บทที่ 1
บทนำ

หนังสือนี้จะประกอบด้วยเทคนิคที่ใช้ในการหาคำตอบสำหรับปัญหาต่างๆ ที่ใช้คอมพิวเตอร์ เทคนิคในที่นี้ ไม่ได้หมายถึงภาษาการโปรแกรม แต่จะหมายถึงวิธีการขั้นตอนที่ใช้ในการแก้ปัญหา ตัวอย่างการค้นหาหมายเลขโทรศัพท์ ชื่อนันทิกา เบญจเทพานันท์ จากสมุดโทรศัพท์ปกขาว สามารถเริ่มจากการค้นตั้งแต่ชื่อ กกกนก ประภาพันธ์, ชื่อที่สอง กกกร ประสพเนตร ชื่อที่สาม กกกร สิทธิทรัพย์ ไปเรื่อยๆ จนกว่าจะพบชื่อ นันทิกา เบญจเทพานันท์ เชื่อเถอะไม่มีใครงุ่มง่ามใช้วิธีแบบนี้ในการหาหมายเลขโทรศัพท์ จากการที่เรารู้ว่าสมุดโทรศัพท์จะพิมพ์รายชื่อและหมายเลขโทรศัพท์ตามลำดับอักษร เราก็จะเปิดสมุดคร่าวๆไปที่หมวดอักษร "น" จากนั้นก็ไล่ขึ้นถ้าลำดับอักษรนันทิกาอยู่ก่อน หรือไล่ลงถ้าลำดับอักษรนันทิกาอยู่หลัง ทำเช่นนี้ไปเรื่อยจนกว่าจะพบชื่อ นันทิกา เบญจเทพานันท์ พอจะนึกออกมั้ยว่าแบบแรกเขาเรียก การค้นหาแบบตามลำดับ (sequencial search) แบบหลังจะเป็นวิธีการค้นหาตามดัชนี(index sequential search)

นั่นคือเริ่มจากการกำหนดปัญหา แล้วหาวิธีขั้นตอนที่จะได้คำตอบ(algorithm) ในหนังสือนี้เราจะไม่เน้นการทำโปรแกรม แต่เราจะพิจารณาเทคนิคการแก้ปัญหาในแบบต่างๆ แล้ววัดว่าแต่ละเทคนิคที่ใช้มีประสิทธิภาพแตกต่างกันอย่างไร เราจะพบบ่อยๆว่า เมื่อกำหนดปัญหาขึ้นมาปัญหาหนึ่ง จะมีวิธีการหาคำตอบได้หลายๆแบบ แต่จะมีวิธีหนึ่งที่มีขั้นตอนวิธีที่เร็วกว่าวิธีอื่นๆ ตัวอย่างการค้นหาชื่อในสมุดโทรศัพท์ จะพบว่าวิธีการค้นหาแบบไบนารี จะทำได้เร็วกว่าวิธีการค้นหาตามลำดับ นั่นคือเราไม่เพียงแต่ต้องการคำตอบเท่านั้น เรายังต้องการวิธีการที่มีประสิทธิภาพในแง่ของเวลาที่ใช้ในการหาคำตอบ(time)และทรัพยากรที่ใช้ (storage) เมื่อขั้นตอนวิธีถูกนำมาทำให้เป็นผลบนคอมพิวเตอร์ เวลาที่ใช้จะหมายถึงจำนวนรอบในการคำนวณ(CPU cycles) และทรัพยากรจะหมายถึงหน่วยความจำที่ต้องใช้ (memory) คุณอาจจะสงสัยว่าทำไมเราถึงต้องตระหนักในเรื่องของประสิทธิภาพ เพราะ คอมพิวเตอร์นับวันมีแต่จะทำงานเร็วขึ้นๆ แถมหน่วยความจำก็โตขึ้นๆ แล้วก็ถูกมากๆ หนังสือเล่มนี้ก็จะไขคำตอบนี้แก่คุณผู้อ่านในลำดับต่อๆไป

1.1 ขั้นตอนวิธี(algorithms)

ก่อนหน้านี้เราพูดถึงคำว่า "ปัญหาโจทย์", "คำตอบ", "ขั้นตอนวิธี" คราวนี้เราจะมาพิจารณาโดยละเอียด

<u>โจทย์</u> จงเรียงลำดับตัวเลขในเซ็ต X จำนวน 10 จำนวน จากน้อยไปมาก

X = {10, 7, 3, 5, 4, 8, 6, 9, 1, 2} และ n = 10

จากโจทย์ ประกอบด้วย พารามิเตอร์ X (เซ็ท) กับ พารามิเตอร์ n (จำนวนตัวเลขในเซ็ต)

<u>คำตอบ</u> คือ X = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

เราสามารถหาคำตอบจากสัญชาตญาณ จากการมองผ่านๆว่าตัวเลขใดน้อยกว่าก็ยกมาวางในตำแหน่งที่ถูกต้องเพราะจำนวนตัวเลขยังน้อยๆอยู่ แต่ถ้าตัวเลขในเซ็ต มีจำนวน 1,000 จำนวน จะมาใช้สัญชาตญาณแบบเดิม คงจะเรียงลำดับไม่ถูกแน่นอน เราจะต้องหาขั้นตอนวิธีมาจัดการให้ได้คำตอบ

