2.1 A Brief Introduction to Recursion :

Mathematical function

1. C=2(F-32)/9

2. y = sin(x) * pi

3. f(x) = 2f(x-1)+x^2

   f(0)=0 , x nonnegative integer

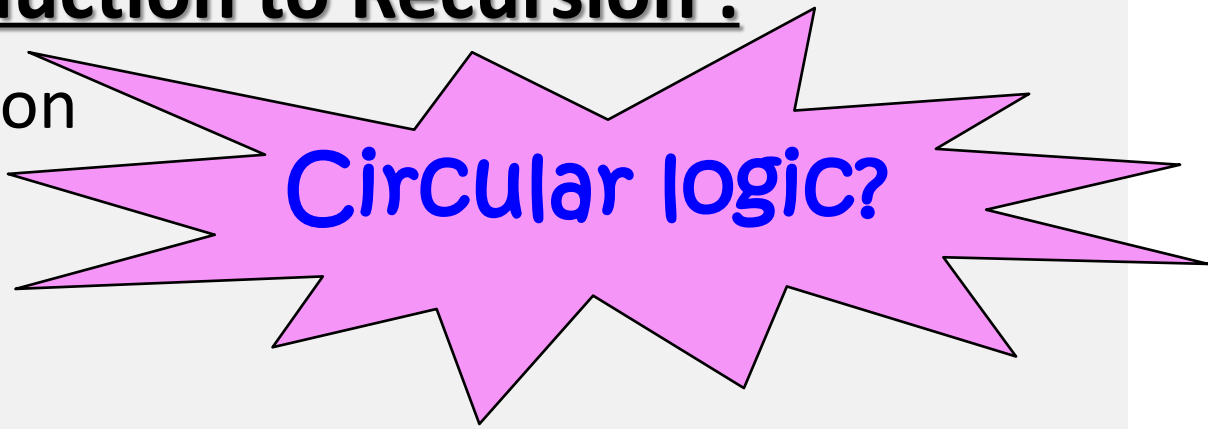
1. ส่วน recursive
2. ส่วน basecase

$f(x-1) = 2f(x-2)+x^2$

2.1 A Brief Introduction to Recursion :

Mathematical function

1. C=2(F-32)/9



2. y = sin(x) * pi

3. $f(x) = 2f(x-1)+x^2$

   f(0)=0 , x nonnegative integer

4. Factorial กำหนด x เป็นจำนวนเต็มที่ไม่เป็นลบ

f(x) = x * f(x-1) , f(1) = 1 และ f(0)=1

5. Fibonacci number

f(n) = f(n-1) + f(n-2)

f(0)=0 , f(1) = 1

n = 3

f(3) = f(2) + f(1)

f(2) = f(1) + f(0)

Example 1

```
#include <stdio.h>
int fact(int x)
{   if(x <= 1)
        return 1;
    else
        return x* fact(x-1);
}
```

return type

```
int main()
{   int ans;
    ans = fact(3);
    cout << ans;
}
```

4!

4! = 4 × 3!

3! = 3 × 2!

2! = 2 × 1!

1! = 1 × 0!

0! = 1

Example 2 Factorial

```
int main()
{   int ans;
    ans = 6
    cout << ans;
}
```

```
int fact(int x)   //2
{  if(x <= 1)
       return 1;
   else
       return x* 1
}
```

2 * fact(1)

```
int fact(int x)   //3
{  if(x <=1)
       return 1;
   else
       return x* 2
}
```

3 * fact(2)

```
int fact(int x)   //1
{  if(x <= 1)
       return 1;
   else
       return x* fact(x-1);
}
```

Example 3

```
0 int bad(int n)
1 {   if (n==0)
2         return 0;
3    else
4         return bad(n/3 + 1+ n – 1);
5 }
```

