Algoritmos y Estructuras de Datos I

Carreras:

Lic. En Informática

Programador Universitario

Facultad de Ciencias Exactas y Tecnología

2020

Elección de un algoritmo

Cuando se resuelve un problema, frecuentemente se debe elegir entre distintos algoritmos.

¿Sobre que base se elige?.

Dos metas contradictorias:

- un algoritmo simple, fácil de entender, codificar y de rastrear errores.
- un algoritmo que haga uso eficiente de los recursos de la computadora.

Rendimiento de un programa

Hay criterios para juzgar un programa directamente relacionados con la performance del mismo:

- **tiempo de calculo**
- **requisitos de almacenamiento**

Definiciones:

- La ***complejidad de espacio*** es la cantidad de memoria que necesita para realizar su ejecución.
- La ***complejidad de tiempo*** es la cantidad de tiempo de computador que necesita para completar su ejecución.

Rendimiento de un programa

- Por lo tanto el análisis del rendimiento abarcará un análisis de la complejidad de espacio y de la complejidad de tiempo del programa.

Complejidad de espacio de un Programa

El espacio necesario para un programa es la suma:

Una ***parte fija*** que es independiente de las características de la entrada y la salida. Esta parte incluye el espacio para las instrucciones (el espacio para el código), el espacio para variables simples y variables con componentes de tamaño fijo, espacio para constantes etc.

Una ***parte variable*** que consiste del espacio necesario para variables con componentes cuyo tamaño depende de la instancia particular del problema que se está resolviendo, espacio para variables dinámicas referenciadas y espacio para el stack de recursión.

Complejidad de tiempo de un Programa

El tiempo de un programa P, es la suma del tiempo de **compilación** y del tiempo de **ejecución.**

El tiempo de compilación no depende de las características de la instancia.

Se supone que un programa compilado se ejecutará varias veces sin recompilarlo.

Por lo tanto lo mas significativo será el ***tiempo de ejecución.***

Tiempo de ejecución de un Programa

Para dar una medida del tiempo de ejecución de un programa hay que analizar varios factores, entre ellos los mas importantes son:

- la **entrada** del programa, esto es, los datos de entrada.
- el **algoritmo** aplicado.
- la calidad del **código generado por el compilador** usado para crear la imagen objeto.
- la naturaleza y velocidad de las **instrucciones de máquina** usadas para ejecutar el programa.

Complejidad de Tiempo de un programa

- Se llama **T(n)** al *tiempo de ejecución de un programa con una entrada de tamaño n*.
- **T(n)** es la llamada ***complejidad de tiempo del programa***.
- T(n) depende del algoritmo usado.
- Se dice que el tiempo de ejecución es proporcional a T(n), la constante de proporcionalidad depende de la computadora.

Tiempo de ejecución de un algoritmo

- Se denota también con T(n) *el tiempo de ejecución de un algoritmo para una entrada de tamaño n*.

- T(n) es el número de instrucciones ejecutadas por el algoritmo en una computadora ideal, Ej. RAM.

- Qué pasa con el tiempo de ejecución T(n) estimado cuando ese algoritmo se implementa en un programa y se ejecuta en una computadora real? Y si se cambia de lenguaje y/o de máquina?

- Una respuesta a esta pregunta viene dada por el *principio de invariancia* que afirma que dos implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en más de alguna constante multiplicativa.

Tiempo de ejecución de un Programa

Para muchos programas el ***costo no es uniforme*** para varias muestras de tamaño n: se puede encontrar que los tiempos de ejecución de un algoritmo varían para entradas del mismo tamaño.

Habrá entonces: un $T_{mejor}(n)$, un $T_{medio}(n)$ y un $T_{peor}(n)$

Cuál elegir entre: $T_{mejor}(n)$, $T_{medio}(n)$ o $T_{peor}(n)$?

Se debe definir T(n) como el ***peor*** tiempo de ejecución, esto es, el máximo sobre todas las posibles muestras de tamaño n.

Ej1. Algoritmo para ***encontrar el mayor elemento*** de un arreglo A[1..n], n≥1.

Algoritmo: Máximo
Entrada: A[1..n], : vector de números enteros,
n:entero
Salida: max: entero

P1. Leer(A,n)
P2. max ← A_1
P3. Para i desde 2 hasta n hacer
Si max < A_i entonces
max ← A_i
P4. Escribir (max)
P5. Fin

Costo uniforme*: hace siempre (n-1) comparaciones*

$T_{mejor}(n) \equiv T_{peor}(n)$

Ej2. Algoritmo de Búsqueda Secuencial para determinar la pertenencia de ***un elemento x*** a un arreglo A[1..n], n≥1.