<u>ขั้นตอนวิธี</u> เปรียบได้กับการจับนักเรียนอนุบาลเข้าแถวเรียงลำดับจากเตี้ยไปสูง โดยเอานักเรียนคนที่ 1 เทียบกับเด็กคนอื่นๆในแถว ถ้ามีเด็กคนใดตัวเตี้ยกว่า(x[i] > x[j]) ก็จะสลับเด็กคนที่กำลังเปรียบอยู่ เข้าไปยืนแทน แล้วเอาเด็กคนที่เตี้ยกว่ามาเทียบกับคนอื่นๆต่อไปจนท้ายแถว ก็จะได้คนที่เตี้ยที่สุดออกมา แล้วทำเช่นเดียวกับเด็กคนที่ 2,3,4 ไปเรื่อยๆ จนเหลือคู่สุดท้าย วิธีการนี้เรียกว่า selective sort

```
public static void selectiveSort(int x[], int n)
{int i,j,k;
for(i=1;i<n;++i)
        {for(j=i+1;j<=n;++j)
                {if(x[i]>x[j])
                        {k=x[i];
                          x[i]=x[j];
                          x[j]=k;
                        }
                }
        }
}//selectiveSort
```

<u>โจทย์</u> จงหาว่า X เป็นตัวเลขในเซ็ต X หรือไม่

X = {10, 7, 3, 5, 4, 8, 2, 9, 1, 6} n = 10 และ key= 5

จากโจทย์ จะมี 3 พารามิเตอร์ คือ พารามิเตอร์ X (เซ็ท) กับ พารามิเตอร์ n (จำนวนตัวเลขในเซ็ต) ตัวเลข key

<u>คำตอบ</u> คือ “yes key is in X”

<u>ขั้นตอนวิธี</u> เริ่มจากเปรียบเทียบค่า key กับ x[1] ถ้าเท่ากัน ก็ให้ตอบว่า yes มิฉนั้น เทียบกับ x[2] ,x[3], x[4], x[5] ตามลำดับ จนกว่าจะพบตัวที่เท่ากับ key หรือ จนกว่าจะหมด ถ้ามีตัวที่เท่ากัน ก็ให้ตอบ “yes” มิฉนั้นตอบ “no”

```
public static int sequential(String x[], String key)
{     int i;
       for(i=1;i<=10;++i)
        {  if(key.equals(x[i])  )
             {return i;}
         }
        return -1;
}//sequential
```

1.2 ประสิทธิภาพของขั้นตอนวิธี

คราวนี้จะเปรียบเทียบประสิทธิภาพของ 2 ขั้นตอนวิธี กับปัญหาเดียวกัน

1.2.1 การค้นหาแบบตามลำดับ กับ แบบไบนารี((Sequential search vs Binary search)

ตอนนี้เราจะเปรียบเทียบขั้นตอนวิธีในการค้นหาชื่อ ในสมุดโทรศัพท์ 2 วิธี

วิธีการค้นหาแบบตามลำดับ(sequential search)

```
 public static int sequential(String x[], String key)
{      int i;
        for(i=1;i<=10;++i)
         {  if(key.equals(x[i])  )
              {return i;}
          }
         return -1;
}//sequential
```

Best case : กรณีที่ key เท่ากับค่า x ตัวแรก (key =x[1]) ทำให้จำนวนรอบที่เปรียบเทียบเพียง 1 รอบ

Worst case : กรณีที่ key เท่ากับค่า x ตัวสุดท้าย (key =x[10]) หรือกรณีไม่พบ จะได้จำนวนรอบที่เปรียบเทียบเป็น 10 รอบ

Average case : โดยเฉลี่ยจำนวนรอบจะเท่ากับ (1+2+3+...+ 10)/10 = 5.5 รอบ

วิธีการค้นหาแบบไบนารี (binary search)

```
public static int binary(String x[], String key)
{
        int mid,top, bottom,i;
        bottom = 1; top = 10;
        mid = (bottom+top)/2;
        do
        {i=key.compareTo(x[mid]);
        System.out.println("i is"+i);
                if(i==0){return mid;}
                else if(i<0){top = mid-1;}
                else {bottom = mid+1;}
                mid = (bottom+top)/2;
        }while(bottom < top);
        return -1;
}// binary
```

Best case : ค่า key ตรงกับค่า x ตำแหน่งตรงกลาง(5) key = x[5] จะทำให้จำนวนการเปรียบเทียบเพียง 1 ครั้ง

Worst case : ค่า key ตรงกับค่า x ตำแหน่งที่ 1 หรือตำแหน่งที่ 10 หรือกรณีไม่พบ จะต้องเปรียบเทียบจำนวน 4 รอบ

Average case: จำนวนครั้งที่เปรียบเทียบจะน้อยกว่ากรณี worst case เล็กน้อย

1.2.2 การเรียงลำดับแบบเลือก กับการเรียงลำดับแบบผสาน (Selective Sort vs. Merge Sort)

