

CHAPTER 2

ALGORITHM ANALYSIS

【Definition】 An **algorithm** is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

- (1) **Input** There are zero or more quantities that are externally **supplied**.
- (2) **Output** At least one quantity is **produced**.
- (3) **Definiteness** Each instruction is **clear and unambiguous**.
- (4) **Finiteness** If we trace out the instructions of an algorithm, then for all cases, the algorithm **terminates** after **finite number of steps**.
- (5) **Effectiveness** Every instruction must be basic enough to **be carried out**, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in(3); it also must be **feasible**.

Note: A **program** is written in some programming language, and does not have to be finite (*e.g. an operation system*).

An **algorithm** can be described by human languages, flow charts, some programming languages, or pseudo-code.

[[Example]] **Selection Sort:** Sort a set of $n \geq 1$ integers in increasing order.

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

Where?

Where and how
are they stored?

Note: A **program** is written in some programming language, and does not have to be finite (*e.g. an operation system*).

An **algorithm** can be described by human languages, flow charts, some programming languages, or pseudo-code.

[[Example]] **Selection Sort:** Sort a set of $n \geq 1$ integers in increasing order.

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

```
for ( i = 0; i < n; i++) {
```

Where?

Where and how
are they stored?

```
    Examine list[i] to list[n-1] and suppose that the smallest  
    integer is at list[min];
```

```
    Interchange list[i] and list[min];
```

Algorithm in
pseudo-code

Sort = Find the smallest integer + Interchange it with list[i].

§ 1 What to Analyze

- Machine & compiler-dependent **run times**.
- **Time & space complexities** : machine & compiler-independent.

- **Assumptions:**

- ① instructions are executed sequentially
- ② each instruction is **simple**, and takes exactly **one time unit**
- ③ integer size is fixed and we have infinite memory

- Typically the following two functions are analyzed:

$T_{\text{avg}}(N)$ & $T_{\text{worst}}(N)$ -- the average and worst case time complexities, respectively, as functions of **input size N** .

If there is more than one input, these functions may have more than one argument.

[[Example]] Matrix addition

```
void add ( int a[ ][ MAX_SIZE ],
           int b[ ][ MAX_SIZE ],
           int c[ ][ MAX_SIZE ],
           int rows, int cols )
{
    int i, j ;
    for ( i = 0; i < rows; i++ )    /* rows + 1 */
        for ( j = 0; j < cols; j++ ) /* rows(cols+1) */
            c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ]; /* rows · cols */
}
```

$$T(\text{rows}, \text{cols}) = 2 \text{ rows} \cdot \text{cols} + 2\text{rows} + 1$$

[[Example]] Matrix addition

```
void add ( int a[ ][ MAX_SIZE ],  
          int b[ ][ MAX_SIZE ],  
          int c[ ][ MAX_SIZE ],  
          int rows, int cols )  
{  
    int i, j ;  
    for ( i = 0; i < rows; i++ )    /* rows + 1 */  
        for ( j = 0; j < cols; j++ ) /* rows(cols+1) */  
            c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ];    /* rows · cols */  
}
```

**Q: What shall we do
if rows >> cols?**

$$T(\text{rows}, \text{cols}) = 2 \text{ rows} \cdot \text{cols} + 2\text{rows} + 1$$

[[Example]] Matrix addition

```
void add ( int a[ ][ MAX_SIZE ],  
          int b[ ][ MAX_SIZE ],  
          int c[ ][ MAX_SIZE ],  
          int rows, int cols )  
{  
    int i, j ;  
    for ( i = 0; i < rows; i++ )    /* rows + 1 */  
        for ( j = 0; j < cols; j++ ) /* rows(cols+1) */  
            c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ]; /* rows · cols */  
}
```

**A: Exchange
rows and cols.**

$$T(\text{rows}, \text{cols}) = 2 \text{ rows} \cdot \text{cols} + 2\text{rows} + 1$$

[[Example]] Iterative
function for summing
a list of numbers

$$T_{sum}(n) = 2n + 3$$

```
float sum ( float list[ ], int n )  
{ /* add a list of numbers */  
  float tempsum = 0; /* count = 1 */  
  int i ;  
  for ( i = 0; i < n; i++ )  
    /* count ++ */  
    tempsum += list [ i ] ; /* count ++ */  
  /* count ++ for last execution of for */  
  return tempsum; /* count ++ */  
}
```