Algoritmo: Búsqueda Secuencial
Entrada: **A**[1..n] : vector de números enteros,
n:entero ; **x**:entero
Salida: encontrado: bool

P1. Leer(A,n,x)
P2. encontrado ← falso
P3. i ← 1
P4. Mientras ($i \leq n$ AND $A_i \neq x$) hacer
i ← i+1
P5. encontrado ← ($i \leq n$)
P6. Escribir (encontrado)
P7. Fin

Costo no uniforme: mejor caso: 1 comparación ,
peor caso: n comparaciones.

$T_{mejor}(n) << T_{peor}(n)$

Eficiencia de algoritmos

Dos enfoques para el análisis de eficiencia de los algoritmos:

- El enfoque ***experimental*** (a posteriori) consiste en programar todos los algoritmos y probarlos con diferentes instancias con la ayuda de la computadora. Luego medir los tiempos de ejecución.
- El enfoque ***teórico*** (a priori) consiste en determinar matemáticamente la cantidad de recursos (tiempo de ejecución, espacio en memoria, etc.) necesarios para cada algoritmo en función del tamaño de la instancia considerada.

Eficiencia de algoritmos

Las ventajas del análisis teórico son:

- No depende ni de la computadora usada ni del lenguaje de programación, ni siquiera de la habilidad del programador.
- Ahorra el tiempo de programar algoritmos ineficientes, así como el tiempo el tiempo de máquina necesario para probarlos.
- Permite además estudiar la eficiencia de un algoritmo cuando se estudia instancias de cualquier tamaño.

Eficiencia de algoritmos

- El análisis requerido para estimar los recursos que usa un algoritmo es un problema teórico, y por ello se necesita un marco formal para su tratamiento.

- Se va a presentar una notación asintótica que permite comparar el orden de crecimiento de T(n), donde n es el tamaño de la entrada del algoritmo.

- Por el ***principio de invarianza*** se puede pensar **T(n) como el número de instrucciones en una computadora ideal.**

Notación O grande

Sea R* reales no negativos

Sea f: N → R* una función arbitraria

Se define O grande de f(n) como el conjunto de todas las aplicaciones:

$$\mathbf{O(f(n)) = \{\ t : N \rightarrow R^* \ / \ (\exists\ c \in R^+)\ (\exists\ n_0 \in N)\ (\forall\ n \geq n_0\ )\ [\ t(n) \leq c\ .\ f(n)]\ \}}$$

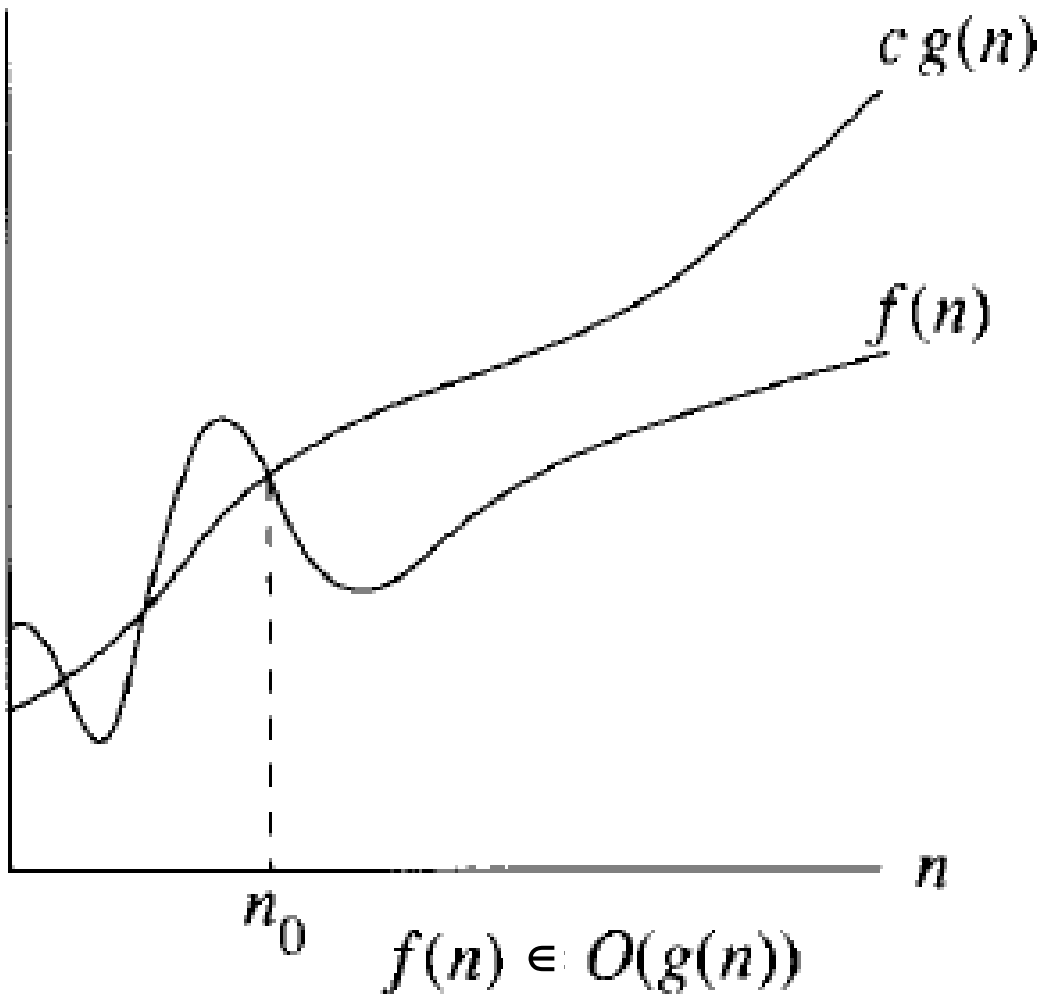
Si una función g ∈ a O(f(n)) se dice que:

g es O grande de f(n)

o que: *g es de orden f(n)*.

Notación O grande

En un gráfico:



Notación O grande

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n)) \Rightarrow O(f(n)) \subset O(g(n))$
- Si $f(n) \in O(g(n)) \Rightarrow c.f(n) \in O(g(n)) \quad \forall\, c \in R+$
- Si $f(n) \in O(g(n))$ y $h(n) \in O(g(n)) \Rightarrow f(n) + h(n) \in O(g(n))$
- $f(n) \in O(g(n))$ y $g(n) \in O(f(n)) \Leftrightarrow O(f(n)) = O(g(n))$
- $f(n) \in O(g(n))$ y $f(n) \in O(h(n)) \Rightarrow f(n) \in O(\min(g(n),h(n)))$

- **Transitiva:**
  $f(n) \in O(g(n))$ y $g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$

- **Regla de la suma**:
  $f1(n) \in O(g(n))$ y $f2(n) \in O(h(n)) \Rightarrow f1(n)+f2(n) \in O(\max(g(n),h(n)))$

- **Regla del producto**:
  $f1(n) \in O(g(n))$ y $f2(n) \in O(h(n)) \Rightarrow f1(n).f2(n) \in O(g(n).h(n))$

Notación O grande
Ejemplos:

Verdadero o Falso

- $3n+2 \in O(n)$?
- $3n \in O(1)$?
- $10n+100 \in O(n)$?
- $(n+1)^2 \in O(n^2)$?
- $\log_4 n \in O(\log_2 n)$?
- $\log_2 n \in O(\log_4 n)$?
- $2^n \in O(3^n)$?

$3n+2 \in O(n)$?

$3n+2 \leq c.n$ para $n \geq n_0$

Dividir por n>0 ambos miembros de la desigualdad:

$(3n+2)/n \leq c.n/n$ para $n \geq n_0$

$3+2/n \leq c$ para $n \geq n_0$

Vale para: $c=5$, $n_0=1$ o para: $c=4$, $n_0=2$ o…

Etc.

➔ $3n+2 \in O(n)$

$3n \in O(1)$?

$3n \leq c.1$ para $n \geq n_0$

$3n \leq c$ para $n \geq n_0$

3n crece con n $\Rightarrow$ no se puede acotar para ninguna constante c cuando n crece.

➔ $3n \notin O(1)$

$10n+100 \in O(n)$?

ya que:

$10n+100 \leq 20n$ para $n \geq 10$

$c=20$, $n_0=10$

➔ $10n+100 \in O(n)$

$(n+1)^2 \in O(n^2)$?

$(n+1)^2 \leq c.n^2$ para $n \geq n_0$

$n^2 + 2.n + 1 \leq c.n^2$ para $n \geq n_0$

Dividir por $n^2 > 0$ ambos miembros de la desigualdad:

$1 + 2/n + 1/n^2 \leq c$ para $n \geq n_0$

De modo que: si $c=4$, $n_0=1$

$(n+1)^2 \leq 4n^2$ para $n \geq 1$

➔ $(n+1)^2 \in O(n^2)$

$\log_4 n \in O(\log_2 n)$?