<u>การเรียงลำดับแบบเลือก</u> (Selective Sort)

```
public static void selectiveSort(int x[], int n)
{int i,j,k;
for(i=1;i<n;++i)
    {for(j=i+1;j<=n;++j)
        {if(x[i]>x[j])
            { k=x[i];
              x[i]=x[j];
              x[j]=k;
            } // if
        } // end j
    } // end i
}//selective
```

x ={8, 3, 13, 6, 2}

i	j	x[1]	x[2]	x[3]	x[4]	x[5]
1	2	3	8	13	6	2
1	3	3	8	13	6	2
1	4	3	8	13	6	2
1	5	**2**	8	13	6	3
2	3	**2**	8	13	6	3
2	4	**2**	6	13	8	3
2	5	**2**	**3**	13	8	6
3	4	**2**	**3**	8	13	6
3	5	**2**	**3**	**6**	13	8
4	5	**2**	**3**	**6**	**8**	13
x={2, 3, 6, 8, 13}						

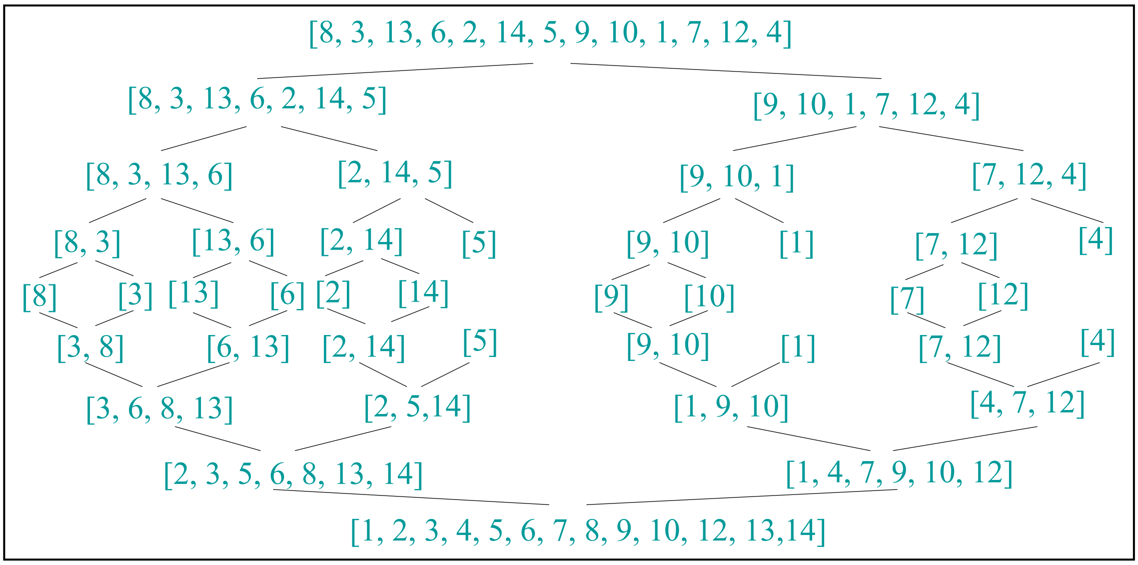
จำนวนครั้งในการเปรียบเทียบเท่ากันในทุกกรณีไม่ว่าจะเป็น best, average หรือ worst case

<u>Best case</u> : กรณีที่ข้อมูลมีการเรียงลำดับอยู่แล้ว จะใช้เวลาน้อยที่สุด เพราะทำให้ไม่ต้องมีการสลับข้อมูล

<u>Worst case</u> : กรณีที่ข้อมูลมีการเรียงลำดับในทางตรงข้าม เช่นต้องการเรียงลำดับจากน้อยไปมาก แต่ข้อมูลมีลำดับจากมากไปน้อย ทำให้ต้องมีการสลับข้อมูลทุกครั้ง

<u>การเรียงลำดับแบบผสาน</u> (Merge Sort)

การเรียงลำดับแบบเมริดจ์ เริ่มต้นด้วยการแบ่งครึ่งอาเรย์ออกเป็น 2 อาเรย์ย่อย และแบ่งครึ่งอาเรย์ย่อยออกไปเป็น 4 อาเรย์ย่อย และแบ่งครึ่งอาเรย์ย่อยออกไปเรื่อย ๆ จนกระทั่งอาเรย์ย่อยมีขนาดเป็นหนึ่ง ซึ่งอาเรย์ขนาดหนึ่งเป็นอาเรย์ที่เรียงลำดับแล้ว หลังจากนั้นจึงทำการผสานอาเรย์ย่อยเข้าด้วยกันเป็นอาเรย์ที่ใหญ่ขึ้น และทำเช่นนี้ต่อไปจนกระทั่งได้อาเรย์ที่เรียงลำดับแล้ว