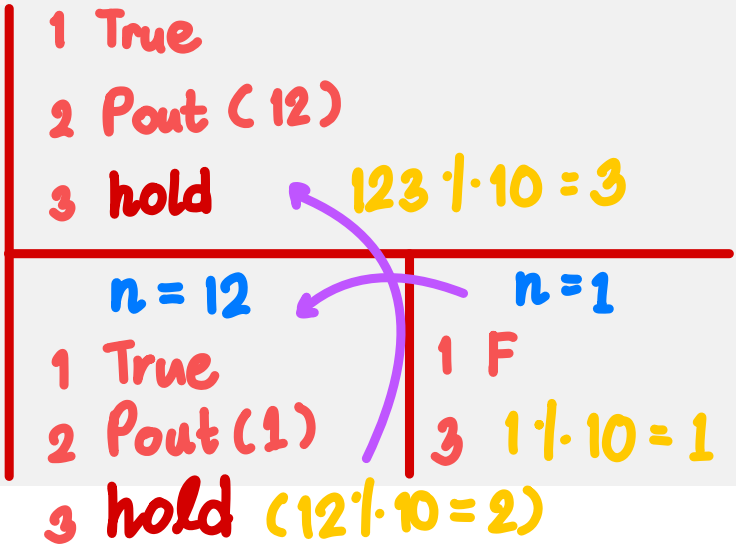
stack OVerflow

Example 4

n = 1234

```
0 void printout(int n)
1 {   if( n >=10 )
2         printout(n/10);
3      cout << n%10;
4 }
```

n = 123



Example 5 n = 123

```
0  void printout(int n)
1  {   if( n >=10 )
2       {  printout(n/10);
3            cout << n%10;
4       }
5  }
```

n = 123

1. T
2. Pout (12) a
3. ต่าง

---

n = 12

1. T
2. Pout (1) b
3. ต่าง

---

n = 1

1. F

การบ้าน 1

1. จงเขียนโปรแกรมหาค่า Fibonacci กำหนด function Fibonacci ดังนี้ $F_0=0$, $F_1=1$ and $F_n = F_{n-1} + F_{n-2}$ for $n>1$

แสดงผลลัพธ์เป็นค่า fibonacci ตั้งแต่ 0-19

F(0) = 1

F(1) = 1

F(2) = 1

...

F(19) = 4181

การบ้าน 2

2. จากตัวอย่าง printout จงเขียนโปรแกรมรับตัวเลขจำนวนเต็ม 1 ค่า พิมพ์เลขจากหลังไปหน้า และหน้าไปหลัง ด้วยการใช้วิธีการ recursive แสดงผลลัพธ์ 4 ค่า ดังตัวอย่าง

- Input ตย. 123456
- Back to front ตย. 654321
- Back to front cut last ตย. 54321
- Front to back ตย. 123456
- Front to back cut last ตย. 12345

ผลลัพธ์

Input : 123456

Output :

1) 654321
2) 54321
3) 123456
4) 12345

2.2 Algorithm Analysis

Definition : An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.

- correct
- time or space

Example 5 Running time Calculations

```
int sum(int n)
{       int partialSum;
        partialSum=0;                   1
        for(int i=1; i<=n; i++)         1+n+1
                partialSum += i*i*i;    4×n
        return partialSum;              1
}
```

partialsum = partialsum + i×i×i

Example 6 Maximum subsequence sum

4 -3 5 -2 -1 2 6 -2

4

4 + -3

4 + -3 + 5

4 + -3 + 5 + -2

…

4 + -3 + 5 + -2 + -1 + 2 + 6 + -2

-3

-3 + 5

-3 + 5 + -2

-3 + 5 + -2 + -1

…

-3 + 5 + -2 + -1 + 2 + 6 + -2

Example 6 Maximum subsequence sum

4 -3 5 -2 -1 2 6 -2

```
int MasSubsequenceSum(int a[], int N)
{       int ThisSum, MaxSum, j;
        for(j=0; j<N; j++)
        {       ThisSum += A[j];
                if( ThisSum > MaxSum)
                        MaxSum = ThisSum;
                else if( ThisSum < 0 )
                        ThisSum=0;
        }
        return MaxSum;
}
```

2.2.1 Mathematics Reviews

1) Exponents

$X^A X^B = X^{A+B}$

$\frac{X^A}{X^B} = X^{A-B}$

$(X^A)^B = X^{AB}$

$X^N + X^N = 2X^N \neq X^{2N}$

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

2) Logarithms

$X^A = B$ -> $\log_x B = A$

$\log_A B = \frac{\log_c A}{\log_c B}$

3) Series

$1 + 2 + 3 + \ldots + N$ = $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

$1^2 + 2^2 + 3^2 + \ldots + N^2$ = $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$