[[Example]] Iterative
function for summing
a list of numbers

$$T_{sum}(n) = 2n + 3$$

```
float sum ( float list[ ], int n )
{ /* add a list of numbers */
  float tempsum = 0; /* count = 1 */
  int i ;
  for ( i = 0; i < n; i++ )
    /* count ++ */
    tempsum += list [ i ]; /* count ++ */
  /* count ++ for last execution of for */
  return tempsum; /* count ++ */
}
```

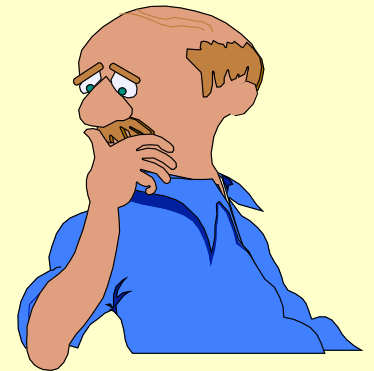
[[Example]] Recursive
function for summing a
list of numbers

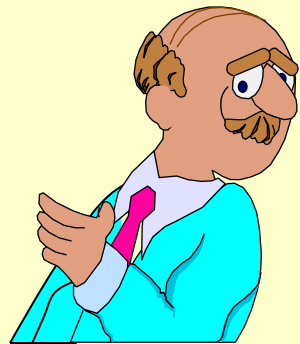
$$T_{rsum}(n) = 2n + 2$$

But it takes more time to
compute each step.

```
float rsum ( float list[ ], int n )
{ /* add a list of numbers */
  if ( n ) /* count ++ */
    return rsum(list, n-1) + list[n - 1];
  /* count ++ */
  return 0; /* count ++ */
}
```

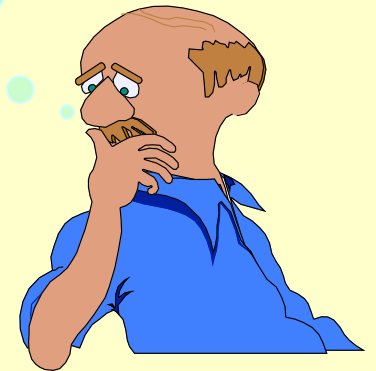
**Is it really necessary
to count the exact
number of steps ?**