รูป 1.1 การเรียงลำดับแบบผสาน แบ่งแยกแล้วผสาน

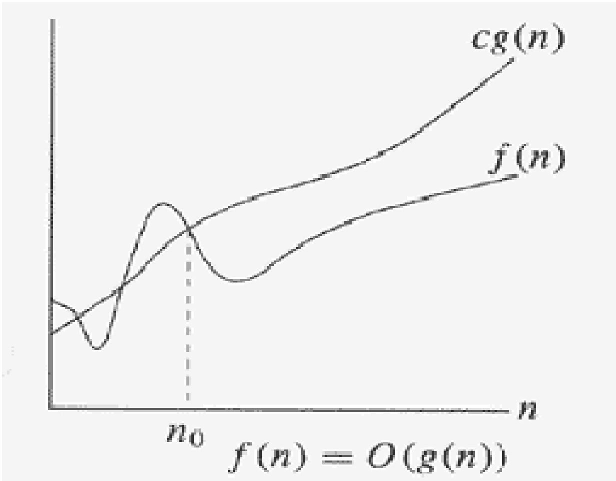
```
public static void mergeSort(int a[], int p, int r)
{     int q;
      if (p<r)
           {  q=(p+r)/2;
              mergeSort(a,p,q);
              mergeSort(a,q+1,r);
              merge(a,p,q,r);
           }
} // end mergeSort
public static void merge(int a[], int p, int q, int r)
{     int n1,n2,i,j,k;
      n1 = q-p+1;
      n2 = r-q;
      int left[] = new int [20];
      int right[] = new int [20];
      for (i = 1;i<=n1;++i)
           { left[i] = a[p+i-1]; } // เก็บข้อมูลครึ่งซ้าย
      for (j = 1;j<=n2;++j)
           {  right[j] = a[q+j]; } // เก็บข้อมูลครึ่งขวา
     left[n1+1] = 100000;      // set ค่า หลังตัวสุดท้ายเป็น ∞
     right[n2+1] = 100000;    // set ค่า หลังตัวสุดท้ายเป็น ∞
     i=1;j=1;
     for(k=p;k<=r;++k)
           {  if (left[i] <= right[j])
                  {  a[k] = left[i];
                     i=i+1;   }
              else
                  {  a[k] = right[j];
                     j=j+1;   }
            }
}// merge
```

ถ้าพิจารณาจำนวนครั้งของการแบ่งครึ่งอาเรย์ออกไปเรื่อย ๆ จนไม่สามารถแบ่งได้อีกแล้ว จะทำได้ไม่เกิน log n ครั้ง และจำนวนการเปรียบเทียบข้อมูลในการผสาน 2 อาเรย์ย่อยเข้าด้วยกันในแต่ละครั้งจะไม่เกินจำนวนสมาชิก ซึ่งเท่ากับ n ครั้ง จำนวนรอบจะเท่ากับ n log n

1.3 สัญลักษณ์เข้าสู่ค่าอนันต์ (Asymptotics Notation)

Big – O

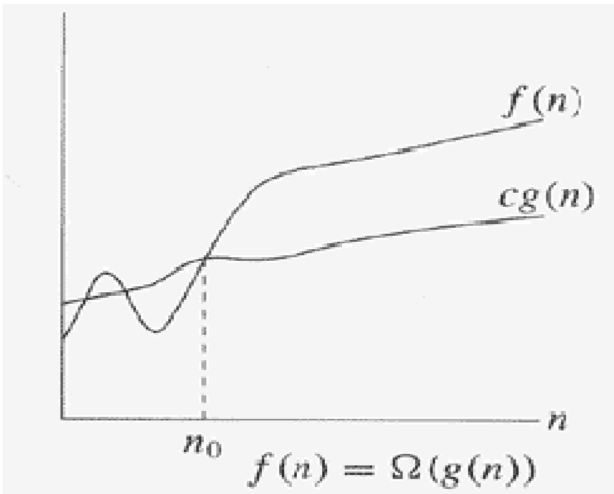
<u>นิยาม</u> $f(n)=O(g(n))$ หากมีจำนวนเต็มบวก c และ n_0 ที่ซึ่ง $f(n) < cg(n)$ สำหรับทุกค่า n, $n \geq n_0$



<u>รูป 1.2</u> $f(n)=O(g(n))$ ที่ $f(n) < cg(n)$ สำหรับทุกค่า n, $n \geq n_0$

Ω *- notation*

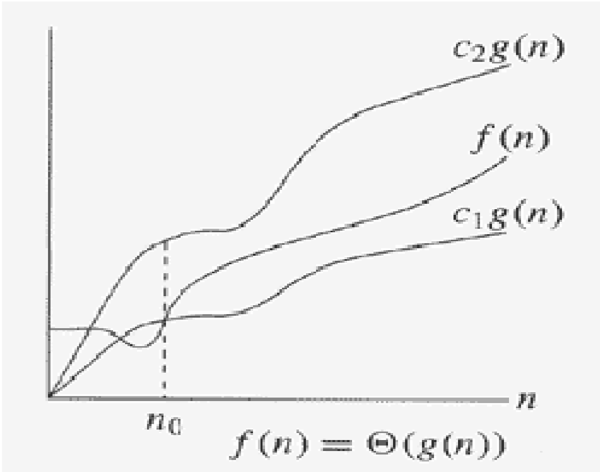
<u>นิยาม</u> $f(n) = (\Omega\, g(n))$ หากมีจำนวนเต็มบวก c และ n_0 ที่ซึ่ง $f(n) > cg(n)$ สำหรับทุกค่า n, $n > n_0$.



