



# Declaring & Reference

การประกาศ var. แบบ array (มุตตัวแปร ซึ่งไปใช้ใน - - \*) โดยใช้ var. ทำได้โดย

## จุดเด่นของ array

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

• สรีวิ่ง var. ชื่อ  
name ซึ่งจะบุ่น  
ด้วยให้เก็บ int ที่เป็น  
array

• สร้างช่องเก็บ (ของ array)  
จุดเด่น length คือ  
(สูงมาก : มีตัวสั้น "new"  
และวิธีการสร้างอย่างง่าย)

- การกำหนดช่วง [=] มักจะกำหนดจากก่อน  
แล้วโอนค่าของทางขวาให้ทางซ้าย
- สร้างช่อง array
- name เก็บค่าช่องที่เพิ่งสร้างเมื่อครั้ง

## ► อธิบายเรื่อง Reference กันหน่อย

ปกติเวลาเราสร้าง variable มาเก็บค่า ตัว compiler จะเขียนตัวจัดการเรื่องการจดจำพื้นที่ใน memory ให้โดยที่เราไม่ต้องทำเอง

int x = 5;  
address = 1082  
↓  
In mem.

4 เช่นถ้าเราประคำส์ int มาตัวซึ่งในชื่อ "x" ... compiler จะไปจดพื้นที่ใน mem. ให้ 32 bit (ขนาดของ int ใน Java) ซึ่งจะจดที่ address ไหนก็ไม่รู้ (ในห้องอย่างที่ address 1082) compiler จะรีบินเด้งว่า นี่เราฝึกเด็กสิววว x ลักษณะไปยัง กับ address 1082 ... เมื่อเราลงกับว่าให้ x เป็น 5 มันก็จะไป set ค่าใน address 1082 ให้เท่ากับ 5 เก็บไว้เพื่อ ?

ตัวแปรแบบนี้ (ที่เก็บ data ของผู้ใดที่ address ของตัวมันเองเลย) เราเรียกว่า **Primitive data type** หรือว่าตัวแปรแบบ basic ๆ มาตรฐานภาษา Java (พิมพ์แล้วมีเส้นหัวใจ, บน EditPlus นะ)

? กันนี่... คิดว่า var. ลักษณะต้องเก็บ data ของมันไว้ที่ตัวมันนี่ ໄ้ลวันนี้จะเก็บที่ไหน  
▷ ก้มันเข้า var. ประเมา **Reference** กับ data ของตัวมันเอง ไม่ได้เก็บอยู่ที่ตัวมัน

Object obj = new Object();  
address: 2212  
↓  
In mem.

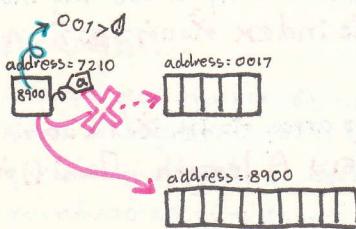
: new

4 ส่วนของการ Reference มักจะใช้กับ obj. (obj. มีรายละเอียดของ method และ var. ยังไงก็จะเก็บไว้ที่ตัวมันเอง) ... การสร้าง new ต้องมัน  
จะไปสร้าง Object ที่ต้องมีอยู่ใน mem. เสร็จก็จะนี่ เราเก็บ reference var.  
ชื่อ "obj;" ซึ่งเก็บค่า เมื่อ object ที่เพิ่งสร้างไป ... แต่ว่าต้อง Object  
ที่เพิ่งสร้าง (เพิ่งใหม่ไป) บางกรณีไว้ที่ตัวมันไม่ได้ บันเลขเก็บ "กันบุญ"  
ก็ต้อง address แทน

\* กันนี่... ลองสังเกตดูว่า เราสร้าง array เราต้องสร้าง new แม่ขอนกัน  
กับแปลงว่า array (เหมือนกับ Java) เมื่อ var. แบบ reference

① int[] a = new int[5];

② a = new int[10];



▷ ข้อดีของ array แบบ obj. (reference) ก็ต้อง เราสร้าง array ตัว  
ใหม่ไปโดยการ new array ตัวใหม่ขึ้นมา ... address ตัวเดียว  
ก็จะถูกโยนทิ้งออกไป และเก็บ address ของ array ตัวใหม่แทน

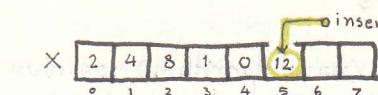


Java จะทำการ สัน mem. ใน  
system เมื่อ obj. (นั่นคือ array)  
ตัวนั้นที่ไม่ถูก reference จาก

var. ตัวไหนเลย... เช่นในตัวบล็อก เรา สั่ง ให้เก็บ  
array ตัวใหม่ mem. ที่ 0017 ก็จะ ถูกล้างทิ้ง!

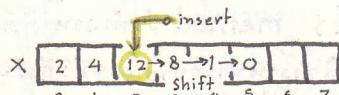
## Insert。

การ insert data หัวในมảng เช่น ให้มี array ลักษณะนี้ 2 แบบ ดังต่อไปนี้ insert เข้าไปที่ตำแหน่งที่ก้าง กับ insert เข้าไปที่ตรงกลางของ array (ตรงกับ index ของ data อยู่แล้ว) ...



insert หัวก้าง

→ จับบัดเข้าหน้าแรก (index)  
ห้ามยุ่ง... จะง่ายๆ



insertตรงกลาง

→ ก่อนจะบัดตัว เราต้องเลื่อน data หัวซึ่นให้ไป远 1 index  
แล้วค่อยแทรก data หัวใหม่เข้าไป

• move : 1

• move :  $1-n+1 \leftarrow$  นำราก data 1 หัว

insert new data

↑ จำนวน data ใหม่ array มองไว้ ↑ index ที่: นำราก data ใหม่เข้าไป ↓ เรียง data + index

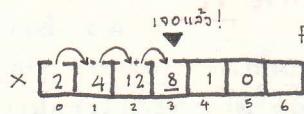
insert new data

\* การ insert ไม่ต้อง  
การ การประเมินตัวชี้บ่ง  
(Comparisons)

## Find。

การ find data หัวอยู่ใน array ก็เพื่อต้องการรู้ว่า data หัวนั้นอยู่ที่ index ที่เท่าไหร่ (หากไม่เจอ return -1) ซึ่งก็มี 2 แบบ ดังนี้ ถ้า find ที่ array ก็ยังไม่เรียงกันเรียงกันแล้ว (Unordered, Ordered array)

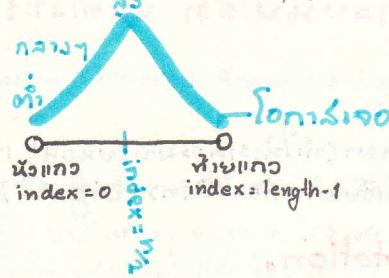
\* สำหรับ Ordered array ทางด้าน 'algorithm' ซึ่งวิธี binary search (โดยใช้เป็นหลัก เอามาใช้พอดี)  
<a href="สูตร Java ของ TA / หน้า 57.txt"> กลับไปอ่าน binary </a> link เลี้ยงกรุณา เชื่อกันด้วย!



• move : none!

• comparisons :  $\frac{n}{2}$  (avg.)

find for: 8



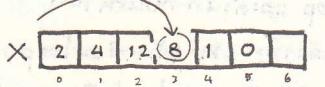
การ find เราต้องใจ naïve  
ไปที่ลับซ้อน จึงกว่าจะ: 100  
รอบส่วนใหญ่ แล้วเราเมิกะ:  
100 มากๆ  $\frac{n}{2}$  (ตรงกลางของ  
array)

find แต่ต้องการ compare ระหว่าง index นั้น เท่ากับ หัวที่เราหา อยู่รึเปล่า จึงไม่ต้องมีการขับ data หัวอยู่ใน array เทย (แต่ขอ data แล้ว check)

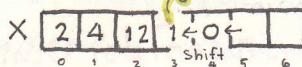
## Delete。

การ delete มันจะต้องกันขั้นตอนกับ insert .. ลักษณะนี้ 2 แบบ ดังต่อไปนี้ delete อยู่หัวก้าง กับ อยู่ตรงกลาง... ถ้าต้องการ delete data เราต้องหาหัวก่อนว่า data หัวนั้นอยู่ที่ index ไหน ซึ่งก็ต้อง Find

delete: 8 ก็ต้องหาหัวว่า 8 อยู่ index ไหน!



จัดการ shift หัวหน้า



จัดการ shift หัวหน้า

.. การ delete ต้องใช้กับ find และกับ Shift

จำนวน data ใหม่เรียบร้อยแล้ว delete

• move :  $1-n$  เด้งกับ insert 甫式: shift

• comparisons :  $\frac{n}{2}$  (avg.)

# Asymptotic analysis

เมื่อกำกับ "analyse" (วิเคราะห์) method ว่า กำลังงานใดที่ดีที่สุดดีที่สุดใน...

▷ การเขียน method ขึ้นมาตัวเอง ให้มีทำงานช้าๆ บ้าง อาจมีวิธีเขียนได้ดีกว่าบ้าง (นิยาม algorithm) เวลาเขียน เราต้องเลือก algorithm ก็ต้องรู้จะใช้ เช่น method sort array ก็มีหลายวิธี sort เช่น bubble, insert, radix ซึ่งมีความเร็วในการทำงานต่างกัน แต่ที่นี่ผลประโยชน์ก็คือ sort array

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

งานคูณทั้งหมด

- method นี้รันค่า  $g(n)$  ฯลฯ ... ถ้า: ผู้บอกรับว่า method ตัวนี้ ทำงานคร่าวๆ ได้ในหนึ่งก้าว นั่นว่ามี statement กี่ตัวทำงานกี่ก้าว
  - declare  $i$  กับ  $sum \dots 1$  ก้าว
  - วนloop  $n$  ก้าว บวกค่า  $sum$  กับ ...  $n$  ก้าว
  - return ... 1 ก้าว
- $\therefore$  method ใช้เวลาทำงาน  $2+n$ ,

❗ แต่เวลาเราตัดกอริธิ์... method นั้นจะไม่ได้มี statement น้อยๆ แบบนี้เท่านั้น  
เราเคยจะนับแต่ statement สำคัญ... โดยไม่สนใจค่าคงที่ (constant)  
method ข้างบนเราจึงนับได้ว่า ผู้บอกรับว่ามีstatement ใช้เวลาประมาณ  $n$ .

## Symbol

ได้มีการคิด สัญลักษณ์ขึ้นมา (ทำให้เราต้องมาเรียนมาก) เมื่อบอกว่า method ใช้เวลาทำงานประมาณนี้ และ<sup>แต่</sup>ใช้เวลาทำงานน้อยลง... ก็รับรองกันตื้อ **O(n)** (big O) เต็มๆ แล้วอีก 2 ตัวที่ควรรู้ คือ  $\Omega$  กับ  $\Theta$

## **O(n)** big O notation.

เรารู้ว่า  $O(n)$  ก็ต้องมีอย่างน้อยการวนloop ว่า method นี้ ทำงาน **กรณี最坏情况 (worst case!)** เช่น... method ข้างบนนั้น ทำเมื่อต้องวนloop  $n$  ก้าว ทุกรอบ อย่างน้อยชั้งที่ 1 ใช้เวลา  $n$  ซึ่งทำงานจน  $O(n)$  ใช้เวลา  $n$  (ตัด constant ที่  $1$  สนใจเฉพาะ ตัว variable)

method: 01

```
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;
}
```

- ที่นี่... method มีการใช้ if-else สืบกัน เส้นทาง กับการทำงานของ

โปรแกรม

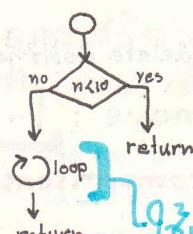
จะเห็นว่า เวลาในการทำงานต้องมากกว่า 2 ขั้นตอน

ตัวนักเรียนจะต้องวนloop สองตัวนักเรียนที่ return เอง

ดัง ตัวนักเรียนใช้เวลาเรื่อง  $n$  (ตัวนักเรียนloop)

หากตัวนักเรียนใช้เวลา คงที่ Constant ต้อง 1 ก้าว

(แต่ return สองตัวไว้)...