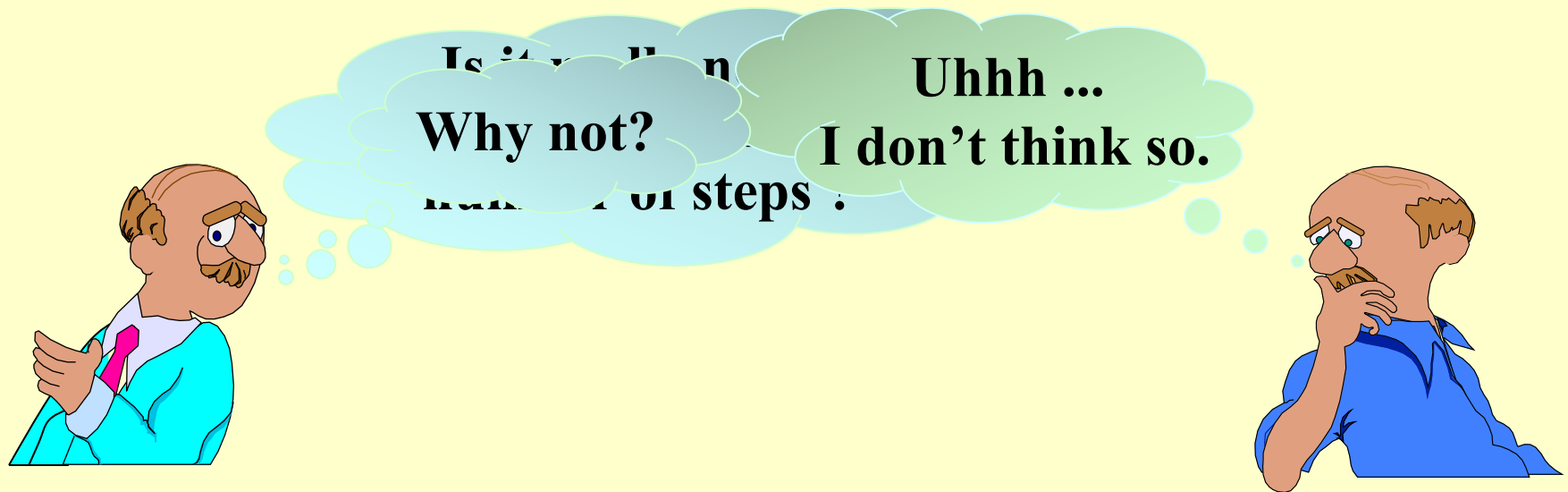


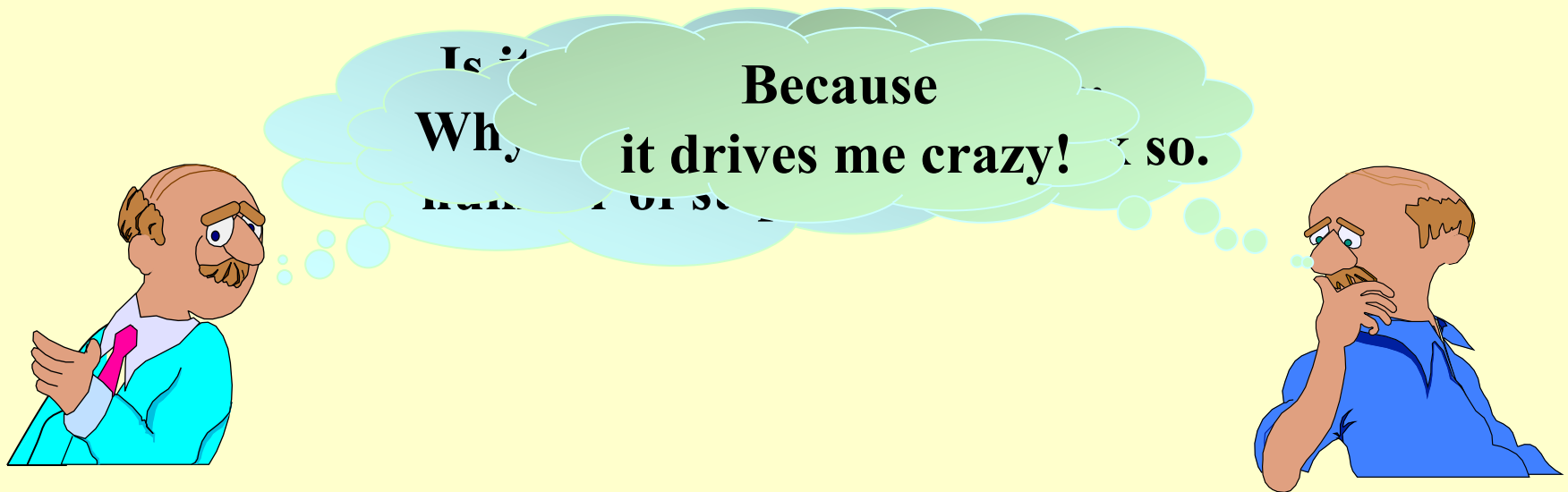


Is it really necessary
to count the exact
number of steps?

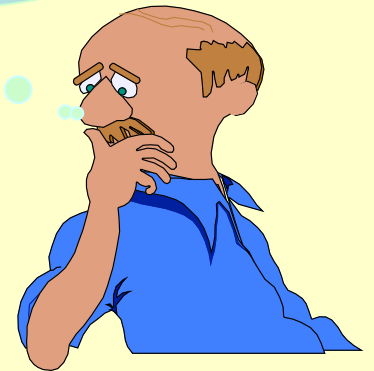
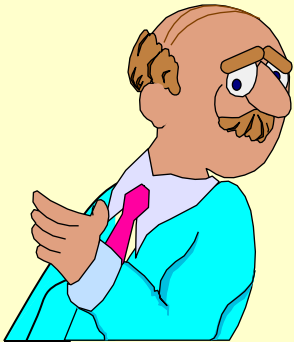
Uhhh ...
I don't think so.