<u>รูป 1.3</u> $f(n)=\Omega(g(n))$ ที่ $f(n) > cg(n)$ สำหรับทุกค่า n, $n \geq n_0$

Θ – notation $(\Theta = O \cap \Omega)$

<u>นิยาม</u> $f(n) = (\Theta\ g(n))$ หากมีจำนวนเต็มบวก c และ n_0 ที่ซึ่ง $c_1g(n) < f(n) < c_2g(n)$ สำหรับทุกค่า $n, n > n_0$.



<u>รูป 1.4</u> $f(n)= \Theta(g(n))$ ที่ $c_1g(n) < f(n) < c_2g(n)$ สำหรับทุกค่า $n,\ n \geq n_0$

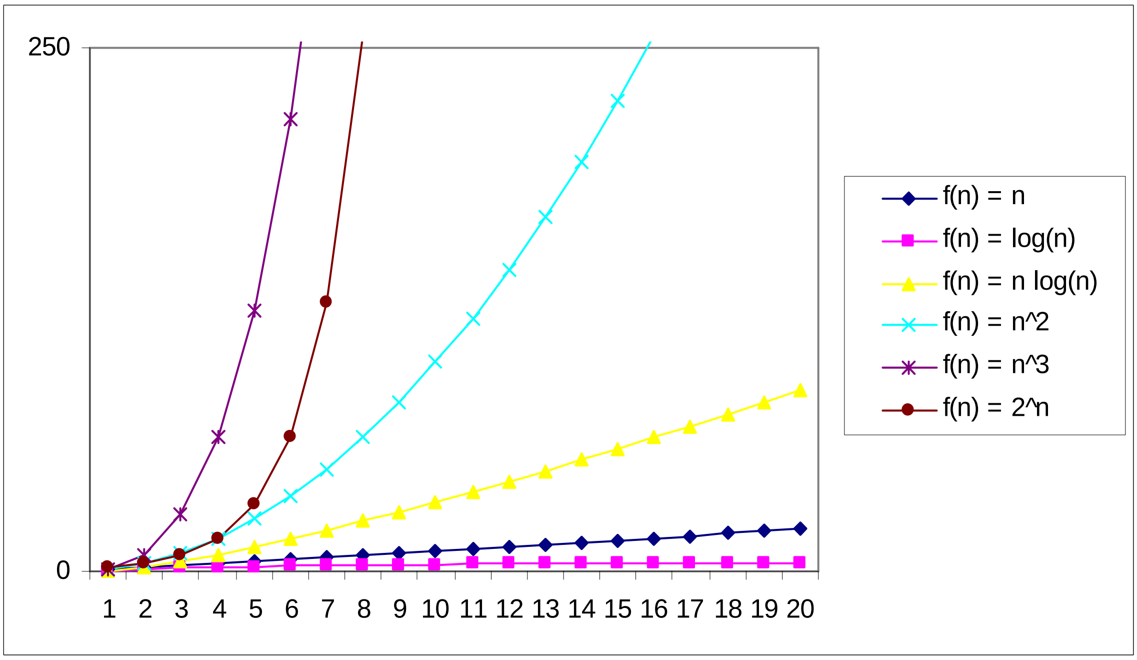
1.4 อัตราการเติบโต (*Growth Function)*

เมื่อมีการใช้คอมพิวเตอร์ในการประมวลผล ซึ่งจะต้องมีการกำหนดขั้นตอนวิธี และเขียนโปรแกรม ดังนั้นเราคงจะไม่ใช้กับขนาดของ input 10-20 จำนวน แต่เราจะต้องคำนึงถึงขนาดของข้อมูลที่โตมากๆ เมื่อขนาดข้อมูลขึ้นมากๆ เวลาที่ใช้ประมวลผลก็มากขึ้นตามจำนวนข้อมูล(n)

เราจะลองเปรียบอัตราการเติบโตของ function ที่มักพบบ่อย ได้แก่

- O(1): constant
- O(n): linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- O(log n)
- O(n log n)

		Instance characteristic n					
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}



ฟังก์ชัน Big-O ที่ได้มักอยู่ในรูป 1, log n, n, n log n, n^2, n^3, 2^n, n!, n^n ซึ่งเมื่อขนาดของข้อมูลนำเข้า(n) มีขนาดใหญ่ขึ้น อัตราการเพิ่มค่าของฟังก์ชันจะเพิ่มขึ้น

จากรูป จะเห็นว่า ฟังก์ชัน log n, n, n log n จะมีอัตราการเจริญเติบโตที่เป็นไปอย่างช้า ๆ แต่ฟังก์ชัน n^2, n^3, และ 2^n จะมีอัตราการเจริญเติบโตของฟังก์ชันอย่างรวดเร็ว ดังนั้น เราจึงต้องการขั้นตอนวิธีในรูปของฟังก์ชัน 1, log n, n, n log n มากกว่าในรูปฟังก์ชัน n^2, n^3, และ 2^n

ถ้าคอมพิวเตอร์สามารถประมวลผลคำสั่งได้ 1000 ล้านคำสั่งต่อวินาที กับขนาดข้อมูล 10000 จำนวน สำหรับขั้นตอนวิธีที่อยู่ในรูปของฟังก์ชัน log n, n จะใช้เวลาไม่ถึงวินาที ในขณะที่ต้องใช้เวลาถึง 115.7 วัน ถ้าเป็นฟังก์ชั่น n^4 และถ้าเป็น 2^n แล้วล่ะก็ต้องใช้เวลาเป็นล้านๆปีเลย