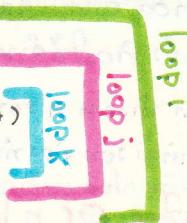


กรณี最坏情况 (worst case)

การทำงานเรื่อง  $n$  นี่ใช้  $1$  big O ซึ่งเท่ากับ  $O(n)$   
ไม่ใช่  $O(1)$

## method: 02

```
public int mto2(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+=n;
    return sum;
}
```



- เขียน loop ซ้อนๆ กันแบบนี้ให้เราจำง่ายดีมากที่สุด:

loop ล่างทุก层  $\times$  (คูณ) กัน

- loop k 9 ชั้นเวลา  $n$  ครั้ง

- loop j 9 ชั้นเวลา 5 ครั้ง

- loop i 9 ชั้นเวลา  $n/2$  ครั้ง

$$\therefore \text{loop } j \text{ วนกี่ } k 5 \text{ ครั้ง} = 5k = 5 \cdot n, \text{ ครั้ง}$$

$$\therefore \text{loop } i \text{ วนกี่ } j n/2 \text{ ครั้ง} = n/2 \cdot (j) = \frac{n}{2} \cdot 5 \cdot n, \text{ ครั้ง}$$

∴ 9 ชั้นเวลาที่ nn ต้อง  $\frac{9}{2} \cdot n^2$  ครั้ง แต่การถูก big O มองเห็น constant กับ co-efficient (ส่วนประ.สิกธ์) ก็  
เหมือน **big O เป็น  $O(n^2)$**  (ตัด  $\frac{9}{2}$  ที่เป็น co-efficient ทิ้ง)

## method: 03

9 ชั้นเวลา  $2n^2 + 500n + \log n$

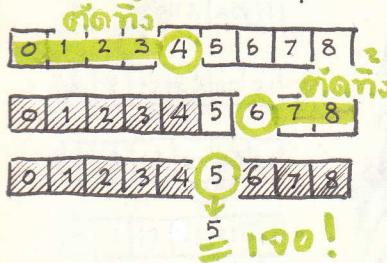
ก็ยังคงมาหากัน!

• เราจะนำว่า big O จาก method นี้ (ซึ่งต้องรู้ว่า ติดมากแล้วว่า 9 ชั้นเวลา เก่าเท่านั้น)  
ก้าวเข้าสู่สมการ喻ๆ 9 นัดๆ term กี่  $n$  มีกำลังมากที่สุด... 10 term นั้น  
มาตอนนี้มีนิยม big O

∴ **big O เป็น  $O(n^2)$**

## method: 04

binary search. ... เป็นการ search หาของใน Ordered Array (array ที่ sort มาแล้ว) โดยสิ่งเดียว  
ที่ต้องใช้ชี้ว่า ไม่เรียงตัว 7 จึงต้องให้น้ำหนักหัว



การใช้ binary search จะหา "ตัวกลาง" หรือ index กี่ อยู่กลางของช่วงที่เรา  
เช่น ก้าว 5. เราจะ array ขนาด 9 ช่อง ตัวกลางก็คือ index กี่ 4 ...  
ประมาณว่า index กี่ 4 (ก้าว 4) มีค่าต่ำกว่า หัวที่เราสนใจ คือเปล่าวว่า index  
กี่ 0 สำหรับ 3 ต้องน้อยกว่า หัวที่เราสนใจ คือเปล่าวว่า array นี้เรียงบัญชีแล้ว... เราต้องสุ่มๆ  
ครั้งหนึ่ง array แทน ซึ่งก็ทำหน้าที่เดินก้าว check หัวกลางของช่วงนั้นแล้ว  
ถ้ายังหาไม่เจอก็ต้องเราดูรอบ: สุดต้านซ้าย นี้เรื่องของช่วงนั้นนี่ ทำแบบนี้  
ไปเรื่อยๆ จนกว่าจะเจอตัวที่เราต้องการ การใช้ binary search ต้องใช้เวลาในการหาได้มากตาม  
ที่ array 1000 ช่อง การนาครึ่งแรกก็ทำให้เรา หัดซองที่ไม่ต้องนำไปใช้แล้ว ประมาณ 500 ช่อง!

รู้ว่า **big O เป็นอะไร?**

worst case ของมันก็คือ "เจอตัวสุดท้าย!" หมายความว่า เราหัดซองที่ไม่เจอ 1000 ชั้น  
เช่นต้องบ้าช้ำบานก่อนเรา 6

- การนา binary search ถ้าเริ่มต้นที่ 1 รอบ 2 จะเหลือ  $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$  จนเป็น 1 (หัวสุดท้าย)
- ลองติดกันบ้าง... เริ่มที่ 1, รอบต่อไปเป็น 2, 4, 8, 16, ... ที่ ก็คือ  $2^x \approx n$  (โดยประมาณ)
- เราจะได้  $x$  จาก  $2^x \approx n$  เราต้องเปลี่ยน  $\exp \rightarrow \log$  [  $b^x = n \rightarrow x = \log_b n$  ] ได้  $x = \log_2 n$
- ฉะนั้น worst cast ก็คือประมาณ  $\log_2 n$  (แต่เราต้องหันกลับ "base" ของ  $\log$  ให้อีก)

∴ **big O เป็น  $O(\log n)$**

**note:**  $O(1) < O(\log n) < O(n) < O(n^{1/2}) < O(n \log n) < O(n^{3/2}) < O(n^2) \dots$

$\dots < O(n^3) < O(2^n)$

method : 05

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

<ดู for ห้องอกด้วย จะ ตอน update คำมั่นไว้  $i^2$  i++ แต่เมื่อ  $i = i*2$

▶ คล้ายๆ กับ method 04... for i คำมั่นเพิ่มคำที่จะเท่ากัน  
อย่างที่เคยพูดไว้แล้ว คือใช้จำนวนครั้งเดียว  $\log n$  ... เท่านั้น

∴ for j นั้นไม่มีข้อบกพร่อง ทำงานทุกครั้งเดียว  $\log n$  แต่เมื่อ for i ที่ต้องทำ  
งาน  $\log n$  รอบเดียวแล้ว รอบต่อไปจะต้องทำงานอีก  $n$  จึงได้ว่า

∴ big O จะเป็น  $O(n \log n)$

## $\Omega$ , $\Theta$ 。

(ของแถม) ยังมี notation อีก 2 ตัวที่ควรรู้ด้วย  $\Omega$  (omega) และ  $\Theta$  (theta)

ที่ผ่านมาเราใช้ big O มากกว่า method นี้ที่ทำงาน最慢 "worst case" เมื่อเทียบกัน

- สำหรับ  $\Omega$  เราใช้เพื่อบอก "best case" หรือกรณีที่เวลาห้องที่สุดที่ดีที่สุด

- สำหรับ  $\Theta$  เราใช้เพื่อบอกว่า method นี้มี "ขอบเขตที่แน่นอน" คือไม่ว่า  
ทุกจุดค่าเมื่อเทียบกันจะมีผลกันที่เวลาตามตัว!

ห้องช่วยเชื่อว่า method 01 จะเป็น  $\Omega(1)$  และ  $\Theta(1)$  เพราะ ขอบเขต - ขอบค่าของห้องการห้องงานไม่เท่ากัน

\* ติด  $\Theta$  ง่ายๆ กับ  $O(n)$  กับ  $\Omega(n)$  กับ  $\Theta(n)$  จะเป็น  $\Theta(n)$

## Sorting basic

Sorting คือการห้อง data ใน array เรียงกัน

โดยเรียงแบบ • น้อย → มาก (Ascending)

• มาก → น้อย (Descending)

\* ส่วนในกฎที่ใช้ น้อย → มาก

- โดยการ sort นั้น จะห้องมี array มาอยู่แล้วซึ่งต้องห้องตัว

แล้วห้อง array มาเก็บไว้เรียก || ยกปะ: เจอกันของ array ได้โดย

1. primitive array เช่น int[] x, char[] c

2. Object array เช่น String[] str

はじめて～  
データ～



DATA-TAN ~

## ข้อแตกต่างระหว่าง primitive กับ Obj. o

ถ้าเรามี int x = 4; int y = 4; x กับ y ก็ถือว่ามีค่า "เท่ากัน"

ในขณะเดียวกัน array [4][4]... มีค่าเท่ากับ [4][4]... (ส่วนลับที่กันไว้ || แล้ว || ไม่共通)

แต่สำหรับ Obj. สมมุติว่าเราห้องลัง sort array of String โดยใช้ length เป็นหัววัด

[abc|xyz]... กับ [xyz|abc]... ถ้า abc แล้ว xyz จะมี length = 3 เท่ากัน แต่เมื่อเรามี data ทางด้านหัวชี้ ไม่เหมือน primitive ที่เรา || ยกไปสอง

∴ สำหรับ เว้นเดียว sort Obj. เช่น 2, 1, 3, 4, 5, 6 ก็ห้อง sort ให้มีเรียง 1, 2, 6, 3, 5, 4 7 แล้ว -  
1, 6, 2, 5, 3, 4 (6 กับ 5 บัญชี 2 กับ 3 ... สำหรับจะมีค่าเท่ากัน แต่ก็มีคนหนึ่งหัวชี้ตัว) นั่นจาก sort แล้ว  
ถ้าเรามี data ที่เท่ากัน แต่เมื่อเรามีคนหนึ่งหัวชี้ตัว หัวหนึ่งคนหนึ่งหัวชี้ตัวไป ... การ sort แบบนี้ทำให้ array ของเรามีความเสียหาย!? (stable sort)

## Simple Sorting

การ sort ล้วนๆ ก็เป็น algorithm มากน้อย sorting แต่ละอย่างก็เนี่ย: ก้ม data อย่างนึง แต่ตอนนี้จะพูดเรื่อง "Simple Sorting" 3 ตัว (ซึ่งเป็นก่อนเข้าห้องว่าง แต่ทำงานเร็วสุดของ advance sorting ไม่ใช่)

- Bubble Sort
- Selection Sort
- insertion Sort

**Swap:** การ sort ก็คือการ จัดเรียง data กันนี่ กอน. มัน ส่วนมาก compare (เปรียบเทียบ) data ให้แล้วก็ว่า ถูกเท่าทัน เว็บ compare และจะ ก่อตัว จับสลับกัน ซึ่งก็คือ swap ค่ากัน... ॥ะ: เท่า: swap มัน ใช้บ่อยมาก จึงแบบดอกมาเขียนเป็น method อะ: เชบ!

```
private void swap(int a, int b) <รัน "index" ของตัวที่ swap บน
{
    int temp = array[a];           // แล้วก็จัดการ swap ตาม index ฉะ!
    array[a] = array[b];
    array[b] = temp;
}
```

## [Bubble Sort]

bubble เป็นการ sort แนว basic ที่สุด... ดังนี้ for loop จับ array ที่: 2 index แล้ว compare ที่ 1 ถึง 0 (ถูกมากน้ำ)

8	2	3	2	12	40	21
↑						
8	2	2	3	12	40	21
↑						
8	2	12	2	3	40	21
↑						
8	2	12	23	40	21	
↑						
8	2	12	23	21	40	
↑						

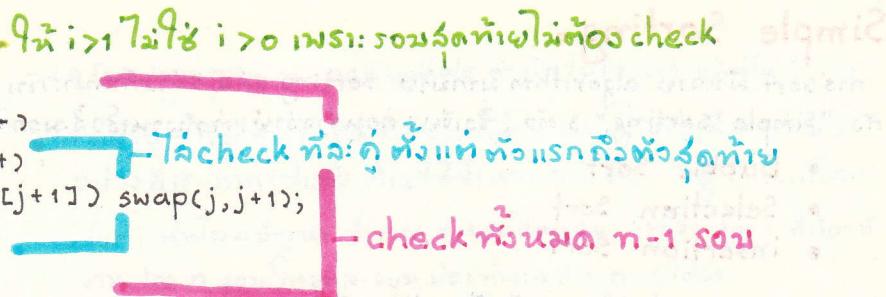
- < เช่น เปรียบเทียบถูก แล้ว ตัวที่ 2 แล้วก็ swap กัน หลังจากนั้น ก็เปรียบเทียบต่อไป ตัวที่ 1 แล้วก็ swap กัน ฯลฯ จนกว่า swap ไม่เกิดขึ้น 再 run ค่า 34.
- สังเกตดูว่า ในการ swap ที่: 2 แรก เราจึงได้ array ช่องที่ 2 ถูกหาย เว้นห่างจากตัวที่ 0 แล้ว ตัวที่ 0 ที่มีค่ามากกว่า จึงถูก swap ไปตัวที่ 2 แล้วก็ swap ต่อไป
  - บังคับๆ ॥นน 23 ก็ถูก swap ตัวไปปีกาก ก้าบไปรีบๆ จนเจตตัวที่มากกว่า ช่อง 23 (ดัง 40) ก็: เมื่อ array นัด 23 ไว้ตรงนั้น กันแล้ว ก็ swap ไปตัว 40 ไปปีกาก ก้าบต่อไป

!  
! กัน... การให้ check ที่: ถูก ॥นน นี่ แล้ว "รอบเดียว" มันชัวร์ไม่ทำให้ array ตัวนี่ sort เรียบร้อย เราต้องทำการวนloop ให้ check แบบอีก (สังเกตว่า การให้ check รอบนึงจะทำให้ array เรียง) เป็นเรื่องมากขึ้น ที่: นิด โดยเฉพาะ: ที่ index สุดท้าย จ: เป็นตัวที่มากสุดแล้ว!