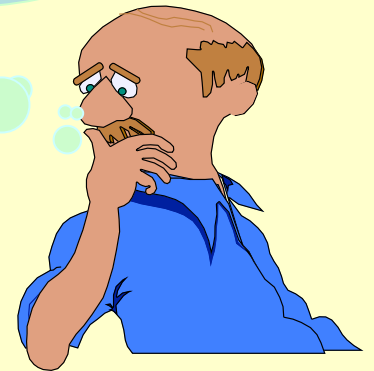
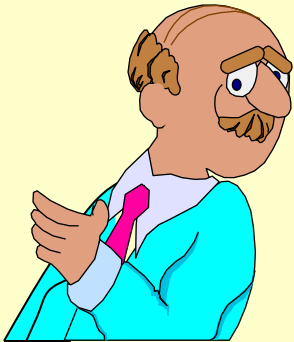




**So it's too complicated sometimes.
But does it worth the effort?**



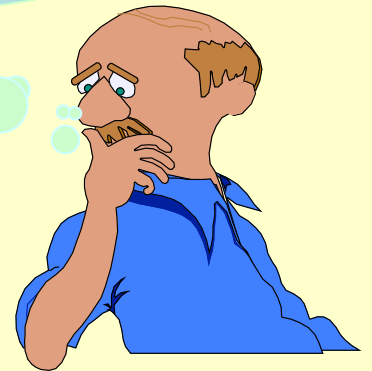
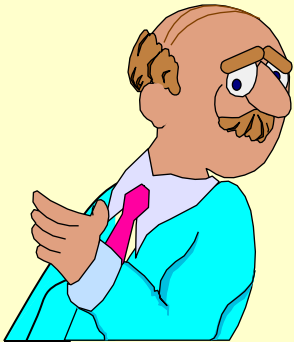
Take the iterative and recursive programs for summing a list for example --- if you think $2n+2$ is less than $2n+3$, try a large n and you'll be surprised !



Take the iterative and
recursive programs for summing
a list for example — if you think $2n+2$ is

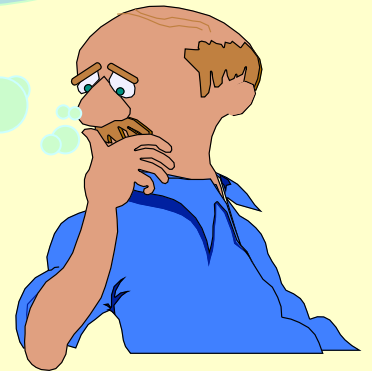
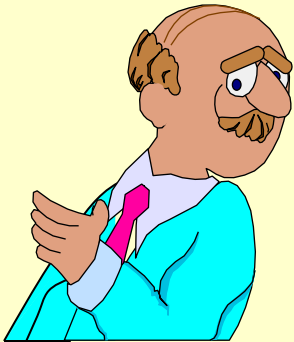
I see ...