คำถาม

$1 + 2 + 3 + \ldots + (N-1) = ?$

$$\frac{(N-1)(N-1+1)}{2} = \frac{N(N-1)}{2} \#$$

2.2.2) Mathematical Background

Big O Notation : เป็น Notation หรือรูปแบบ หรือระบบทางคณิตศาสตร์

- ใช้ในการ อธิบายประสิทธิภาพของอัลกอริทึม หรือฟังก์ชัน
- เมื่อขนาดของข้อมูลอินพุต (N) มีขนาดใหญ่ขึ้นมากๆ

Definition : T(N)=O(f(N)) if there are positive constants c and n_0 such that T(N) ≤ cf(N) when N ≥ n_0.

T(N): ฟังก์ชันที่แทนเวลาจริง หรือ จำนวนการดำเนินการจริง ที่อัลกอริทึมใช้ในการทำงาน โดยขึ้นอยู่กับขนาดของข้อมูลอินพุต N

F(N): คือฟังก์ชันที่ใช้ เป็นตัวแทนหรือขีดจำกัดบน (upper bound) ของการเติบโตของ T(N) , F(N) มักจะเป็นฟังก์ชันที่ง่ายกว่า เช่น N, N^2 , logN, NlogN, หรือ 1 (ค่าคงที่)

เราเลือก F(N) ที่เป็นฟังก์ชันที่ครอบคลุมการเติบโตของ T(N) ในกรณีที่ N มีค่ามากพอ

Definition : T(N)=O(f(N)) if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

(6N+4)(4.N)

c (positive constant): คือ ค่าคงที่ที่เป็นบวก มีไว้เพื่อ "ปรับขนาด" หรือ "ขยาย" ฟังก์ชัน f(N) ให้ครอบคลุม T(N) ได้

เมื่อ N มีค่ามากพอ T(N) ก็จะไม่มีทางเกิน c·F(N)

เราไม่สนใจประสิทธิภาพของอัลกอริทึมเมื่อ N มีค่าน้อยๆ

แต่เราสนใจเมื่อ N โตขึ้นมากๆ ว่าอัลกอริทึมจะยังคงมีประสิทธิภาพดีอยู่หรือไม่

Definition : T(N)=O(f(N)) if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

- Give two functions
- we compare their relative rates of growth.

Although 1000N is larger than N^2 for small value of N, N^2 grow at a faster rate, and thus N^2 will eventually be the larger function.

T(N)=O(f(N)) if there are positive constants c and n_0 such that
T(N) ≤ cf(N) when N ≥ n_0.

- T(N) = 1000N
- F(N) = N^2
- เมื่อ N=1, c=1 จะเห็นว่า 1000N > N^2
- เมื่อ N มีค่ามากขึ้น จนถึง 1000 1000N = N^2
- เมื่อ N มากกว่า 1000 1000N <= N^2

We can say that 1000N = O(N^2)

1000N เป็นฟังก์ชันที่โตไม่เร็วกว่า N^2

T(N)=O(f(N)) if there are positive constants c and n_0 such that T(N) ≤ cf(N) when N ≥ n_0.