ดังนั้นการวิเคราะห์ประสิทธิภาพของขั้นตอนวิธี ก็คือการประกันได้ว่าจะสามารถประมวลผลแล้วเสร็จในเวลาไม่เกินเท่าไร นั่นก็คือการหาว่าขั้นตอนวิธีนั้นมีฟังก์ชัน Big-O เป็นอย่างไร

Times on a 1 billion instruction per second

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01µs	.03µs	.1µs	1µs	10µs	10sec	1µs
20	.02µs	.09µs	.4µs	8µs	160µs	2.84hr	1ms
30	.03µs	.15µs	.9µs	27µs	810µs	6.83d	1sec
40	.04µs	.21µs	1.6µs	64µs	2.56ms	121.36d	18.3min
50	.05µs	.28µs	2.5µs	125µs	6.25ms	3.1yr	13d
100	.10µs	.66µs	10µs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00µs	9.96µs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10.00µs	130.03µs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100.00µs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$yr	$3.17*10^{43}$yr	

µs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years

ตัวอย่าง 1.1 การหาฟังก์ชัน f(n) ต่าง ๆ ในรูป Big-O

ก. $f_1(n) = 5 = O(1)$ เนื่องจาก $5 < 6(1)$ เมื่อ $c = 6,\ g(n) = 1$

ข. $f_2(n) = 3n+2 = O(n)$ เนื่องจาก $3n+2 < 4n$ เมื่อ $n \geq 3$

ค. $f_3(n) = 10n^2+4n+2 = O(n^2)$ เนื่องจาก $10n^2+4n+2 < 11n^2$ เมื่อ $n \geq 6$

ง. $f_4(n) = 3n^3+2n^2 = O(n^3)$ เนื่องจาก $3n^3+2n^2 < 5n^3$ เมื่อ $n \geq 5$

จ. $f_5(n) = 6*2^n+ n^2 = O(2^n)$ เนื่องจาก $6*2^n+ n^2 < 7(2^n)$ เมื่อ $n \geq 5$

ฉ. $f_6(n) = 4 \log n = O(\log n)$ เนื่องจาก $4 \log n < 5 \log n$ เมื่อ $n > 0$

คุณสมบัติของ Big-O

1. ถ้า $f(n) = O(g(n))$ และ $g(n) = O(h(n))$ แล้ว $f(n) = O(h(n))$
2. $O(f(n) + g(n)) = \max\{O(f(n)), O(g(n))\}$
3. $f(\log_a n) = O(\log_2 n)$ สำหรับ $a > 1$

1.5 การนับจำนวนครั้งของการเปรียบเทียบและกำหนดค่า

การนับจำนวนครั้งของการเปรียบเทียบและกำหนดค่า สามารถทำได้โดย

1. นับจำนวนครั้งของการเปรียบเทียบและกำหนดค่าของแต่ละคำสั่ง (statement)
2. รวมจำนวนครั้งที่ได้จากแต่ละคำสั่งเข้าด้วยกัน

ตัวอย่าง 1.2 แสดงการนับจำนวนครั้งของการเปรียบเทียบของการบวกค่า 1 ถึง n

คำสั่ง	จำนวนครั้งของการเปรียบเทียบหรือกำหนดค่า

int sum = 0;
for (i=1; i<=n; i++)
sum = sum + i;
	1
	n+1
	n

รวม	2n+2 = O(n)

ตัวอย่าง 1.3 แสดงการนับจำนวนครั้งของการเปรียบเทียบและการกำหนดค่าของการบวกเมตริกซ์

คำสั่ง	จำนวนครั้งของการเปรียบเทียบหรือกำหนดค่า

for(i= 0; i < row; i++)
for(j=0; j< col; j++)
c[i][j] = a[i][j]+b[i][j];
	row+1
	row(col+1)
	row*col

รวม	2*row*col + 2*row +1 = $O(row^2)$ เมื่อ row > col หรือ = $O(col^2)$ เมื่อ col > row

ตัวอย่าง 1.4 จำนวนครั้งของขั้นตอนวิธี selective sort

วิธีทำ จำนวนครั้งจะเท่ากับ

คำสั่ง	จำนวนครั้ง

public static void selectiveSort(int x[], int n)
{int i,j,k;
for(i=1;i<n;++i)
{
for(j=i+1;j<=n;++j)
{ if(x[i]>x[j])
{k=x[i];
x[i]=x[j];
x[j]=k;
} // if
}// next j
}//next i
}//selective
	$\sum_{i=1}^{i=n-1}$
	$\sum_{j=i+1}^{n} 1$

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

$O(n^2)$

ตัวอย่าง 1.5 จงนับจำนวนครั้งของการคำนวณฟังก์ชัน max และทำผลลัพธ์ให้อยู่ในรูป Big-O

ก.

```
for ( i = 1; i < n; i++ )
     for ( j = 1; j < n; j++ )
          for ( k = j; k < n; k++ )
               A[k] = max(A[k],A[2k]);
```