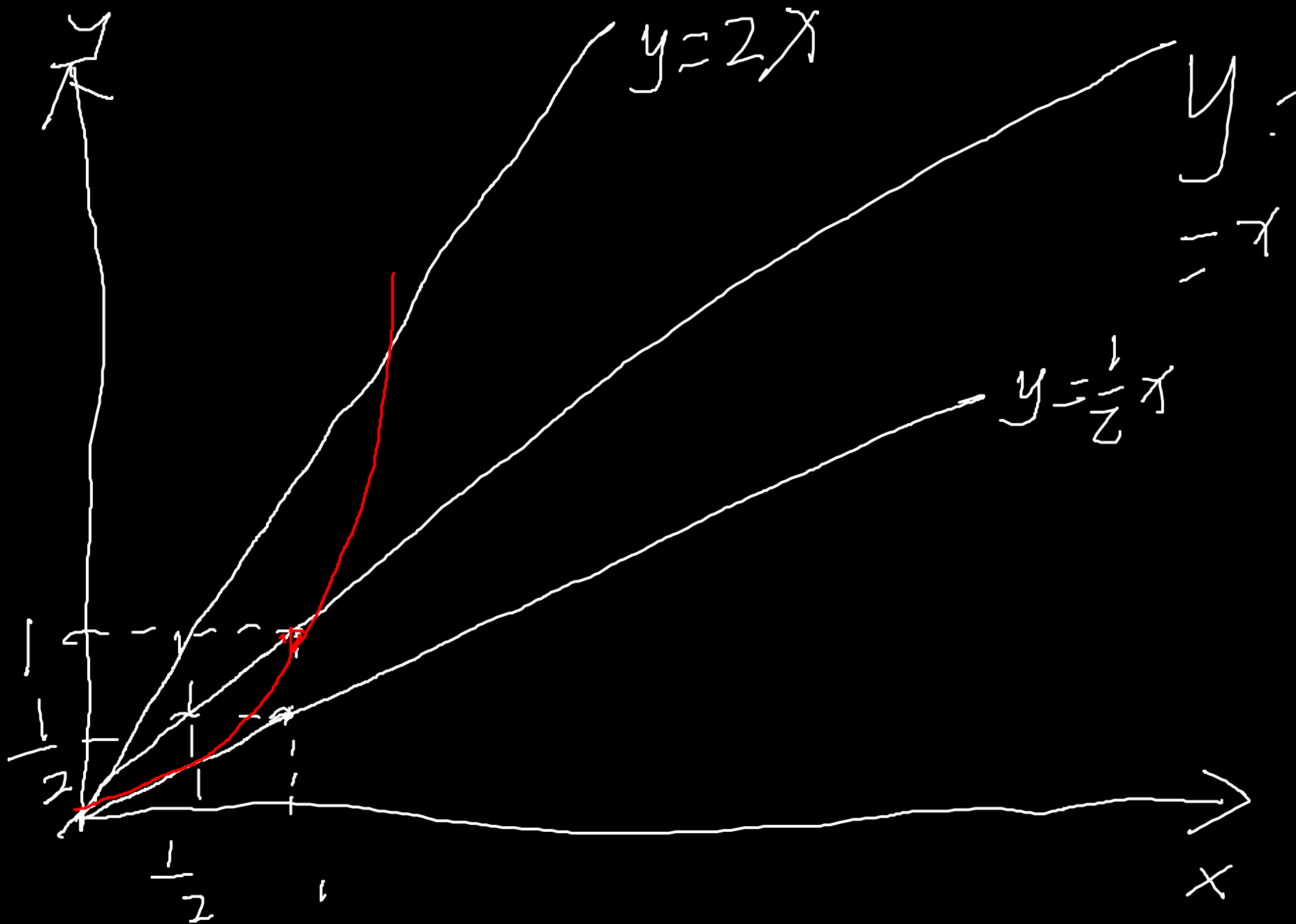
Then what's the point of
this T_p stuff?



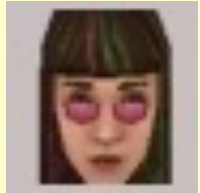
Take the iterative and
recursive programs for summing
a list for example, if you think $2n+2$ is

Good question !
Let's ask the students ...





§ 2 Asymptotic Notation (O , Ω , Θ , o)



The point of counting the steps is to **predict the growth** in run time as the N change, and thereby **compare the time complexities of two programs**. So what we really want to know is the **asymptotic behavior** of T_p .

Suppose $T_{p1}(N) = c_1N^2 + c_2N$ and $T_{p2}(N) = c_3N$. Which one is faster?

No matter what c_1 , c_2 , and c_3 are, there will be an n_0 such that $T_{p1}(N) > T_{p2}(N)$ for all $N > n_0$.



I see! So as long as I know that T_{p1} is **about** N^2 and T_{p2} is **about** N , then for **sufficiently large** N , P2 will be faster!

【Definition】 $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c \cdot f(N)$ for all $N \geq n_0$.

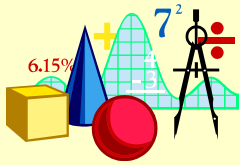
【Definition】 $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c \cdot g(N)$ for all $N \geq n_0$.

