recursion" โดยการแบ่ง array ออกเป็น 2 ส่วน (ไม่ต้องเท่ากันเหมือน merge sort) โดยจะมี pivot เป็นตัวแบ่ง



- ก่อนอื่น เราต้อง กำหนด pivot มาก่อน (ในที่นี้เราให้ เป็น ตัวขวาสุด (rigth index) ของช่วงที่เรา จะ sort ... แล้ว data ตัวไหน ที่ มีค่าน้อยกว่า pivot จะถูกย้ายไปอยู่ใน left index, ตัวที่มากกว่า pivot จะถูกย้ายไปอยู่ใน rigth index
- แล้ว เราก็ recursion left กับ rigth อีก



※ code ของ quick sort เขียนได้หลายแบบ แต่มี concept เหมือนกันนะ!

- ให้ตัว rigth เป็น pivot
- set pointer 2 ตัว ที่หัวกับท้าย (i กับ j)
- ในการทำ 1 รอบ เราจะใช้ while loop วนหาจนกว่าจะเจอตัวที่ มากกว่า pivot ในข้าง left (i++)
  และหาจนกว่าจะเจอตัวที่ น้อยกว่า pivot ในข้าง rigth (j--)
  จากนั้นทำการ swap 2 ตัวนั้น แล้ววนหาหัวน้อยกับตัวมากกว่าใหม่...
- ทำไปจนกว่า ค่า i จะมากกว่า j ให้หยุด แล้วทำการ swap ตัวที่ index i กับตัว pivot
- แล้วเราก็มาทำ quick sort ในฝั่ง left กับ rigth อีก!

* code ดูในชีทอีกละกัน!

Efficiency

best case → $\Omega(n \log n)$
average → ประมาณ $n \log n$ (กว่าๆ นิดๆ)
worst case → $O(n^2)$

[Radix Sort]

เป็นการ sort ที่เร็วมากวิธีนึง (อาจจะเร็วกว่า quick sort ด้วยซ้ำ) ... concept คือ เรียง data ใน array ที่ละหลัก เริ่มจาก หลักหน่วย → สิบ → ร้อย → พัน ...

0	0	9	6
1	4	2	8
2	9	5	9
3	9	0	0
4	8	5	8
5	2	1	1
6	6	2	5

9	0	0
2	1	1
6	2	5
0	9	6
4	2	8
8	5	8
9	5	9

sort ด้วย หลักหน่วย

9	0	0
2	1	1
6	2	5
4	2	8
8	5	8
9	5	9
0	9	6

sort ด้วย หลักสิบ

0	9	6
1	5	9
2	1	1
4	2	8
6	2	5
9	0	0
9	5	9

sort ด้วย หลักร้อย

เป็นการ sort ที่ concept ง่ายมากมาย แถมเร็วด้วย!

▶ เหตุผลที่ต้องเริ่มจากหลักหน่วย เพราะ: เราต้องเก็บตัวที่ มีผลต่อการ sort มากที่สุด (หลักที่มาก ที่สุด) ไว้ทำสุดท้าย หลักหน่วยไม่ค่อยสำคัญเท่าไหร่ เลย sort ก่อน

method getDigit()

```
public static int getDigit(int src, int d)
{
    src%= Math.pow(10,d);
    src/= Math.pow(10,d-1);
    return src;
}
```

เรื่อง Radix เราจำเป็นต้อง ดูด เลขเป็นหลักๆ ออกมา คิดหลายครั้งมา เวลาทำก็ควรเขียน method เอาไว้ เช่น getDigit(1453, 3) จะได้ = 4

Coding!

มีอยู่ 2 วิธี (เท่าที่เราคิดออกตอนนี้!...น: ^^)

แบบใช้ array นับจำนวนเลขแต่ละหลัก

```
int[] countNum;
int[] temp=new int[arr.length];
int l=1, i, j;
for(i=1; i<=l; i++)
{
   countNum=new int[10];
   for(j=0; j<arr.length; j++)
   {
      if(Integer.toString(arr[j]).length()>l)
         l=Integer.toString(arr[j]).length();
      countNum[getDigit(arr[j],i)]++;
   }
   for(j=1;j<10;j++)
         countNum[j]+=countNum[j-1];
   for(j=arr.length-1; j>=0; j--)
         temp[--countNum[getDigit(arr[j],
           i)]] = arr[j];
   arr=temp.clone();
}
return arr;
```

แบบใช้ queue เก็บตัวเลยเลย

```
Queue[] qNum = new ArrayQueue[10];
int i, j, k, l=1;
for(i=0; i<qNum.length; i++)
   qNum[i] = new ArrayQueue();
for(k=0; k<l; k++)
{
   for(i=0; i<arr.length; i++)
   {
      if(Integer.toString(arr[i]).length()>l)
         l=Integer.toString(arr[i]).length();
      qNum[getDigit(arr[i],k)].enqueue(arr[i]);
   }
   for(i=0, j=0; i<qNum.length; i++)
   {
      while(!qNum[i].isEmpty())
         arr[j++] = qNum[i].dequeue();
   }
}
return arr;
```

Efficiency ใช้เวลาทำงานประมาณ $O(kn)$ เมื่อ k เป็นจำนวนหลักที่มากที่สุด...จะเห็นว่า มันทำงานเร็วกว่า quick sort ซะอีก แต่ ข้อเสีย คือ มัน sort ได้เฉพาะ: "number" เท่านั้น เพราะ: มันไม่มีการ compare เลย sort พวก array ของ Obj. ไม่ได้!!

Shell	Quick	Radix
- concept เหมือน insertion แต่ ช่วงการ sort ไกลขึ้น	- แบ่ง array เป็น 2 ส่วนโดยใช้ pivot - มีการใช้ recursion	- sort ทีละหลัก - ไม่มีการ compare เลย - sort Object ไม่ได้!!
worst (ประมาณ) $O(n^{3/2})$ Avg. (ประมาณ) $O(n \log n)$	worst $O(n^2)$ Avg. $O(n \log n)$	worst (ประมาณ) $O(kn)$ Avg.