ตัวอย่างเพิ่มเติม

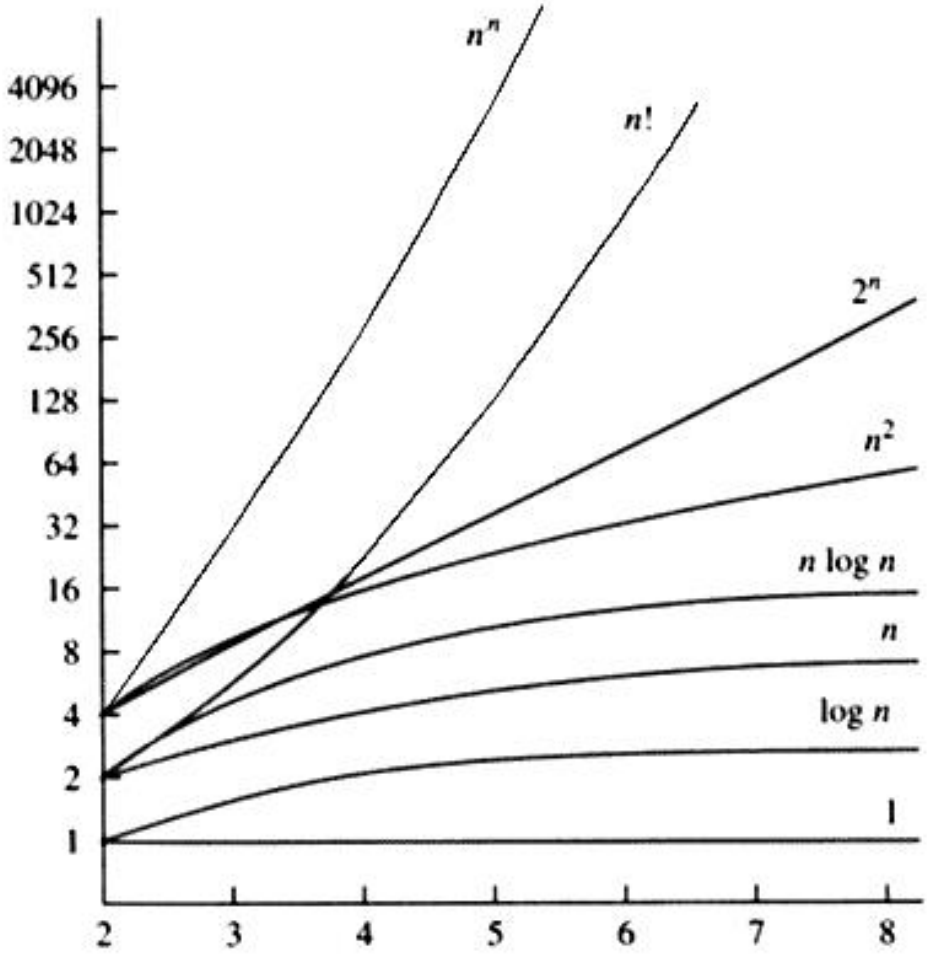
- $T(N) = 2N^2 + 4$
- $F(N) = N^2$
- เมื่อ N=1, c=3 จะเห็นว่า $2N^2 + 4 > 3*N^2$
- เมื่อ N =2 จะเห็นว่า $2N^2 + 4 <= 3*N^2$

  เมื่อ N มากกว่า 2 จะเห็นว่า $2N^2 + 4 <= 3*N^2$

We can say that $2N^2 + 4 = O(N^2)$

ตามนิยามตัด c ออก

Function	Name
C	Constant
logN	Logarithmic
$\log^2 N$	Log-squared
N	Linear
NlogN	
N^2	Quadratic
N^3	Cubic
2^N	Exponential



2.2.3) General Rules

Rule 1– For loop

The running time of a for loop is at most the running time of the statements inside the for loop(including tests) times the number of iterations.

```
for(i=0; i<n; i++)      1+n+1+n  = 2n+2
        k++;            n        O(N)
```

Rule 2– Nested loops

Analyze these inside out. The total running time of a statement inside a group of nested loop is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for(i=0; i<n; i++)
        for(j=0; j<n; j++)          O(N^2)
                k++;
```

Rule 3– Consecutive statement For loop

Add.

```
for(i=0; i<n; i++)
        a[i]=0;
for(i=0; i<n; i++)
        for(j=0; j<n; j++)
                a[i]+=a[j]+i+j;
```

$O(N+N^2)$

$O(N^2)$

Rule 4– If/Else

The running time of an if/else statement is never more than the running time of the test plus the running times of S1 and S2

```
if (condition)  ✓
        S1      ✓
else
        S2
```

O(2)

O(1)

คำถาม จาก code ด้านล่าง จงหาค่า bigO

1.

```
int example(int numbers[], int size)
{
    int total = 0;
    for (int i = 0; i < size; i++)
    {
        total += numbers[i];
    }
    return total;
}
```

O(N)

คำถาม จาก code ด้านล่าง จงหาค่า bigO

2.

```
int example(int matrix[][3], int rows, int cols) {
    int total = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            total += matrix[i][j];
        }
    }
    return total;
}
```

$O(N^2)$

คำถาม จาก code ด้านล่าง จงหาค่า bigO

3.

```
int example(int numbers[], int size)
{     if (size > 0)
            return numbers[0];
       else
            return -1;
}
```

O(1)