【Definition】 $T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

【Definition】 $T(N) = o(p(N))$ if $T(N) = O(p(N))$ and $T(N) \neq \Theta(p(N))$.

Note:

- $2N + 3 = O(N) = O(N^{k \geq 1}) = O(2^N) = \dots$ We shall always take the **smallest** $f(N)$.
- $2^N + N^2 = \Omega(2^N) = \Omega(N^2) = \Omega(N) = \Omega(1) = \dots$ We shall always take the **largest** $g(N)$.



Rules of Asymptotic Notation

☞ If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

(a) $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,

(b) $T_1(N) * T_2(N) = O(f(N) * g(N))$.

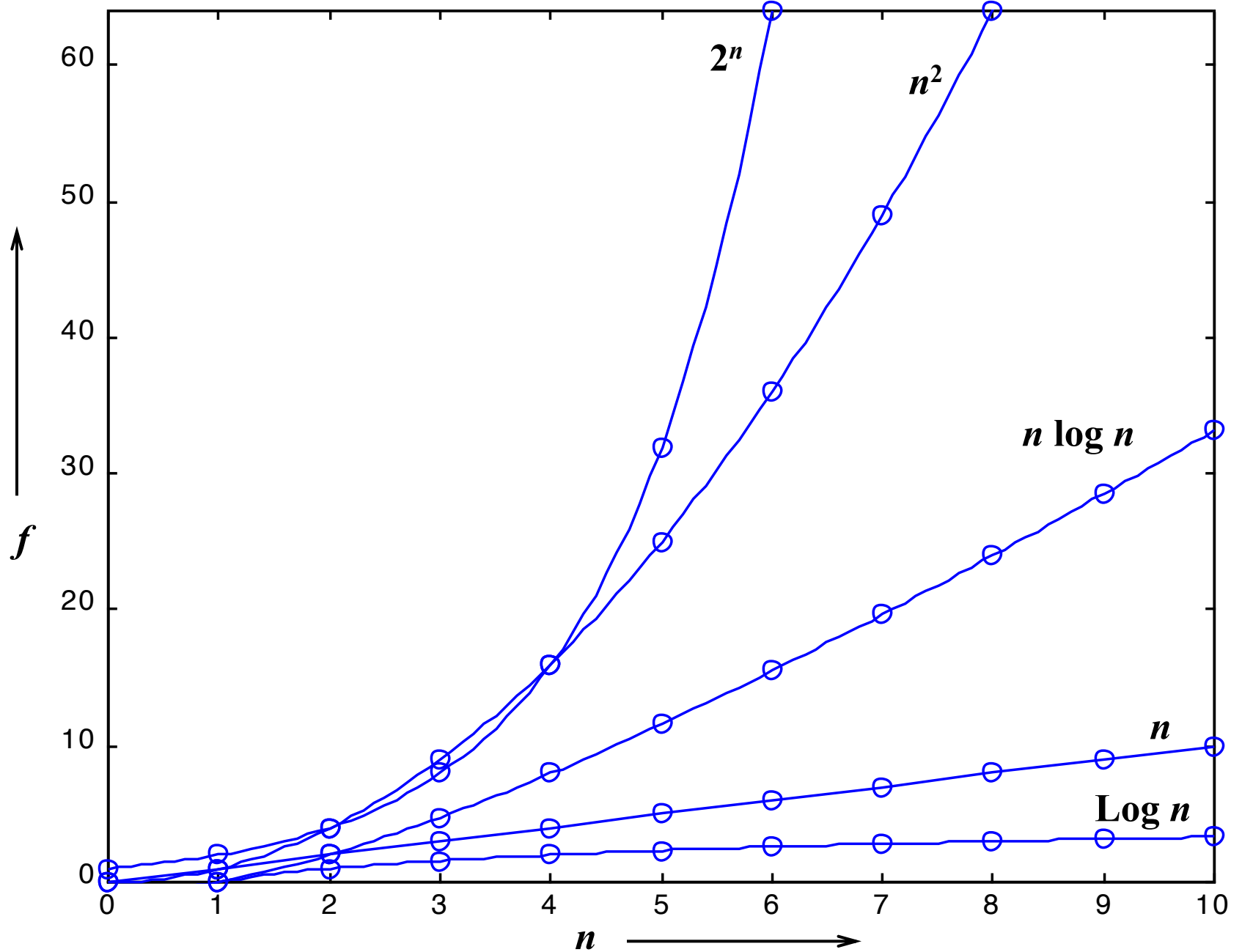
☞ If $T(N)$ is a polynomial of degree k , then $T(N) = \Theta(N^k)$.