? คำนึงถึง อย่างนี้ต้องให้ check ตึ๊ง แต่ทั้งๆ ก็ น้ำใจ แล้วต้องก่อตัว ซึ่งจะ: ชัวร์ว่า array นี่เรียบร้อย... ฉันก็ต้องการการที่รับเรื่องนี้ index สุดท้าย ถูก (ทำให้ ตัวมากสุดตามบัญชีของสุดท้ายได้) ก็ array ที่ n ช่อง ก็เท่ากับ ต้องใช้ n-1 รอบ เท่า: วนครั้งนึงจะ: ทำให้ ช่องที่ 1 ถูกต้องแน่นอนมากขึ้นเรื่อยๆ (และที่ต้อง -1 เท่า: ในการ check ครั้งสุดท้าย จ: เหลือ data ตัวเดียว... ฉันเมื่อ ตัวเดียวมันอยู่ถูก ตำแหน่งที่กันหมดแล้ว ถึง check ไป ลัพ ก็ต้องอยู่ตรงนี้แน่: วันที่รอบสุดท้าย ก็ เศษไม่ต้องทำ 0: ใจ ชัวร์ๆ

code : bubble sort

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



## อธิบายเพิ่มเติม...

สมมุติว่ามี arr [12 64 32 8 2]

รอบแรก [12 64 32 8 2]  $i$

รอบ 2 [12 32 8 2 64]  $i$

รอบ 3 [12 8 2 32 64]  $i$

รอบ 4 [8 2 12 32 64]  $i$

เสร็จ! [2 8 12 32 64]

↑ ใน code bubble ใช้  $i$  nested for loop (for ซ้อน for) 2 ตัว  
โดย  $i$  (for  $i$ ) ทำการ  $i-1$  check ทั้ง  $j$  แล้ว  $j+1$  swap  
ส่วน  $i$  วนลอก (for  $i$ ) กำหนดให้  $i$  คล่องๆ เป็น pointer ของ  
loop หนึ่งว่า รอบนี้  $i$  ที่ check คือตัวไหน

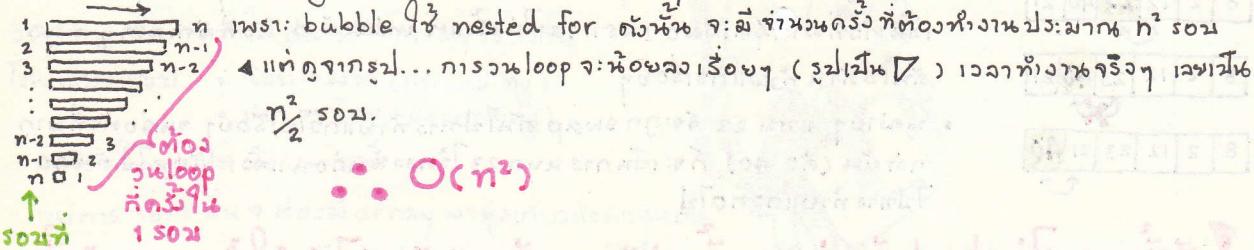
↑ ในการเขียน เราต้องรู้แล้วว่า for  $j$  รอบ  $i$  หันสุดท้าย มันกูก็ต้องซ้อน  
การวน check รอบ  $i$  ต้องไปที่ check ทั้ง  $i$  รอบสุดท้ายของรอบที่-  
แล้ว ก็พอ

↑ หมายความว่า การ check  $i$  ทั้ง  $i$  รอบ index สุดท้าย จะ วนลง  
เรื่อยๆ เราเลยใช้ for  $i$  เป็น loop แบบ decrement

## เรียบเรียง

## Efficiency...

bubble sort เป็น algorithm ที่ใช้หลักการร่วงหล่อแล้ว แต่ต้นการทำงานมันไปคิดง่ายดี เนื่องจาก



\* แต่ข้อเสียของ bubble คือมีรีกับข้างต้น... มันใช้เวลาทางตัวไม่รู้ว่า data ใบ array จะมีการเรียงยังไง  
แนะนำว่า ถ้าเราสร้าง array ที่ sort ไปให้ bubble รีกวนloop บันทึก: ต้องไม่เผลอเรียบเก็บกัน  $O(n^2)$  เลย!  
(ที่เนื้อหาแนะนำนี่เป็นรูป: loop ที่ใช้ทั้งหมดมี  $n-1$  รอบ for ซึ่งทำงานตามตัว)

# [Selection Sort]

Selection sort คือวิธีการดึง "เลือก" นี้เรียกว่า Selection ตามชื่อชันอ่: สังฆภานา: เลือก data ที่มีค่าต่ำสุดมาอยู่ข้างหน้า... หลักการ Selection เป็นการ sort ที่เน้นจัดความติดของคนงานมากๆ เช่นเวลาเราจราจ: เรื่อยๆ ง่ายๆ ก็จะซักกันข้างเรา ก็จะเลือกคนที่ต่ำสุดในทุกๆ จ: นับตั้งแต่คนที่ต่ำสุดกันไปเรื่อยๆ แล้วเดินไปป่วยที่หัว ก็จะกันหัวไป รวมๆ นี้ก็叫: ให้ตัวเลือกคนที่ต่ำสุดกันไปเรื่อยๆ

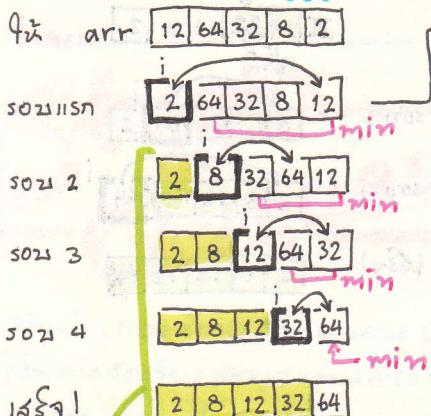
code : Selection sort

โจทย์: แก้ปัญหา:

```
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);
}
```

↑ สับตัวต่อไปนี้ด้วยตัวต่อไปนี้

อธิบายเพิ่มเติม...



เรียนรู้ว่า

- คล้ายๆ bubble ... for ตัวแรกที่น้อยที่สุด swap pointer (แต่เลื่อนจาก นั้น → หลังจาก)
- โดยใน 1 รอบ จะทำการหาตัวที่ต่ำสุดในช่วง index ที่เปลี่ยน ใช้ for loop เพราะ เราต้อง check ทุกตัวก็จะรู้ว่าตัวไหนต่ำสุด
- min เอามาไว้เก็บ "index" ของตัวที่ต่ำสุด

จุดเด่น

< รอบยัง ... for นาตัว min

min = 8(2)  
∴ i = 4

• ตาม concept... เราต้องเปลี่ยนตัวบวก min ที่อยู่ห่างกัน远 ไม่ได้เรียง (ตัวบน) แล้วเอามาใส่ไว้ตัวนั้นที่ข้าง array

• แล้วก็ต้อง swap เพราะไม่วันเราต้อง shift ทุก index ไปทางขวา (เช่นเวลา swap เริ่มต้น)

Efficiency...

มาลองคิดกันว่า Selection ใช้เวล่าเท่าไหร่ใน...

- ทุกรอบมีตัวห้องว่างนาตัว min ตัวแล้ว i ก็ต้องแลกของ array ตัวนั้นแล้วรอบกัน for i หัวนอกตัวยังไง มันจะได้  $\frac{n^2}{2}$

(คล้ายๆ bubble เลย)

< เลื่อนไปเรียนเที่ยวน้ำตก (แสดงช่วงการหา min ใหม่ต่อ: รอบ) ของเขบท 2 รอบๆ เพราะหลัง for i 1 รอบ ก็จะมีตัวถูกเพิ่มขึ้นทีละตัว

$O(n^2)$

# [Insertion Sort]

Insertion sort อาศัย concept ของการเอา data ไปแทรกใน index ที่มั่นใจว่าอยู่ แล้วคัดกรองให้เดียว กับความต้อง汎ของ index เพื่อ: หลักการของ insertion sort แต่ส่วนการห่วงงานกันดี๊ดี๊ selection จะทำการหา index ก่อน แล้วจึงมาใส่ใน ตำแหน่งที่แท้หนอน (เรียกว่า insertion ใช้เริ่มโดยการเอา data จากหัวแนบกันหนอน (i) ไปนำที่ลง หรือ แทรกนั่นแหละ: ณ index ที่มั่นใจว่า

code : Insertion Sort

```

for(i=1; i<n; i++)
{
    type arr j=i;
    int temp = arr[i]; ← ตัว data 00 กما
    while(j>0 && arr[j-1] >= temp) ← j=1
    {
        arr[j] = arr[j-1];
        j--;
        ถ้า array ตัวเดียว มากกว่า temp
        เรียง งาน ก็กล่าว
    }
    arr[j] = temp; ← insert
}

```

ค่าตัวเดียว → ฟังก์ชันเดียว จัดให้ดีๆ เลย

- ในการ insertion เราจะเริ่มที่ index ที่ 1 ไปเริ่มที่ 0 เพราะ: 1. จ. 0 ของ data ณ index นั้น ใช้มาใช้ในการซ้าย แล้ว index 0 มันอยู่ชั้นสุดบนๆ แล้ว เดย์มันต้องขยับไปไหน แล้ว 2...

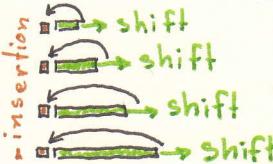
- เราจะ: ทำการนับ data ที่ index นั้น 00 กما ก็จะได้ 00 กัน แล้ว ทำการ shift data ณ index ก่อนหน้านั้น (หัวซ้าย) ทางขวา เรื่อยๆ จน data ตัวที่นับ 00 กما "หายไป" เราจะ: insert ผ่านล่างตรงนั้นแหละ:

- \* ตัวนับ data 00 กما แล้ว แต่ index ก่อนหน้านั้นจะกว่า 00 นัน ก็จะ: ไม่มีการเข้า while loop (ไม่มีการ shift) และ ก็จะ data ที่นับ 00 กما ยังคงลับเข้าไปต่อต่อ!