Recordar que: **$\log_a n = \log_b n / \log_b a$**

Entonces ya que:

$\log_4 n = \log_2 n / \log_2 4$ para $n \geq n_0$

$\log_4 n = \log_2 n / 2$ para $n \geq n_0$

Si: $c=0.5$, $n_0=1$

$\log_4 n = 0.5 \log_2 n$ para $n \geq 1$

➔ $\log_4 n \in O(\log_2 n)$

$\log_2 n \in O(\log_4 n)$?

ya que:

$\log_2 n = 2 . \log_4 n$ para $n \geq 1$

$c=2$, $n_0=1$

➔ $\log_2 n \in O(\log_4 n)$

$2^n \in O(3^n)$?

$2^n \leq c.3^n$ para $n \geq n_0$

$2^n / 3^n \leq c.3^n / 3^n$ para $n \geq n_0$

$(2/3)^n \leq c$ para $n \geq n_0$

para: $c=2/3$, $n_0=1$

$2^n \leq 2/3 \, . \, 3^n$ para $n \geq 1$

➔ $2^n \in O(3^n)$

Notación O grande
Ejercicios:

Verdadero o Falso

- $4n+1 \in O(n^2)$?
- $n - 1 \in O(1)$?
- $\log_2 n \in O(n)$?
- $2n+ \log_4 n \in O(n^2)$?
- $\sqrt{n} \in O(n)$?
- $3^n \in O(2^n)$?

Algoritmo

Sea T(n) el tiempo de ejecución de un algoritmo para una entrada de tamaño n.

Se dice que ***T(n) es de orden f(n)*** o que ***T(n) es O(f(n))*** si existen dos constantes: c(real) y n_0(natural) tales que:

$$T(n) \leq c.\ f(n) \text{ cuando } n \geq n_0$$

Cuando se dice que T(n) es O(f(n)) se está garantizando que la función f(n) es una cota superior para el crecimiento de T(n).

Notación O grande

Funciones de complejidad notables:

O(1) Complejidad constante

O(log n) Complejidad logarítmica

O(n) Complejidad lineal

O(n log n)

$O(n^2)$ Complejidad Cuadrática

$O(n^3)$ Complejidad Cubica

$O(n^a)$ Complejidad Polinomial

$O(2^n)$ Complejidad Exponencial

$O(c^n)$ Complejidad Exponencial, $c \geq 2$

Notación O grande

Se verifica que:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(c^n) \subset O(n!)$$

RAM
(RANDOM ACCESS MACHINE)

Cómo medir el tiempo de ejecución en función del tamaño de la entrada de los datos del problema?