☞ $\log^k N = O(N)$ for any constant k . This tells us that **logarithms grow very slowly**.

Note: When compare the complexities of two programs asymptotically, make sure that N is **sufficiently large**.

For example, suppose that $T_{p1}(N) = 10^6 N$ and $T_{p2}(N) = N^2$. Although it seems that $\Theta(N^2)$ grows faster than $\Theta(N)$, but if $N < 10^6$, P2 is still faster than P1.

		Input size 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	2092278988000	26313×10^{33}



	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10sec	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84hr	1ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	6.83d	1sec
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56ms	121.36d	18.3min
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25ms	3.1yr	13d
100	.10 μ s	.66 μ s	10 μ s	1ms	100ms	3171yr	$4 \cdot 10^{13}$ yr
1,000	1.00 μ s	9.96 μ s	1ms	1sec	16.67min	$3.17 \cdot 10^{13}$ yr	$32 \cdot 10^{283}$ yr
10,000	10 μ s	130.03 μ s	100ms	16.67min	115.7d	$3.17 \cdot 10^{23}$ yr	
100,000	100 μ s	1.66ms	10sec	11.57d	3171yr	$3.17 \cdot 10^{33}$ yr	
1,000,000	1.0ms	19.92ms	16.67min	31.71yr	$3.17 \cdot 10^7$ yr	$3.17 \cdot 10^{43}$ yr	

 μ s = microsecond = 10^{-6} secondsms = millisecond = 10^{-3} seconds

sec = seconds

min = minutes

hr = hours

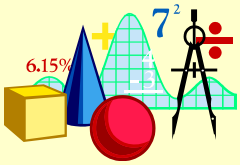
d = days

yr = years

[[Example]] Matrix addition

```
void add ( int a[ ][ MAX_SIZE ],  
           int b[ ][ MAX_SIZE ],  
           int c[ ][ MAX_SIZE ],  
           int rows, int cols )  
{  
    int i, j ;  
    for ( i = 0; i < rows; i++ )    /*  $\Theta(\text{rows})$  */  
        for ( j = 0; j < cols; j++ )    /*  $\Theta(\text{rows} \cdot \text{cols})$  */  
            c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ];    /*  $\Theta(\text{rows} \cdot \text{cols})$  */  
}
```

$$T(\text{rows}, \text{cols}) = \Theta(\text{rows} \cdot \text{cols})$$



General Rules

- 👉 **FOR LOOPS:** 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**.
- 👉 **NESTED FOR LOOPS:** The total running time of a statement inside a group of nested loops is the running time of the **statements multiplied** by the **product of the sizes** of all the for loops.
- 👉 **CONSECUTIVE STATEMENTS:** These just **add** (which means that the **maximum** is the one that counts).
- 👉 **IF / ELSE:** For the fragment

```

if ( Condition ) S1;
else S2;

```

the running time is never more than the running time of the **test plus** the **larger** of the running time of S1 and S2.

👉 RECURSIONS:

[[Example]] Fibonacci number:

$$\text{Fib}(0) = \text{Fib}(1) = 1, \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

```
long int Fib ( int N ) /* T ( N ) */
{
    if ( N <= 1 ) /* O( 1 ) */
        return 1; /* O( 1 ) */
    else
        return Fib( N - 1 ) + Fib( N - 2 );
} /*O(1)*/ /*T(N-1)*/ /*T(N-2)*/
```

Q: Why is it so bad?

$$T(N) = T(N-1) + T(N-2) + 2 \geq \text{Fib}(N)$$

$$\left(\frac{3}{2}\right)^N \leq \text{Fib}(N) \leq \left(\frac{5}{3}\right)^N \rightarrow T(N) \text{ grows exponentially}$$

Proof by induction