## Efficiency...

ก็พึ่งจะมา เราเจอก nested for 2 ตัวซ้อนกันทำให้มี  $O(n^2)$  ... สิ่งที่เราinsert ตัวจาก 2 ตัวแรก ต่อ มันเป็น for ซึ่ง while เลยต้องมาวิเคราะห์ loop while ก่อน เพราะ: มันเป็น 1 loop ที่ลื้ ทำงานครั้ง 1 แล้ว 2 แล้ว 3 แล้ว 4 แล้ว ... ดังนั้น! 2 | 8 | 12 | 32 | 64

- worst case ก็คือ ถ้าสุดของ while ก็คือ เราจะนับ data 00 กما แล้วทำ 00 ก็จะ: 00 ไปinsert ก็ต้อง ที่นับ 00 แล้ว หรือ ก็ต้องที่ index=0 (ต้อง shift ทุกตัวเลย) เมื่อเรา 00 แบบนั้น อุตสาห์ที่จะ while ไล่ก็ไม่ต่างจาก for เลย :: ก็ต้อง big O 00 แบบ for ซ้อนกัน



$$\therefore O(n^2)$$

## อธินายเพิ่มเติม...

ก็ arr [12 64 32 8 2]

รวมแล้ว: ← i  
[12 64 32 8 2]

so 2 2 I. [12 64 32 8 2] ตัวเดียว  
temp: 32

II. [12 64 -> 64 8 2] รวมยังอยู่ shift  
temp

III. [12 32 64 8 2] insert  
temp

รวม 3 [8 12 32 64 2]

รวม 4 [2 8 12 32 64]

เสร็จ!

- best case เนื่องจากว่ามันจะ while มันก็จะใช้เวลา O(n) ต่อไป ก็ต้องการที่ data ที่ index ก่อนหน้ามันให้มีค่ามากกว่ามันอยู่แล้ว ก็ต้องตัวมันไว้ต่อไปยังขั้นตอนถัดไป
- ∴ เมื่อไหร่เมื่อ loop while เลย ก็จะกล่าวว่า method วนลặp 1 loop for ขั้นตอน ซึ่งมี กวนส์ เศรษฐ์ (กรุณาชี้ array ที่ sort มาเรียบร้อยแล้ว) เป็นที่ ∴  $\Omega(n)$

## Summary: Sorting。

bubble	Selection	Insertion
<ul style="list-style-type: none"> <li>- ใช้การ check ที่ตัวแล้ว swap ครั้นแล้ว</li> <li>- วน nested for 2 ชั้น</li> <li>- ตัวมากสุด: เบิญไปทางขวา (ห่าง array) เว่อๆ</li> </ul> <p><math>O(n^2)</math> <math>\Omega(n^2)</math> compare <math>n^2</math>, swap <math>n^2</math></p>	<ul style="list-style-type: none"> <li>- ใช้การแทะบันวน O(n) นาที min</li> <li>- swap ตัว min กับ index ที่ i</li> <li>- สูงเวลา 100n นาที min ในรูป: ต้องวน for check ทุกตัว</li> </ul> <p><math>O(n^2)</math> <math>\Omega(n^2)</math> compare <math>n^2</math>, swap <math>n</math></p>	<ul style="list-style-type: none"> <li>- ใช้การนับ data 00 กว่า (<math>temp</math>)</li> <li>- shift data ทางซ้ายของมันไป เรื่อยๆ จนนาทีลงได้ (insert)</li> <li>- ทำงานเร็ว (กว่า 2 ตัวนั้น) เพราะไม่มีการใช้ while</li> </ul> <p><math>O(n^2)</math> <math>\Omega(n)</math> compare <math>n^2</math>, swap <math>n^2</math> Simple Sorting / จด.</p>

## Stacks & Queues

Stack (กองช้อน) กับ Queue (คิวสำลับ)  
เป็นการจัดเรียงข้อมูลแบบ先进后出 ใจไปที่ "ลิสต์"  
ทางการ input data เท่านั้น...

### Stack

- ↳ Last in, First out
- สิ่งที่เข้ามาที่หลังสุดจะออกไปก่อน
- เนื่องมาจากกองนั้นหันส่องช้อนๆ กัน เล่นที่ลิสต์ท้ายกูกันบ่อก

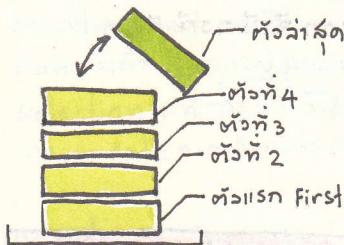
### Queue

- ↳ First in, First out

- เข้ากันดีกัน
- เนื่องจากการเข้าคิว ซึ่งมี (อย่างง่ายก็ไม่เคยเห็นไหม?)

\* ในการเรียน Stack กับ Queue... เราต้องมีการเก็บ data ไว้ในลิสต์ ต้องซึ่งกันและกันไม่ใช่ array อีกแล้ว ^^!

# [Stack]



เวลาเรียกใช้ stack จะมี method ดังนี้คือ...

- push ใช้เพื่อ add ของแท้ stack (ตัวที่แน่นบนสุด)
- pop ใช้เพื่อ remove ของ (ตัวล่างสุด) และ return กลับไปปั๊วะ
- peek ใช้คล้ายๆ pop แต่ return ตัวล่างสุดกลับไป แทนที่ remove
- isEmpty ใช้ check ว่า ตอนนี้ stack ของเรามี 0 หรือยัง มี
- isFull ใช้ check ว่า ช่องเก็บใน stack ตัวนี้เต็มรึยัง ( เพราะใช้ array )

**size** เอาไว้ check ขนาดของ array

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; }  
}

กำหนดให้ arr เป็น pointer ซึ่งมากกว่า ตัวแหน่ง  
 ลักษณะที่ add data ทำไว้คือ index ใน array ของ Object ไว้เก็บข้อมูล

} constructor ทำการ set size และ new array  
 ตอนนี้ ให้ i เป็น -1 เมื่อ ยังไม่ล้วง data เจาะหักตัว

} add data ทำไว้ก็ตามเนื่อง "ตัดจากตัวที่แล้ว"  
 \* ตัวที่แล้วเก็บที่ index i

} remove data โดยการเลื่อน i (ไม่ได้ลบจริงๆ ก็แล้ว  
 ไม่ว่าจะลบไว้ !)

} return ตัวล่างสุด

} ตรวจสอบว่า Stack Era ว่า (ไม่มี data อยู่เลย) หรือลอกโดยตรวจสอบ  
 โดยการ check i ซึ่งหันหน้าที่เป็น pointer

} check คล้ายๆ isEmpty ... ดู array ว่า ต้มกันหมดเมื่อ  
 i ซึ่งมาถึงตัวสุดท้าย (index ที่ size-1)

**arr[++i] คือ... ?**

$++(i++)$  เป็นการ increment variable ตัวหนึ่งขึ้น 1 โดย  $++$  หมายความว่า ให้นำมือ 1 นับ var. ให้ไป แล้วค่า - 1 นับอีกครั้งหนึ่ง increment แต่ที่  $i++$  แตกต่างคือ คือ ซึ่งเวลาของ การ increment

- $i++$  หมายถึง เติบต่อ i ไป 1 ครั้งก่อนแล้วค่อย increment
  - $++i$  หมายถึง ก่อนที่เติบต่อ i ไป ก็ต้องเพิ่มให้ increment ก่อนแล้วค่อย i ที่ increment แล้วไปใช้
- เช่นๆ  $i = 0$

↳  $arr[i++]$  จะนำ去กึ่งการเติบต่อ arr ตัวที่ 0 แล้วนำออกจากนั้นค่อย increment i ไป 1.  
 ↳  $arr[++i]$  จะนำไปกึ่ง increment i ให้เป็น 1 ก่อนแล้วค่อยใช้ ∴ ต้องถึง arr ตัวที่ 1

## Stack និងចំណាំទីតាំង

- Web browser ប្រើ back / forward ... ការកត់ url link វិញដោយពេលថ្មី នៅព័ត៌មាន ប្រើជំនួយ stack ដែលកត់ back ក្នុងការកត់ថ្មី ដូច ខ្លួនរាយក្រឹត ដឹងដំណឹង និងការបង្កើតសម្រាប់ការការពារការិយាល័យ។
- undo. ការកត់ថ្មីដែលការពារការិយាល័យ។
- Parsing (check syntax) ដែលនឹងការលេចតាមដំឡើង ( ), {}, [] តើកាត់ និងចំណាំសម្រាប់ការការពារការិយាល័យ។

## Postponing.

- នឹងការការពារការិយាល័យ សម្រាប់ការលេចតាមដំឡើង "operator" ទៅគ្មាន៖ ការអនុវត្តការការពារការិយាល័យ

- Infix ការកែតាមលេខជាអឺវ៉ូវ៉ូ: number + a + b

- Prefix ការកែតាមលេខជាអឺវ៉ូវ៉ូ: number + z + a + b (នមាយទាមរបៀប a + b)

- Postfix ការកែតាមលេខជាអឺវ៉ូវ៉ូ: ab + (នករាយទាមរបៀប a + b)

※ នៅលើ computer ទៅតុកសម្រាប់ការប្រើប្រាស់ នឹងចំណាំទីតាំង: ចំណាំទីតាំង Infix → Postfix

### How to Infix → Postfix

Number (operand)	copy លេខដែលត្រូវចំណាំទីតាំង ទៅ output ឡើ
(	push ទីតាំង Stack វិញឡើ
)	pop data ឬការណា Stack វិញទៅឯង នាមទាំង ( ) និងការណា (pop ទៅឯង output)
Operator	<ul style="list-style-type: none"> <li>- ការណា stack វិញទៅឯង (Empty) ឬតុក</li> <li>- push op សង្ឃឹមទីតាំង stack វិញឡើ</li> <li>- ការណា data ឬ push ទីតាំង stack វិញឡើដើរ</li> <li>- ការណា ( ) ឬដឹងពី push op ទីតាំង stack វិញឡើ</li> <li>- ការណា data ឬ push ទីតាំង stack វិញឡើដើរ និង op ឬដឹងពី ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក</li> <li>- ការណា: * , / , % ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក</li> <li>- ការណា: + , - ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក</li> <li>- ការណា: &lt; , &gt; ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក</li> <li>- ការណា: = ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក</li> </ul>
No more!	pop ទៅ Stack ទៅឯង output ទៅឯង នាមអនុវត្ត

$$3 + (2 - 7 / 5) \times 2 - 4 + ((9 - 4) \times 8)$$

$$3 + 2 - 4 + 9 - 8 \times 5$$

ការរំភោល់ infix ទៅ postfix  
ទៅការការពារការិយាល័យ String បានការងារ: រំភោល់  
(charAt(0) តើ charAt(1)) ឬការការពារការិយាល័យ ទៅការការពារការិយាល័យ ទៅការការពារការិយាល័យ ឬការការពារការិយាល័យ ទៅការការពារការិយាល័យ ឬការការពារការិយាល័យ ទៅការការពារការិយាល័យ

\* output តួត ពំលេចពារិយាល័យ

\* stack តាមវិធីការណា ឬ:

- ( ) ឬដឹងពី ទៅឯង ដើរ
- + - ឬដឹងពី ទៅឯង ដើរ
- < > ឬដឹងពី ទៅឯង ដើរ
- = ឬដឹងពី ទៅឯង ដើរ

① ការណា ( ) ទីតាំង stack

② ការណា + - ទីតាំង stack ដើរ

③ ការណា < > ទីតាំង stack ដើរ

ការណា = ទីតាំង stack ដើរ

ឬដឹងពី ទៅឯង ដើរ

សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក ឬដឹងពី សាមសិទ្ធិ និងតាមការស្វែងរក

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

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

คณิตศาสตร์ ก้าวต่อไป

คณิตศาสตร์ ไม่ใช่เรื่องเดียว ก็เป็น

number  $\rightarrow$  oon output เลย

Op  $\rightarrow$  stack ว่าง (Empty) push 9进 stack เลย  
(  $\rightarrow$  9进 stack เลย)

number  $\rightarrow$  oon output เลย

Op  $\rightarrow$  หัวล่าสุดใน Stack เป็น ( , push op进 stack.

number  $\rightarrow$  oon output เลย

Op  $\rightarrow$  หัวล่าสุดใน Stack เป็น op ที่มากกว่า, push op.

number  $\rightarrow$  oon output เลย

)  $\rightarrow$  pop stack oon output งานเงื่อน ( ก่อนบุญด้วย )

Op  $\rightarrow$  หัวล่าสุดใน Stack เป็น op ที่มากกว่า, push op.

number  $\rightarrow$  oon output เลย

Op  $\rightarrow$  op进 stack ร: ดับงู/เท่ากันมั้ง, pop 7进 output

number  $\rightarrow$  oon output เลย ( รอ งานไม่ตรงกันอยู่ )

Op  $\rightarrow$  op进 stack ร: ดับงูเท่ากัน, pop 7进 output เลย!