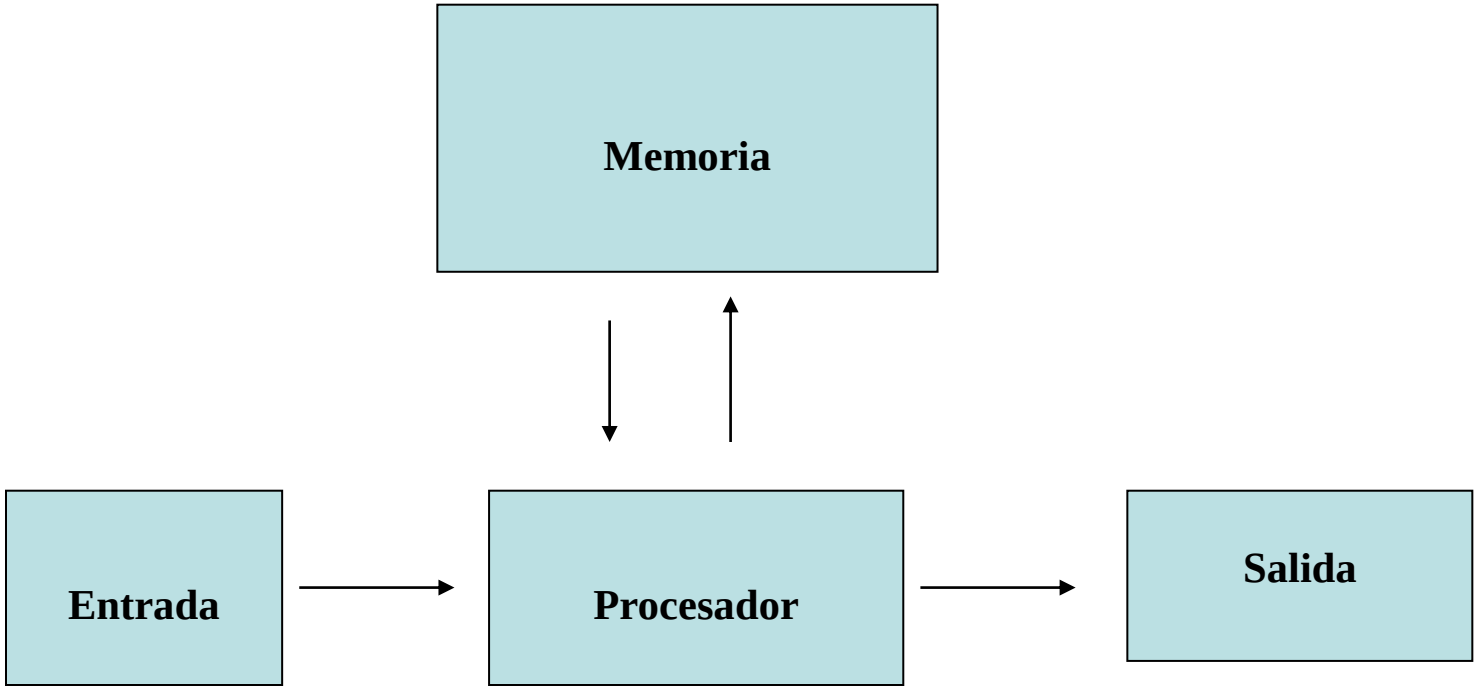
- RAM: modelo que se usa para evaluar la complejidad de un algoritmo.

- *Un modelo de máquina de acceso directo RAM es una computadora de un acumulador cuyas instrucciones no pueden modificarse a sí mismas.* Consta de:

  unidad de memoria
  unidad de entrada
  procesador
  unidad de salida
  conjunto de instrucciones

- Un ***programa*** RAM es una *secuencia finita de estas instrucciones*

RAM



RAM

En este modelo se supone que:

- La ***entrada*** es una secuencia representada por una cinta de cuadrados en cada uno de los cuales está almacenado un número entero.

- Cuando un símbolo se lee de la cinta, la cabeza ***avanza*** al próximo cuadrado.

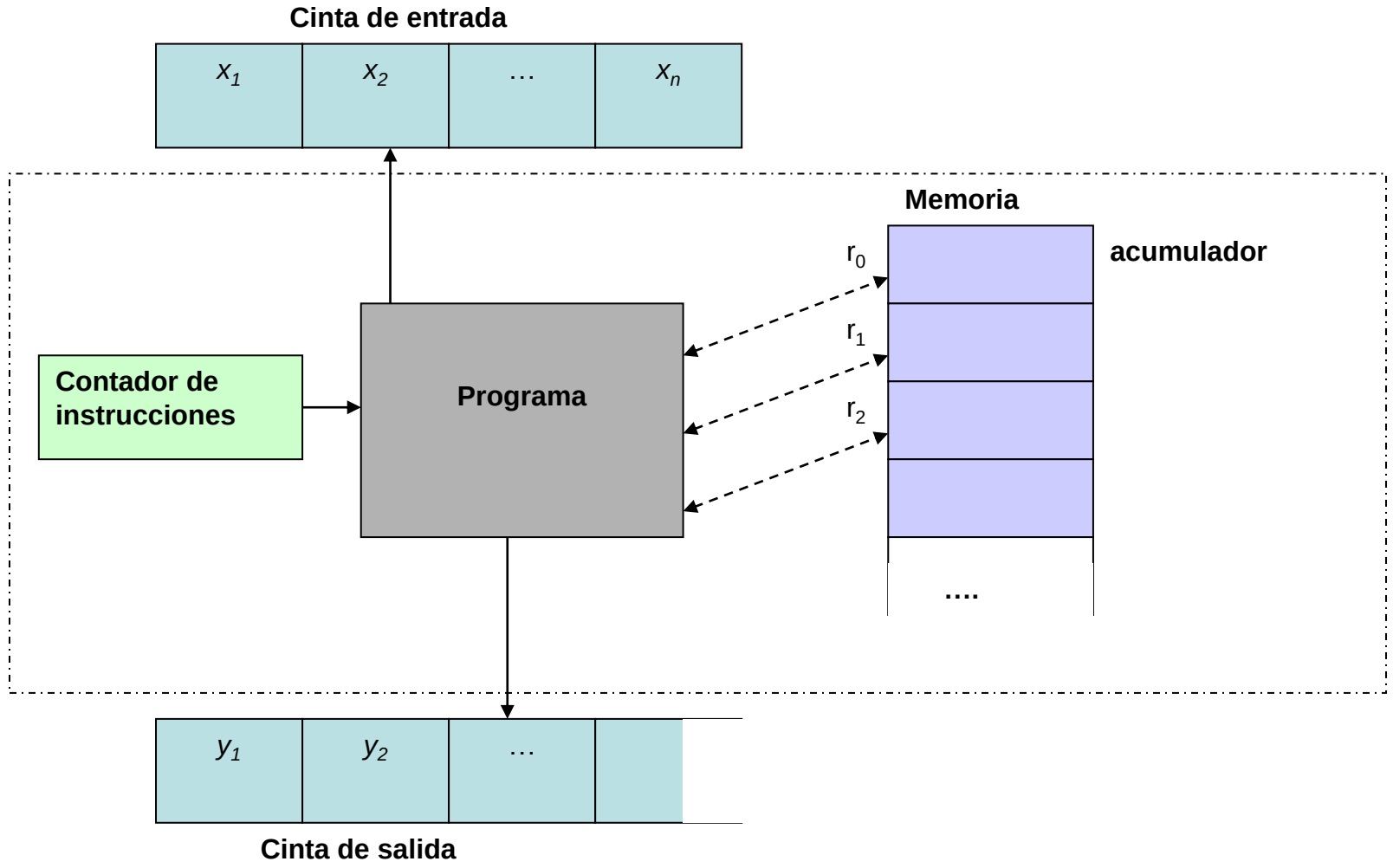
- La ***salida*** es una cinta de cuadrados, que está inicialmente en blanco, en cada uno de los cuales se puede escribir un entero.