<u>วิธีทำ</u>

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-1}\sum_{k=j}^{n-1}1 = \sum_{i=1}^{n-1}\sum_{j=1}^{n-1}(n-1-j+1)$$ (พิจารณาในทำนองเดียวกับ $\sum_{i=2}^{5}1=5-2+1=4$)

$$= \sum_{i=1}^{n-1}\sum_{j=1}^{n-1}(n-j)$$
$$= \sum_{i=1}^{n-1}((n-1)+(n-2)+\ldots+2+1)$$
$$= \sum_{i=1}^{n-1}(1+2+\ldots+(n-2)+(n-1))$$
$$= \sum_{i=1}^{n-1}\sum_{j=1}^{n-1} j$$
$$= \sum_{i=1}^{n-1}\frac{n(n-1)}{2}$$ เนื่องจาก $\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$
$$= \sum_{i=1}^{n-1}\frac{n^2-n}{2}$$
$$= \frac{(n-1)(n^2-n)}{2}$$ เนื่องจาก $\sum_{i=1}^{n-1}1=n-1$
$$= \frac{n^3-2n^2+n}{2} = O(n^3)$$

ข. for (i = 0; i <= n; i++)

```
for ( i = 0; i <= n; i++ )
{ A[i] = max( A[2i] , A[2i+1] );
    for ( j = 1; j <= n-p; j++ )
        for ( k = 1; k <= p; k++ )
            A[k] = max( A[2k] , A[2k+1] ); }
```

<u>วิธีทำ</u>

$$\sum_{i=0}^{n}(1+(\sum_{j=1}^{n-p}\sum_{k=1}^{p}1)) = \sum_{i=0}^{n}(1+(\sum_{j=1}^{n-p}(p-1+1)))$$

$$= \sum_{i=0}^{n}(1+(\sum_{j=1}^{n-p}p))$$

$$= \sum_{i=0}^{n}(1+(n-p)p) \qquad \text{เนื่องจาก } \sum_{j=1}^{n-p}1=n-p$$

$$= \sum_{i=0}^{n}(1+np-p^2)$$

$$= (n+1)(1+np-p^2) \qquad \text{เนื่องจาก } \sum_{i=0}^{n}1=n+1$$

$$= n^2p+n(1+p-p^2)-p^2+1$$

$$= O(n^2) \text{ เมื่อ } n > p \text{ หรือ } O(p^2) \text{ เมื่อ } p > n$$

1.6 ความซับซ้อนของขั้นตอนวิธีเรียกซ้ำ(*Complexity of recursive algorithms)*

คราวนี้จะหา complexity (Big-O) ของขั้นตอนวิธีที่มีการเรียกซ้ำ (recursive) โดยวิเคราะห์กรณีของ merge sort ซึ่งจะแบ่งเป็น 3 ส่วนดังนี้

Divide : ในขั้นตอนการแบ่งครึ่งอาเรย์ จะใช้เวลาเท่ากับค่าคงที่ คือ D(n) = $\Theta(1)$

Conquer : เราจะเรียกการทำซ้ำ (recursive) เพื่อแก้ปัญหาย่อย 2 ส่วน แต่ละส่วนมีขนาดเท่ากับ n/2 และใช้เวลาในการประมวลผล = T(n/2)

Combine : ในขั้นตอนการ merge จะใช้เวลาเท่ากับ $\Theta(n)$

นั่นคือ

$$T(n)=\begin{cases}\Theta(1) & if \quad n=1\\ 2T\left(\dfrac{n}{2}\right)+\Theta(n) & if \quad n>1\end{cases}$$

$$T(n)=\begin{cases}c & if \quad n=1\\ 2T\left(\dfrac{n}{2}\right)+cn & if \quad n>1\end{cases}$$

$$2T(\frac{n}{2})=2\left(2T(\frac{n}{4})+c\frac{n}{2}\right)=2.2T(\frac{n}{4})+cn$$

$$2.2T(\frac{n}{4})=2.2\left(2T(\frac{n}{8})+c\frac{n}{4}\right)=2.2.2T(\frac{n}{8})+cn$$

$$T(n)=2^iT(\frac{n}{2^i})+icn \qquad where\ i=\log_2 n$$

$$T(n)=nT(1)+(\log n)cn$$

$$T(n)=cn+cn\log n$$

$$T(n)=cn(\log n+1)$$

$$\mathrm{O}(n\log n)$$

ตัวอย่าง1.6 จงแทนความสัมพันธ์ซ้ำ (recurrence) ในรูปแบบทั่วไป และทำผลลัพธ์ให้อยู่ในรูป Big-O

$$\mathrm{T(n)=2T(n-1)+1} \qquad \text{เมื่อกำหนดให้} \quad \mathrm{T(1)=1}$$

<u>วิธีทำ</u>

$$\begin{aligned}\mathrm{T(n)} &= \mathrm{2T(n-1)+1}\\ &= \mathrm{2(2T(n-2)+1)+1=2^2T(n-2)+2+1}\\ &= \mathrm{2(2(2T(n-3)+1)+1)+1=2^3T(n-2)+4+2+1}\\ &= \vdots\\ &= \mathrm{2^{n-1}T(1)+2^{n-2}+2^{n-3}+\cdots+2+1}\\ &= \mathrm{2^n-1} \qquad \text{เนื่องจาก} \quad \sum_{i=0}^{n-1}2^i=2^n-1\\ &= \mathrm{O(2^n)}\end{aligned}$$