(  $\rightarrow$  9进 stack เลย

(  $\rightarrow$  9进 stack เลย

number  $\rightarrow$  oon output เลย

Op  $\rightarrow$  หัวล่าสุดใน Stack เป็น ( , push op进 stack

number  $\rightarrow$  oon output เลย

)  $\rightarrow$  pop stack oon output งานเงื่อน ( ก่อนบุญด้วย )

Op  $\rightarrow$  หัวล่าสุดใน Stack เป็น ( , push op进 stack.

number  $\rightarrow$  oon output เลย

)  $\rightarrow$  pop stack oon output งานเงื่อน ( ก่อนบุญด้วย )

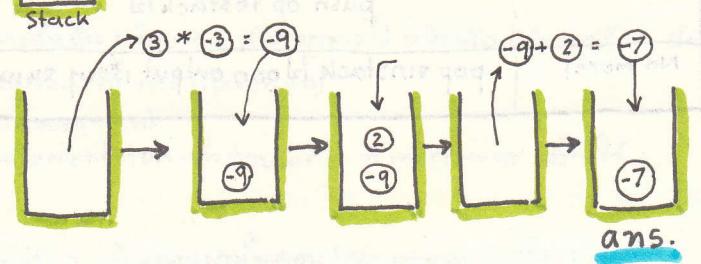
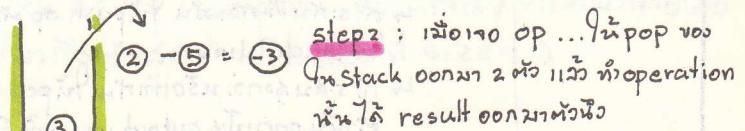
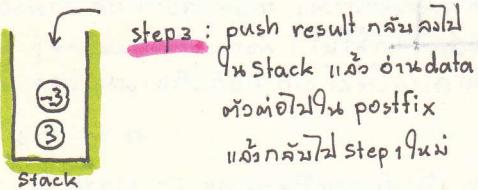
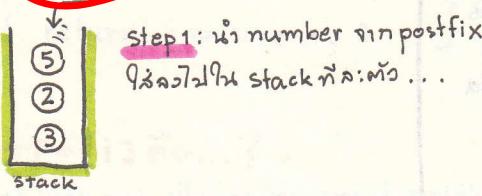
none! หลุดแล้ว ( รอปุ๊ ! ) pop ทุกตัว

Outstack oon output 9进 หมด !!

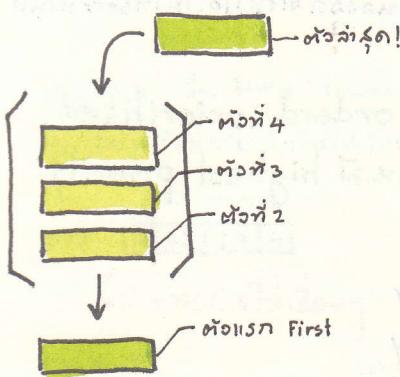
## Computer ก็สามารถเรียนรู้ไปแบบ postfix。

เวลาเราเขียน code statement เราจะเขียนในรูป infix ซึ่งเวลา com. จะเอาไปตัด บันท้องและpush ให้ 9进 ไป postfix กันเลย... เช่น

$$3 * (2 - 5) + 2 \rightarrow 325 - * 2 +$$



# [Queue]



method ของ queue...

- enqueue insert data เป็นสมาชิก queue

- dequeue (เหมือน pop) นำ data ออกจาก queue

- peek

- isEmpty, isFull } เหมือน stack

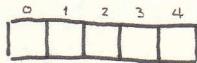
- isEmpty, isFull }

## Concept การเขียน algorithm Queue.

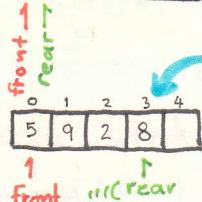
- การเขียน class queue โดยใช้ array เป็น basic โครงสร้างข้อมูล คือ array ที่มี pointer ชื่อ index สำหรับเพิ่ม เก็บ ลบ หรือ ดูตัวที่ i ของ queue สำหรับ enqueue ห้องเวลา dequeue ก็ ห้อง array [0] ห้องแรก กับ ห้อง [queueSize - 1] แล้วห้อง shift data ที่หลัง เงยบขึ้นมา 1 index

\* แท้ไม่ต้อง...!? เรา: ถ้าเราเขียนแบบนี้ method "dequeue" ของเรายังเป็น method เดิมที่มี big O เป็น  $O(n)$  (method ที่อยู่ใน stack แล้ว queue มันเป็น  $O(1)$  แน่นอน!) เรา: เราต้อง shift data 'เดือน' ทั้ง array

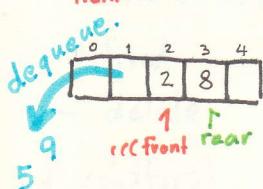
- เราคงจะเขียนให้โดย มอง array เป็น "วงกลม" แล้ว pointer 2 ตัวซึ่ง



- ตอนแรกรับว่า array ไปแล้ว โดยมี pointer ชื่อ front ที่จุดเริ่ม 2 ตัว

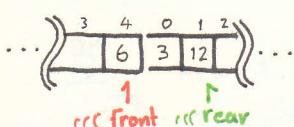


- เมื่อ add ของเข้ามา ก็กำหนด pointer rear (เข้าหนึ่ง)  
ก็จะเป็นตัวบวกกว่าต่อ add คราวนี้ และ rear++



- เมื่อ dequeu ออกไป ก็จะให้ pointer front เป็นตัวบวกกว่า จะ return ตรงนั้นกลับ แล้วมันก็จะ front++

\* การที่ pointer ยังคงเดินต่อไปจนกว่า data ที่ต่อไปไม่ front เกิดรักษาไว้ ก็เนื่องจาก delete ไปแล้ว!



- สำหรับการที่ pointer ที่ไปถัดคล้าย array แล้ว มันจะติดเนื่องกับ ว่า วนกลับไปที่นั้นแล้วต่อ แนวๆ กันหนา (ว้าว哉!) เนื่อง K-map

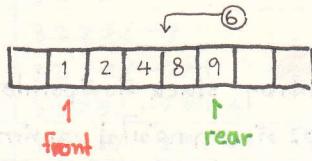
## Priority queue。

ลัง queue ก็มีการ สั่งตัวไว้ได้! หรือก็คือการที่ data หัวที่เข้ามาแล้วจะ: ถูก一页ไป insert ที่ใน กิตติ์ใน array

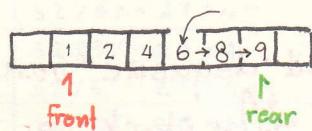
- Priority queue ก็คือ การที่ใน queue น้ำย่างเป็น ordered (prioritized) โดยจะมี key value เป็นตัว check ว่า data ตัวไหนเป็น highest priority

โดยรูปแบบของ Priority queue จะมีดัง 2 แบบดัง...

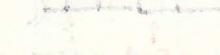
- Ascending ให้ key ว่า ตัวที่ smallest จะเป็น highest priority
- Descending ให้ key ว่า ตัวที่ biggest จะเป็น highest priority



การเรียง priority จะทำให้ data หัวแล้วที่เพิ่ง add เข้ามา จ.ไม่เข้าไปอยู่ที่ index rear เนื่องจาก แท้จริง: หัวที่ insert (ติดตัวอยู่ เรื่อง insertion sort อยู่)



การแก้ไข queue ลดลง ลดลง จนหมดจังหวะ queue หายไป



# Linked List

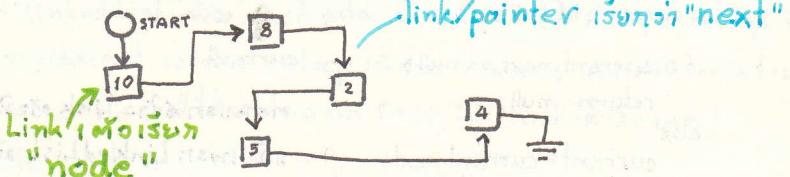
\* ចាត់បន្ទាន់លើកស្រួល ងាយស្រួល មិនមែន  
អ្នក 58-60 តែងទេ !

LinkedList នឹង Data Structure ឬបាន Obj. ទីនៃនាំងការកែង data ដែលប្រើបានបានរាយ...

នៅរៀង: data ធំពីខ្លួន កែងនូវ Linked List ទីនៃកែងនូវ នូវ memory ដែលតែងការណ៍បាន array



គឺជាដំណឹងកែងនូវ



គោលរាលីថា array ការកែងនូវ data នូវចំណាំនេះ នូវចុះតូលីយ index (ដែលចុះតូលីយ arr[4] កីឡុយចុះតូលីយ data នៃ array នៅ + 4 (ឬបាន assembly) ...) នៅក្នុង linkedlist data នូវតែងតាំង នូវក្រសែងកែងនូវ នូវ memory (ឬបាន random) ឱ្យលើលក្ខុងមិនសងគេង address រាលីថា ធំពីខ្លួន កែងនូវទរពីខ្លួន !

## សំគាល់របៀបនៃ Link ។



- Object data នៅក្នុងកែងនូវខ្លួន

- Link next នៅក្នុងកែងនូវ address ធំពីខ្លួន

LinkedList កីឡុយការទោះ Link នៅក្នុងគាន់កែងនូវ នៅក្នុងកែងនូវ L.L. នូវក្រសែង នូវ index ឬ size នៅក្នុងកែងនូវ Link នូវមាត្រាតំនើន ឬថា "first" នៅក្នុងកែងនូវ Link នៅក្នុងនៅក្នុងកែង... នៅក្នុងកែងនូវ next នូវមិន កិឡុយ null (ឬមិន នូវតែងតាំង ! )

## basic LinkedList.

- addFirst, addLast

- find

- delete

### • addFirst, addLast

នៅក្នុងកែងនូវ LinkedList នៅក្នុងកែងនូវ data នៅក្នុងកែងនូវកែងនូវ តូលីយ first តូលីយ null

កែងនូវកែងនូវ add data នៅក្នុងកែងនូវកែងនូវ នៅក្នុងកែងនូវ newData.next = first.next តួនតែនូវ

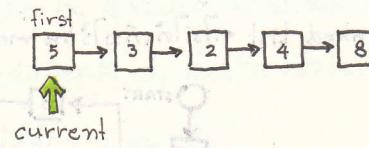
កែងនូវកែងនូវ newData នៅក្នុងកែង

កែងនូវកែងនូវ addLast ... នៅក្នុងកែងនាមីកែងកែងកែង នៅក្នុងកែងនូវ index នៅក្នុងកែងនូវ នៅក្នុងកែងនូវ នាមីកែងកែង (តួនតែនូវកែងកែងកែង next = null) នៅក្នុងកែងនូវ Link នៅក្នុងកែងនូវ នាមីកែងកែង

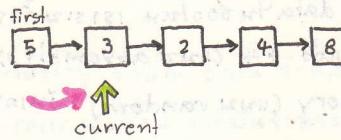
## • find

อนึ่ง LinkedList การสร้าง class Link เราอาจให้ Link มี properties เพิ่มขึ้นชื่อนึงคือ id ซึ่งเอาไว้ เช็ค key word ว่า เราต้องการหา Link ที่มี key word ตรงกับที่เรา

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



พอกะนี้เราต้องการหา Link ที่มี id นี้ หมายความว่า ตอนนี้เรา find ยังไม่ได้ในLinkedList แต่ใน index

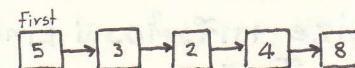


ตอนนี้... เราใช้ while วนloop หา Link ที่มี id ตรงกับ key กันเรามา แล้วในกระบวนการนี้ ก็ต้อง check ก่อนว่า ลักษณะของ current นั้น ถ้าไม่เป็น null (null) ก็จะป้องกันไม่เจอ แต่ถ้าหันมีต้อง ก็จะบัน current ไปสู่ตัวต่อไป

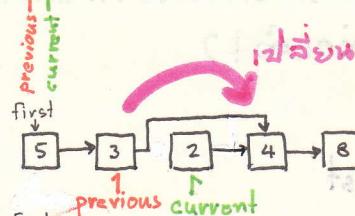
## • delete

การลบ Link ที่หันมือออกจาก LinkedList เราต้องนึกก่อนว่า Link หันมืออยู่ตรงไหน (ก็เดือนห้องที่การ find ที่ก่อน)

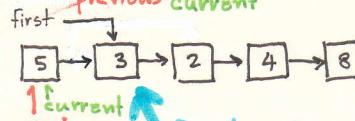
```
Link current = first;
Link previous = first;
while (current.id != key)
{
    if (current.next == null)
        return null;
    else
    {
        previous = current;
        current = current.next;
    }
}
if (current == first)
    first = first.next;
else
    previous.next = current.next;
return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



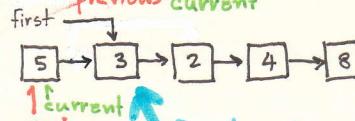
src



delete  
ตรงกว่า



delete  
first



first ขยับ 1 ตัว

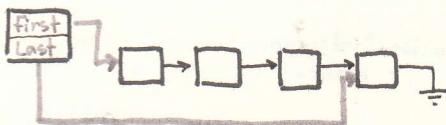
เมื่อเรา find ต้องที่ delete ออกจาก list เมื่อ เราจะต้องการลบหันมือออกจาก List โดยการเปลี่ยน next ของตัว previous ไปสู่ตัวต่อจาก current เลย (โดยนำ current ไปเชื่อม)

\* เราต้องมี previous เป็นpointer ของตัว current นั้น!  
เราต้องสับ set ตัว next ของตัวก่อนหน้า current ใหม่!

แต่การ delete ต้องระวังอยู่เรื่องนึงคือ เมื่อเราลบ first ล่ะ  
เราต้องอยู่ที่ first previous กับ current ขึ้นอยู่ที่เดียว กันอยู่... เราจะพบว่า ลักษณะ first = first.next เลย

**[Note]** อนึ่ง delete นี้เรื่อง Data Structure ส่วนใหญ่ จะไม่ได้มุ่งเน้นการ "ลบ" data ที่ 72 แต่จะเน้นการ "เลิกหันใจมัน" ซึ่งมากกว่า

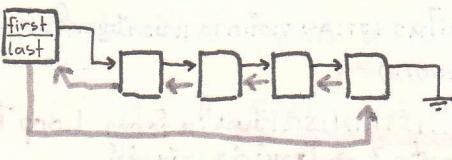
## Double ended List.



เขียนรูป || บันทึกย่อ code Linked List ให้ node แรก นั่นคือ first มีการ Link ไปยัง node ต่อไป 2 ลักษณะ  
สัก Link ไปยัง ตัวแรก first และ ตัวสุดท้าย last

**ข้อดี:** เวลาเรา add data ให้ Linked List เพิ่ม... ต้อง add เข้าที่ first ก็ไม่ใช่ปัญหาต้อง big O เท่าไหร่เพรฯ; แต่ O(1) แต่ถ้าเรา addlast เรายังต้องวนloop เพื่อมาห่างจาก ผู้เดียวเวลาจะมากกว่า การที่เราเมื่อกำหนด Link จาก หัวไป → หางกลับ จะทำให้เราต้องจาก first ไป last ได้เร็วมาก!  
ซึ่งก็ให้ในรูปของการ add และ delete ตัว last.

## Double linked List.

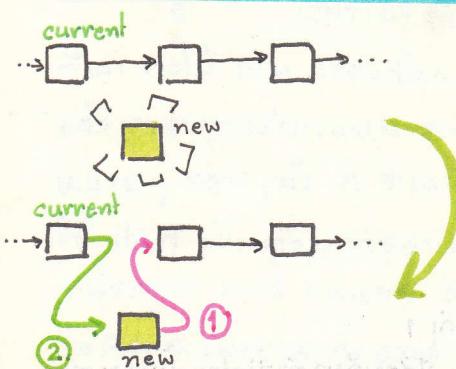


เขียนรูป || บันทึกย่อ Linked List ก็ต้องรูปแบบนี้มากกว่า  
เพรฯ; ผู้เดียวสามารถ ร่องไป ร่องมาได้โดยที่... โดยเราต้อง  
เพิ่ม properties ชื่อว่า "previous" เข้าไปใน class  
Link (next ก็เป็นตัวต่อตัวกัน, previous ก็เป็นตัวกัน)

**ข้อดี:** ทำให้การ insert / เรียกใช้ data ของ List ของเรามากกว่าง่าย ส่วนใหญ่เพรฯ; อันดับที่ 2 ใน  
กรณี (กรณีที่ array อยู่แล้ว!)

**ข้อเสีย:** เวลา add / delete จะทำได้ยากมาก!! ลองคิดดูดี ต้อง add กับ delete แบบ basic  
ธรรมดานะ ก็ต้องรู้ว่า next ต้องต่อตัวกันต่อไปนั่น (next), แล้วนั่นเมื่อเพิ่ม previous ขึ้นมา ต้อง  
code ใหม่ method ต่างๆ ก็จะหาย + ซับซ้อนมากกว่า!

### การ add Link เข้าที่กลาง Linked List



▷ การ add ตรงกลาง (สมมุติว่า add "new" ต่อจาก current)  
ก่อนซึ่งเราต้อง...

1. บอกว่า new.next = current.next เพรฯ; คำแนะนำที่มันจะ:  
ต่อต่อต่อต่อต่อต่อต่อต่อต่อต่อต่อต่อต่อต่อ...

current, เช่น add แรก วันนี้: กว่างเข้มต้องที่ต่อจาก new

2. เปลี่ยน current.next = new ให้ตัวต่อจาก current  
ให้เป็น new ก็ต่อ List ภูมิที่มี new มาแทนกวางไว้!

# Recursion

\* ตัวนี้เพิ่งสร้าง Java ของเรางานที่ 52-55 แล้ว

"Recursion" หรือ "Recursive" คือการเขียนฟังก์ชัน method (หรือ function) หรือก็คือตัวเอง ให้สามารถเรียกตัวเองได้

## Divide & Conquer

เป็น concept การแก้ปัญหาแบบแบ่งส่วน ซึ่งมีเน้นหลักของ Recursion อย่างเดียว

"แล้วแบ่งส่วนต่อ...? ก็มีเพื่อ...?"

ปัญหานำมา ตัว อาจมีน้ำหนักที่ใหญ่จนเราตัดหัวเดียวไม่ได้ เรายุบ จัดการ แบ่ง ปัญหานั้นๆ ออกเป็น ชิ้นๆ ลงๆ ๆ ต่อ (แบ่งไปเรื่อยๆ จนกว่าจะตัดออก)

**[ข้อบอกร]** การเขียน Recursion ส่วนใหญ่... เราสามารถเขียนใน form Loop ได้ > เช่น: ... แต่จะเขียน Recursion ยังไง? → โดยสั้น, พฤกษา

## Form หลักของ Recursion

```
type methodName(int n)
{
    if (n == กรณีเล็กสุด) return ลักษณะ;
    else
    {
        // แบ่ง ที่มีส่วนเล็กๆ เรื่อน n-1
        return methodName(n-1);
    }
}
```

● รูปแบบทั่วไปของ Recursion ดังนี้: เมื่อ check if-else ... ต่อถ้ามีเงื่อนไขที่เกิดขึ้นแล้ว ก็ return ค่าตอบแทน แต่ถ้ามีเงื่อนไขที่ต้องการลดขนาดแล้ว เริ่งการ Recursion

## Step ในการทํา recursion (ดูตัวอย่างไปด้วย)

1. ให้พยายามหากรณีที่ solution กับเราได้เคลียร์ของ problem

เช่น - การหา factorial กรณีเล็กสุดที่ 0! และ 1!

- หาตัวค่านอก array n ชิ้น กรณีเล็กสุดที่ n=1, มีช่องเดียวตัวที่ index นั้นแน่น: min

- Tower of Hanoi กรณีเล็กสุดที่ 1 disk ก็ใช้ source → target

2. เตรียม solution สำหรับกรณีที่ไม่เล็กที่สุดโดย แยก problem ออกเป็น 2 ส่วน ดัง ประวัติการทําทั้ง

กันก่อนมาหานั้น

→ นาทีที่ 1 recursion

เช่น - factorial n! หากเป็น n \* (n-1)!

→ recursion

- นาทีที่ min ใน array n ชิ้น หากเป็น array ช่องที่ 0 กับ [array ช่องที่ 1 ถึง n]

- Tower of Hanoi มองว่า เน้น disk 2 ชิ้น (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);
}
```

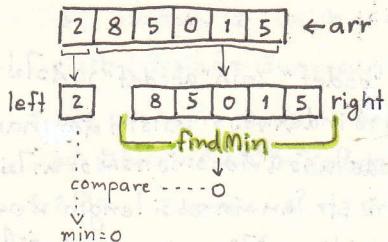
เรียก recursion  
ส่วนที่เล็กลง

Ex2. find min ของ 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); // นี่ min ช่วงที่เหลือ
        if(arr[start] < min) return arr[start];
        else return min;
}
```

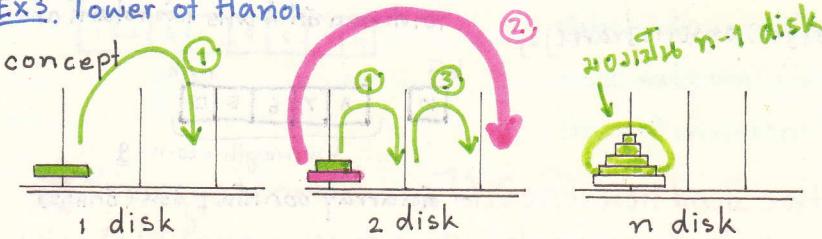
ที่เป็น min ... กรณีในบุ๊ก ส. 0 แต่ array 00 ก็เป็น 2 ต่อไป (left: right ต่อไป...) ส่วน right  
เราจะทำ การ recursion (assumption ว่า) ในหน้าต่อ min ให้ช่วง right (ตั้ง 0) และ return กัน  
มาให้เรา จากนั้น เมื่อเราถูกตัว left (ส่วนที่ตัวเดียว) กับตัว min ของช่วง right ก็แปลงตัว  
กันอย่างที่ 2 ตัวนี้ หันแผล: ถ้า min ของ array index ที่ start ถึง end

► เรา กด ก่อนว่า กรณีเล็กสุด ก็ต้อง  $n=0$  กับ  $n=1$  เช่น  
return ค่าใดๆ ก็ได้เลย (ans ต้อง 1)  
แต่ถ้าไม่ใช่กรณีเล็กสุด ก็ให้ return  $n * (n-1)$   
ซึ่งตัว ans ของ  $(n-1)!$  ก็เรียก recursion คิดต่อไป  
ให้เลย!



► ตอนแรกให้ดู ก่อนว่า ตัว start กับ end  
เท่ากัน (ถ้า นำ min ของ array ช่วงเดียว)  
ซึ่งจะนับ 0 ถึง 5 ซึ่งเดียว ตัวนี้แผล:

**[Tip]** การที่เราเขียน method recursion แบบนี้ได้ เราต้อง "ชี้รู้" ว่า  
เวลาเราทำ การ recursion แล้ว มันจะ เข้าไป ใกล้ กรณีเล็กสุด  
ที่เรา วางแผน เช่น เรื่อง fac กรณีเล็กสุด ต้อง 0, 1 และ เราเริ่ม เรียกที่ ก. เป็น: ก  
แล้ว ก จะถูกเรียก แบบ ก-1 ผ่าน recursion แปลว่า นั่นก็การ เรียก recursion  
หลักๆ รอบ แล้ว ก จะ: เหลือ 0 กัน 1 ตามที่เรา set ไว้, เรื่อง findMin เรา  
จะ start เป็นตัว ยังไม่ถูกเรียก ต้อง กรณีเล็กสุด ต้อง start = end ตอนเริ่มต้น start  
จะ: น้อยกว่า end แบบนี้ ตอน recursion เราก็ เช่น start + 1 เนื่องตัว นี้จะเรียกๆ จน  
ใหญ่สุดๆ: เพิ่มงาน เท่า end ก็ตรงกับ กรณีเล็กสุดที่ เตรียมไว้!

Ex3. Tower of Hanoi

\* code Hanoi หมายความว่า ต้องการจัดเรียง ให้เป็น 2 disk กับ n disk บนมือกัน แต่ตอนที่ จัดเรียง step ① เราจะ นับ ว่า มี n เมื่อ n-1 disk ซึ่งจะ เท่า n-1 นี่ ข้างไปแล้ว ก็ เรียก recursion ต่อ กันแผล:

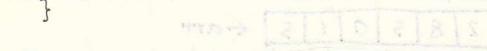
► ใช้ลักษณะว่า มี 2 disk กับ n disk บนมือกัน แต่ตอนที่ จัดเรียง step ① เราจะ นับ ว่า มี n เมื่อ n-1 disk ซึ่งจะ เท่า n-1 นี่ ข้างไปแล้ว ก็ เรียก recursion ต่อ กันแผล:

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;
    }
}