หลังจากปูพื้นฐานวิธีการวิเคราะห์ complexity ของ algorithms ในรูปแบบ การวนซ้ำ(loop) และแบบการเรียกซ้ำ(recursive) แล้ว ในบทถัดๆไป เราจะนำความรู้ที่ได้ไปวิเคราะห์ complexity ของขั้นตอนวิธีที่ใช้กับปัญหาต่างๆ

Mathematics Functions and notations

Floors and ceiling

$x-1 < \lfloor x \rfloor \le \lceil x \rceil < x+1$

$\lceil n/2 \rceil + \lfloor n/2 \rfloor \quad = n$

$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil$

$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor$

$\lceil a/b \rceil \ \le \ (a+(b-1))/b$

$\lfloor a/b \rfloor \ \ge \ (a-(b-1))/b$

Modular arithmetic

$a \bmod n = a - \lfloor a/n \rfloor n$

อนุกรม(summation)

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} c = cn$$

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

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

$$\sum_{i=0}^{n} a^i \le \frac{1}{1-a} \quad ; 0 < a < 1$$

$$\sum_{i=1}^{n} \frac{1}{i} = \log n$$

$$\sum_{i=1}^{n} (a+id) = na + d\,\frac{n(n+1)}{2}$$

$$1 - r + r^2 - r^3 + \cdots = \sum_{i=0}^{\infty} (-r)^i = \frac{1}{(1+r)}$$

$$\sum_{i=0}^{\infty} ir^{i-1} = \frac{1}{(1+r)^2}$$

$$\sum_{i=0}^{\infty} \frac{r^i}{i} = -\ln(1-r)$$

เลขยกกำลัง (exponentials)

$$x^a x^b = x^{a+b}$$

$$\frac{x^a}{x^b} = x^{a-b}$$

$$(x^a)^b = x^{ab}$$

$$x^n + x^n = 2x^n$$

$$2^n + 2^n = 2^{n+1}$$

logarithm

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(\frac{x}{y}) = \log_b x - \log_b y$$

$$\log_b(x^n) = n\log_b x$$

$$\log_b x = \frac{1}{\log_x b}$$

$$\log_b x = \log_b a \log_a x$$

$$x^{\log_b y} = y^{\log_b x}$$

$$a^{\log_b c} = c^{\log_b a}$$

แบบฝึกหัด

1. จงหาฟังก์ชัน f(n) ต่าง ๆ ในรูป Big-O

ก. $f(n) = 4n^2 + n - 1$

ข. $f(n) = \sqrt{n}$

ค. $f(n) = n^3 + 8n^2 + 2^n$

ง. $f(n) = \log_2 n + 3n + 50$

จ. $f(n) = 6\log_2 n + 9n$

2. จงนับจำนวนครั้งของการเปรียบเทียบและกำหนดค่าของการคูณและการทรานสโพสเมตริกซ์และทำผลลัพธ์ให้อยู่ในรูป Big-O

ก.
```
for( i = 0;  i < n;  i++ )
        for( j = 0; j < n; j++ )
            for( k = a[i][j] = 0;  k < n;  k++)
                a[i][j] += b[i][k] * c[k][j];
```

ข.
```
for( i = 0;  i < n-1;  i++ )
        for( j = i+1; j < n; j++ )
        {    tmp = a[i][j];
             a[i][j] = a[j][i];
             a[j][i] = tmp;
        }
```

3. จงนับจำนวนครั้งของการเพิ่มค่า cnt และทำผลลัพธ์ให้อยู่ในรูป Big-O

ก.
```
for( cnt1 = 0, i = 1;  i <= n;  i++ )
        for( j = 1; j <= n; j++ )
             cnt1++;
```

ข.
```
for( cnt2 = 0, i = 1;  i <= n;  i++ )
        for( j = 1; j <= i; j++ )
             cnt2++;
```

ค.
```
for( cnt3 = 0, i = 1;  i <= n;  i *= 2 )
        for( j = 1; j <= n; j++ )
             cnt3++;
```

ง.
```
for( cnt4 = 0, i = 1;  i <= n;  i *= 2 )
        for( j = 1; j <= i; j++ )
             cnt4++;
```

4. จงแทนความสัมพันธ์ซ้ำ (recurrence) ในรูปแบบทั่วไป และทำผลลัพธ์ให้อยู่ในรูป Big-O

ก. $T(n) = T(n-1) + n$ เมื่อ $n \geq 2$ และ $T(1) = 1$

ข. $T(n) = T(n/2) + 1$ เมื่อ $n \geq 2$ และ $T(1) = 0$

ค. $T(n) = T(n/2) + n$ เมื่อ $n \geq 2$ และ $T(1) = 0$

ง. $T(n) = 2T(n/2) + n$ เมื่อ $n \geq 2$ และ $T(1) = 0$

จ. $T(n) = 2T(n/2) + 1$ เมื่อ $n \geq 2$ และ $T(1) = 0$