```



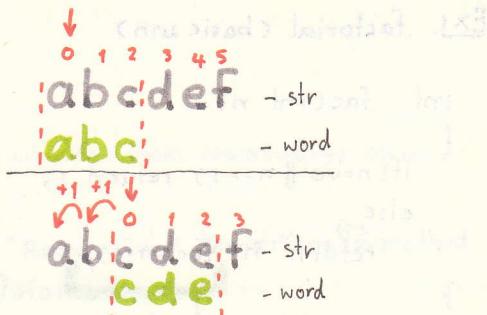
ก็สังเกตช่วง "bcdef" (จาก "abcdef") ที่วนใน indexOf ถ้า กันมันเจอกันก็จะ: return 0; กันมา แต่ถ้า 0 ตัวนั้นไม่ใช่ indexOf จริงๆ เนื่องจาก char ช่วงนี้อ่านไป 1 ตัว เราจะยังต้อง +1 ตัวให้มัน  
- การเขียน code ดูนี้ ห้องจะว่างการที่ร่วงไว้ เช่น length (ถ้า word ไม่อยู่ใน str) ต้องเมื่อเราหัด str ไว้เรื่อยๆ จนช่วงหนึ่งที่ str โอนติดจะมี length น้อยกว่า word ซึ่งแนะนำ... ไม่มีทางที่ word 0:0 ถึง str แล้ว เราต้อง condition นี้ไว้บนสุดเลย ถ้า สังเกต return -1;

Ex 5. Knapsack Problem. เรียนโจทย์ที่เรามีของชิ้นหนักต่างกันหลายชิ้น และเราต้องการรู้ว่าเมื่อเราต้องการหันน้ำรวมหันน้ำหนักห้อง ห้องน้ำบินชิ้นไหนมี 11, 8, 7, 6, 5, 2 หันน้ำรวม = 20 เราก็ห้องน้ำ 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] = temp[j];
        return ans;
    }
    return new int[0];
}

```



41 note: โจทย์นี้ไม่ควรใช้เป็น recursion

แก้ไข: (แต่ใช้เพิ่มก็ยังได้!), ต้องแยก... การหา indexOf ก็ง่ายกว่าด้วยการที่เรา return 0; ได้แล้ว... แล้วก็จะเรียก recursion

note: knapsack มีการเขียน code ไว้หลายวิธี... รันนี้เรียนแบบห้องชั้นใช้ array เป็นตัว return (code นี้เรียกค่า 0 )... อาจมี Linked List มาซึ่งแทน array ก็ได้ด้วย:

41 method knapsack เราทำกิจกรรม array ที่เก็บของ (หันน้ำ), pointer ที่จุดเริ่มของ array, weigh ที่ห้องน้ำ ผลลัพธ์: return เรียง array ที่เก็บห้องน้ำ (ans)

- เริ่มด้วยการ check กรณีที่ weigh ไม่ได้ก่อตัว ต้อง weigh มันติดลบ หรือ start ว่าง array ไม่แล้ว ก็ return array size = 0 (no ans!)

weigh = 20

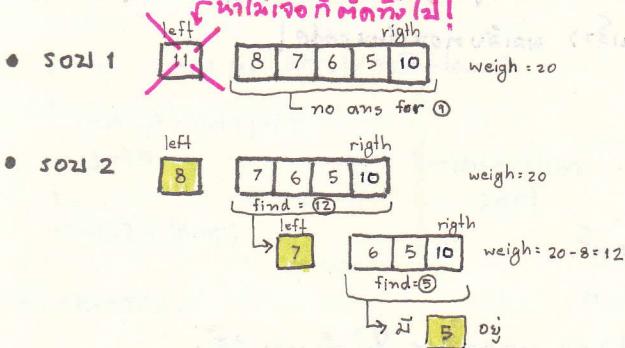
- หากน้ำเรียกว่า loop น้ำว่างใน array เราเพิ่งผ่านหัว "weigh" เท่ากับ weigh ที่ต้องการน้ำซึ่งก็มี 5 หันน้ำ array ans ขนาด 1 ช่อง แล้ว 10 หันน้ำยังคงอยู่ใน ans และ return ans เลย

left right.  
weigh = 20-11 = 9

- ตัว array ตอนนี้มี 2 หันน้ำ (สักสอง)

- กรณีส่วน right ให้ assume ว่าเราฯ นำ weigh = weigh - arr[i] ( เช่น weigh = 20, ก้าวงานยัง 11 รอบไป เมื่อตัวแรก ก็แปลงว่า กรณี right ตัวเราฯ นำนั้นของที่มี weigh = 9 ได้ เมื่อเราฯ วนกัน 11 ก้าว 7 ถึง 20 รอบต่อไป ก้าวฯ ฯลฯ แบบ 11 ช่วง right เราเก็บผลการเรียงกัน recursion โดย ข้อมูล index ก็จะ หาฯ เลื่อนไป กันช่วงๆ 1 (มันก็จะกลับๆ ตัว array เป็น 2 ส่วน ) และ weigh ก็จะบวกลงด้วย ให้หนังสือ left

- แต่ถ้าเราฯ นำ weigh กรณี right แล้ว return **array size = 0** ก็จะมา... ก็แปลงว่า กรณี right ไม่มี weigh ก้าวฯ ต้องการเรียง ( ก็แปลงว่า หาฯ ไม่ได้ )  $\rightarrow$  เราเก็บตัว left ที่ไปเหลือ ( ก็มันนาฯ ควรไว้ได้ ) แล้วรับ...



- เมื่อเราฯ ตัดตัวแรก ที่ไป ( ตัดก็วนกัน )  
 for loop ... วนต่อไป เราเก็บหัว left ใช้ใน 8, ॥ลากหัว 12 (weigh - 8 = 12) ฯลฯ ช่วง right ( ซึ่งมันก็จะเรียก recursion อีก )
- เมื่อ 170 5 ก้าวฯ return 5 ก้าวฯ ไป รวมกัน 7  
 และ return กลับมายืนหัว ชุดแรก ก้าวฯ ไป รวมกัน 8 รอบ  $\therefore \text{ans} = 8, 7, 5$

### Ex6. Merge Sort, Quick sort

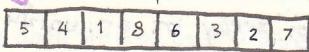
ทั้ง Merge, และ Quick คือหลักของ การ divide and conquer โดยมี concept ว่า แบ่ง array ออกเป็น 2 ส่วน 1 อย่าง 2 ส่วนนั้นนำไปเรียงนั้นเรียกว่า sort ( sort คือการเรียง recursion ) แล้วก่อต่อ 1 อย่าง ส่วนของอยู่ที่ sort แล้ว ก้าวฯ วนกันต่อ กันต่อ กันต่อ...

Merge Sort : 101 array แบ่ง ครึ่งๆ เหลือ ( ที่ 2 array ที่มี size เท่ากัน ) แต่ละขั้นตอน array ยังมี 10 อยู่

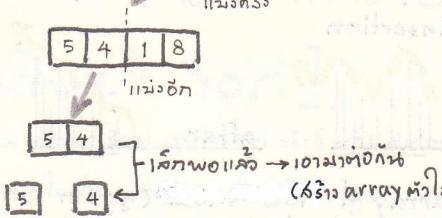
Quick Sort : 101 array แบ่ง 2 ตัว มี size ไม่เท่ากัน โดยมี pivot ใช้หัวหินหก เดียวคืออยู่ดูๆ advance sort!

### • Merge Sort

ตอน merge sort ผู้สอนยกให้ 1 ขั้นตอน 101 array ( ประมาณ 10 รอบ )



เราฯ ใช้ Merge Sort ด้วยการ ตัด array ออกเป็น 2 ส่วน ประมาณ 10 รอบต่อไป ก้าวฯ ก็จะ ส่วน ก็จะ เอาไป recursion ตัด 10 รอบ เมื่อเข้าช่วงๆ จะไปต่อ ก้าวฯ วน เหลือ ขนาด sort ได้ ( เช่น เนื้อ 2 ชุด เว็บๆ ) ก็จัดการ sort ซึ่งเริ่มๆ พอกันนั้น ให้เสร็จ แล้ว ก่อต่อ 1 อย่าง ส่วนของ ก้าวฯ แล้ว มาต่อ กัน ต่อ กัน



### • 1 อย่าง ต่อ กัน โดย 1 รอบ $\Rightarrow$ while loop ( วนซึ่งกัน กัน )

( 1 รอบ วน ก้าวฯ data ทาง บน ก้าวฯ ทาง ล่าง แล้ว วน กัน ให้ )  
 การ check ว่า ที่ index แรก ของ array สอง 2 ตัว ตั้งใหม่ที่นี่อยู่กว่า ก้าวฯ ไหน ที่ array ใหญ่... ทำ วน วน หรือ data ทาง array ยัง ไม่ได้ array ใหญ่ ที่จะ หัว ( นั่นไงได้ แล้ว ก้าวฯ ) จนนั้นเป็นหลัก ก้าวฯ ! )

\* merge sort ก้าวฯ ได้เร็วกว่า พอก basic sort  $O(n \log n)$  [ คิดโดย มันต้อง แบ่งครึ่งๆ ไปเรื่อยๆ ( $\log_2 n$ ) และ แบ่ง + เอาจาต่องกัน (merge) ที่นั้นดู ก้าวฯ ]

แต่ ข้อเสีย ของ merge sort ต้อง ใช้ space memory มากมาย ( ต้องสร้าง array ใหม่ )

# Advance Sorting

## Advance Sort.

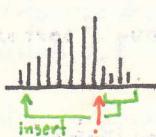
เป็น algorithm ที่ดีกว่า insertion sort ที่มีความซับซ้อนต่ำกว่า insertion sort แต่ใช้เวลาในการ sort นานกว่า insertion sort แต่การ sort ของ insertion sort สามารถ sort ได้ในคราวเดียว แต่ advance sort ต้อง sort ในคราวๆ กันๆ จึงทำให้ใช้เวลาในการ sort นานกว่า insertion sort

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

## [Shell Sort]

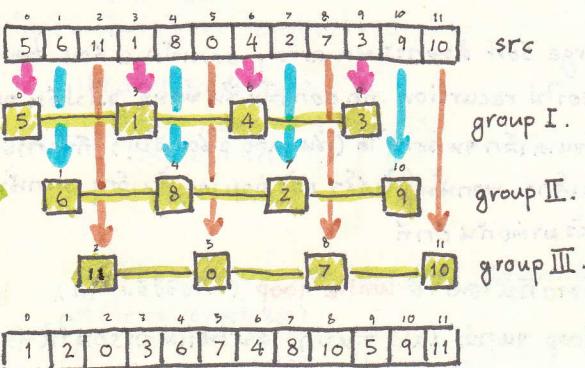
Shell Sort เป็นการ sort "Insertion Sort" ที่ upgrade ให้ทำงานดีขึ้น

- เวลาเรา sort insertion sort ... เราจะทำการนับ data 00 ถึง 1 ตัว แล้ว วนไปอีก 1 ตัว ฯลฯ น้ำหนักที่จะ sort คือ index ไม่เรียบๆ งานๆ ชั้นการกำองบ่งนี้ ถ้าเรา sort data ที่มีตัว น้อยๆ อยู่ ปลายเก็บมากๆ ที่ทำให้ในloop insert ใช้เวลา的工作 มาก เพราะ สันติํงวังวังกลับมาก!



< หัวน้อยๆ อยู่ ปลายมากๆ ทำให้ insertion ทำงานนาน เพราะ ต้องนำ data พอกันนี้เข้าไป ที่ไปกลับมาก (สับสนวุ่นวายเลย!)

**concept :** แบ่ง data เป็นช่วงๆ แล้วทำการ sort คล้าย insertion... จัดให้ 有序 ตามแน่น ของ data ใน array ด้วยที่จะ sort



src

group I.

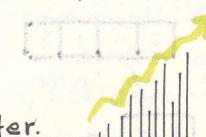
group II.

group III.

ตอนแรกในนั้นจะ array src ถูกแยกเป็น n group แห่งละ 5 ตัว ห่างๆ กัน และเรา sort group ตัวๆ ตามลำดับ insertion

before.

after.



ผลที่ได้คือ array ก็ "เร็ว" อะ เรียบ (ดูรูปนี่: จนทุกๆ กลับเข็น) สังเกตว่า นันเร็ม อะ: แบ่ง กันๆ หัวน้อยๆ: ถูกซ้ายมาต้านหน้า หัวลงกันๆ ใจไปกากๆ กันๆ

ที่ดีหนักวัน ... วิธีนี้จะเร็ว การจัดหัว array มาทำ insertion sort เพราะ ชั้น array เล็กจะทำให้ insertion ที่เร็วขึ้น จากนั้น เราจะ sort ที่ได้มาทำ insertion sort โดยการแบ่งกันๆ ออก แล้ว ชั้นนี้ให้เล็กลง (นั่น 3 รอบประมาณ) เช่น หัวห้องๆ งานใช้ชั้น h=3 ประมาณแรก การ sort รอบ 2 คือ หัวตัว หัวตัว หัวตัว หัวตัว หัวตัว หัวตัว insertion เลย แต่การทำ insertion แบบนี้จะทำให้เร็ว เพราะ data เก็บกลุ่มกันเรื่อง: แล้ว ( วันนี้ที่ insert ไม่ไกล ก็จะได้แล้ว! )

```

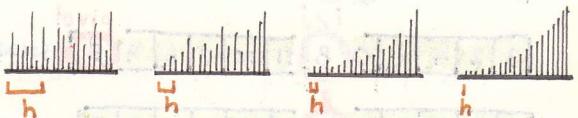
int i, j;
int temp, h=1;
while (h <= arr.length/3) h=h*3+1; calculate h
while (h>0)
{
    for (i=h; i< arr.length; i++)
    {
        temp = arr[i];
        for (j=i; j>h-1 && arr[j-h] >= temp);
        arr[j] = arr[j-h];
        j-=h
        arr[j] = temp;
    }
    h=(h-1)/3;
}

```

การเขียน code ของ shell sort ส่วนใหญ่ จะ  
ให้แล้วเดี๋ยว insertion มาก่อน แต่จะมีเพิ่ม h  
เข้ามาโดย h จะเป็นตัวบวก ช่วงๆ ของ group

- ใน insertion เรา ขยับที่ ก: index (ก้าวเท่ากับ h)  
กดีรึ h=1)

- ณ Shell เราจะขยับที่ ก: ห้องค่า h ที่จะเปลี่ยน  
เรื่องๆ (โดยประมาณ ก: ลดลง:  $\frac{1}{3}$ )



ในการวน for sort 1 รอบ h จะเล็กกว่าเรื่องๆ กล่าวคือ กัน  
กว่า เริ่มต้นของการ sort บนบานๆ และทำหน้าที่เดียวกันเรื่องๆ

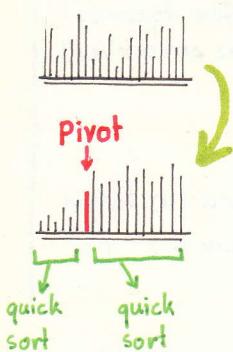
**[Note]** การทำ insertion sort 9 ॥ต่อ: group มันไม่ได้ทำ group I ให้  
เสร็จก่อน แล้วถึงทำ group II แล้วถึง group III, แต่มันจะ: ก้าว  
ที่ ก: n group พร้อมกันเลย ดัง ติด ถูก ก: 1 ของ group I  $\rightarrow$  ถูก ก: 1 ของ group II  $\rightarrow$   
ถูก ก: 1 ของ group III แล้ววนกลับมาติด ถูก ก: 2 ของ group I ตาม (ดังว่า มันจะ: ติดแต่ละชุด  
ไปพร้อมๆ กัน)

### Efficiency...

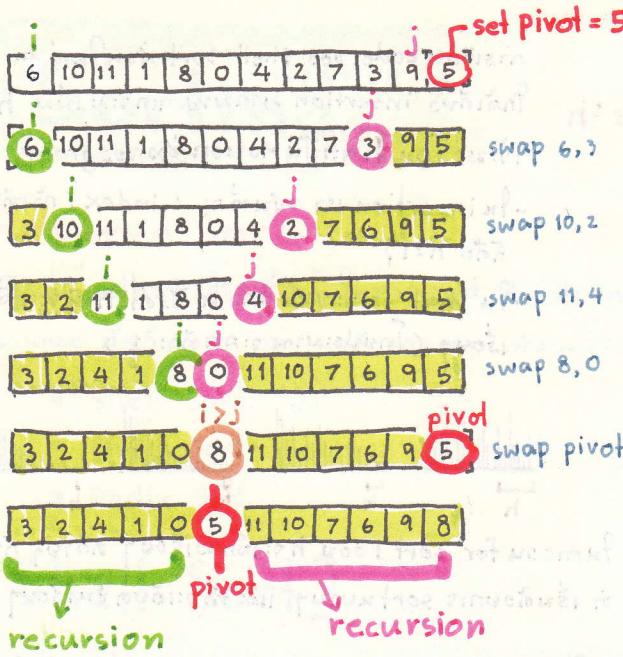
Shell Sort เป็น sort ที่มีเวลาในการทำงานໄส์คงที่ ขึ้นอยู่กับการ set ต่ำ h และ การเรียง data ที่ array  
มากแรก ... แล้วเมื่อมันมี Logic เดียวกับ insertion กรณี best case 9: เท่ากับ  $\mathcal{O}(n)$  ซึ่ง  
จะเมื่อ data มีที่เรียง 0% แล้ว ส่วนต่ำ average 9: ใช้เวลาประมาณ  $n \log n$  ส่วนใหญ่แล้ว 9: สำหรับ  
กรณี worst case 9: เท่ากับ  $\mathcal{O}(n^2)$

### Quick Sort

เป็น algorithm 9: sort ที่เรียกว่าหุ่นตอกหิน ... 9: concept "recursion" โดยมี array 00 กว่า 2  
ส่วน (ไม่ต้องเท่ากันเหมือน merge sort) โดยจะมี pivot เป็นหัวใจ



- ก่อนอื่น เราต้องกำหนด pivot มาก่อน (ในที่นี้เราให้ เมื่อ ตัวขวาสุด (right index)  
ของช่วงที่เรา 9: sort ... แล้ว data ตัวไหน ก: มีค่า น้อยกว่า pivot 9: ถูก ซ้ายไปอยู่  
ก: left index, หัวที่มากกว่า pivot 9: ถูก ซ้ายไปอยู่ ก: right index)
- แล้ว เราก็ recursion left กับ right อีก



\* code vos quick sort เป็นที่สุดในความเร็วแต่  
มี concept ให้ค้นหานั้นมาก!

- ให้ตัว right เป็น pivot
  - set pointer 2 ตัว ที่นั่งกันท้าย (i กับ j)
  - วนการทำ 1 รอบ เริ่มต้น: while loop วนนานจนกว่า  $j > i$  ตัวที่มากกว่า pivot ให้เข้า left ( $i++$ )
  - จนกว่า  $j < i$ : จงวนตัวที่น้อยกว่า pivot ให้เข้า right ( $j--$ )
  - จากนั้นทำการ swap 2 ตัวนั้นแล้ว วนนานๆ วนกันไป...
  - ให้ปีปัจจุบัน ตัว  $i$  มากกว่า  $j$  ให้เข้าชุด และทำการ swap ตัวที่ index  $i$  กับตัว pivot
  - แล้วเราก็มาทำ quick sort บน  $i+1$  ถึง  $i$  กับ  $i+1$  ถึง  $n$  อีก!
- \* code ของชากอร์ด กัน!

## Efficiency

best case  $\rightarrow \Omega(n \log n)$

average  $\rightarrow \text{ประมาณ } n \log n$  (กว่าๆ)

worst case  $\rightarrow O(n^2)$

## [Radix Sort]

เมื่อกำลัง sort ที่เรื่องมากว่าซึ่ง (อย่างเช่น quick sort ด้วยซึ่ง) ... concept คือ เรียง data ใน array ที่คละๆ กัน เริ่มจาก หลักหน่วย  $\rightarrow$  สิบ  $\rightarrow$  ร้อย  $\rightarrow$  พัน ...

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

sort ด้วย  
หลักหน่วย

sort ด้วย  
หลักสิบ

sort ด้วย  
หลักร้อย

} ตั้งแต่แรก (ตั้งแต่ int แรกจนถึงตัวที่เป็น int)  
(int)(cr/lu)

เมื่อกำลัง sort ที่ concept ว่ายังไงน้ำด้วย แก้มเร็วตัวเอง!

► เนტิผิดก็ต้องเริ่มจากหลักหน่วย เช่น: เราต้องเก็บตัวที่มีผลลัพธ์ของการ sort

มากที่สุด (หลักที่มากที่สุด) ไว้ก่อนแล้ว หลักหน่วยไว้ค่อยลำดับตัวกันไปเรื่อยๆ

จะ sort กัน

## method getDigit()

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

## Coding!

มีอยู่ 2 วิธี (ทำทั้ง 2 ก็ได้)

แบบ 1 ใช้ 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() > 1)
            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;
```

## Efficiency

ใช้วิธีการทั่วไปคือ **Ockn)** เมื่อ k เป็นจำนวนหลักที่มากที่สุด... จะเห็นว่า มันทำงานเร็วกว่า quick sort ถ้า k มาก ส่วน quick sort ต้องพูด "number" เท่านั้น เพราะมันไม่มีการ compare และ sort ของ array ของ Obj. ไม่ได้ !!

เรื่อง Radix เราจำเป็นต้อง ดูด เลข เช่น หลัก 0 0 ก็  
ติด หลักเดียว แล้ว ทำก็ควรเขียน method ให้ไว  
เช่น  $\text{getDigit}(1453, 3)$  จะได้ = 4

## แบบ 2 queue เก็บตัวเลข เก็บ

```
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() > 1)
            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;
```

Shell	Quick	Radix
-concept เหมือน insertion แต่ ช่วงการ sort 7 กลุ่ม	- ใช้ array เป็น 2 ส่วนๆๆ กับ pivot - มีการใช้ recursion	- sort ทั้งหมด - ไม่มีการ compare เลย - sort Object ไม่ได้ !!
Worst (最坏情况) $O(n^{3/2})$ Avg. (平均情况) $O(n \log n)$	Worst $O(n^2)$ Avg. $O(n \log n)$	Worst (最坏情况) $O(kn)$ Avg.