- Cuando un símbolo se escribe en la cinta, la cabeza ***avanza*** al próximo cuadrado. Cuando ya se escribió un símbolo, no se puede cambiar.

RAM

- La ***memoria*** es una sucesión (cantidad ilimitada) de registros *r0*,*r1*,.., cada uno de los cuales puede almacenar un entero de tamaño arbitrario.
- Los cálculos se hacen en el primero de ellos, *r0* , llamado ***acumulador.***
- El ***programa*** es una secuencia de instrucciones.
- Cada instrucción tiene un ***código de operación*** y una ***dirección.***
- El programa no se almacena en la memoria y no se modifica a si mismo.
- El ***conjunto de instrucciones*** disponibles es similar al de las *computadoras reales* (operaciones aritméticas, de entrada/salida, de bifurcación*…)*

RAM


Cinta de entrada
x_1
x_2
…
x_n
Memoria
acumulador
r_0
r_1
r_2
....
Contador de instrucciones
Programa
y_1
y_2
…
Cinta de salida

Notación O grande
Reglas Generales

Cero pasos, se desprecia

Todo lo que sea no ejecutable: comentarios, declaraciones, tipificaciones, etc.

Operaciones elementales, se consideran O(1)

- leer un valor
- escribir un valor o expresión simple
- asignar, excepto de estructuras
- evaluar un expresión que no tenga invocación a una función
- evaluar la condición de una selección o iteración

Notación O grande
Reglas Generales

Secuencia

El tiempo de ejecución de una secuencia de instrucciones esta dado por la regla de la suma. Esto es el mayor tiempo de ejecución de cada instrucción de la secuencia.

Selección

Hay que sumar el tiempo de evaluar la condición de selección (en general es O(1)), mas el mas grande de los tiempos que se necesitan para cada una de las distintas alternativas.

Notación O grande
Reglas Generales

Iteración

Hay que sumar el tiempo de todas las instrucciones que abarca la iteración (el cuerpo), mas el tiempo de evaluar la condición de terminación y todo multiplicarlo por el numero de iteraciones máximo que se efectúen.

La condición es en general O(1).

Notación O grande
Reglas Generales

Funciones no recursivas

El tiempo de cada función se debe evaluar por separado, comenzando por las funciones que no invoquen a otras. Usando esos tiempos se evalúan las funciones que las invocaron y así hasta llegar al programa principal.

Funciones recursivas

Se asocia una función de tiempo desconocida T(n). Se plantea la recurrencia de T(n) donde n mide el tamaño de la entrada. Luego se desarrolla la recurrencia, esto es una ecuación para T(n) en términos de T(k) para distintos valores de k.

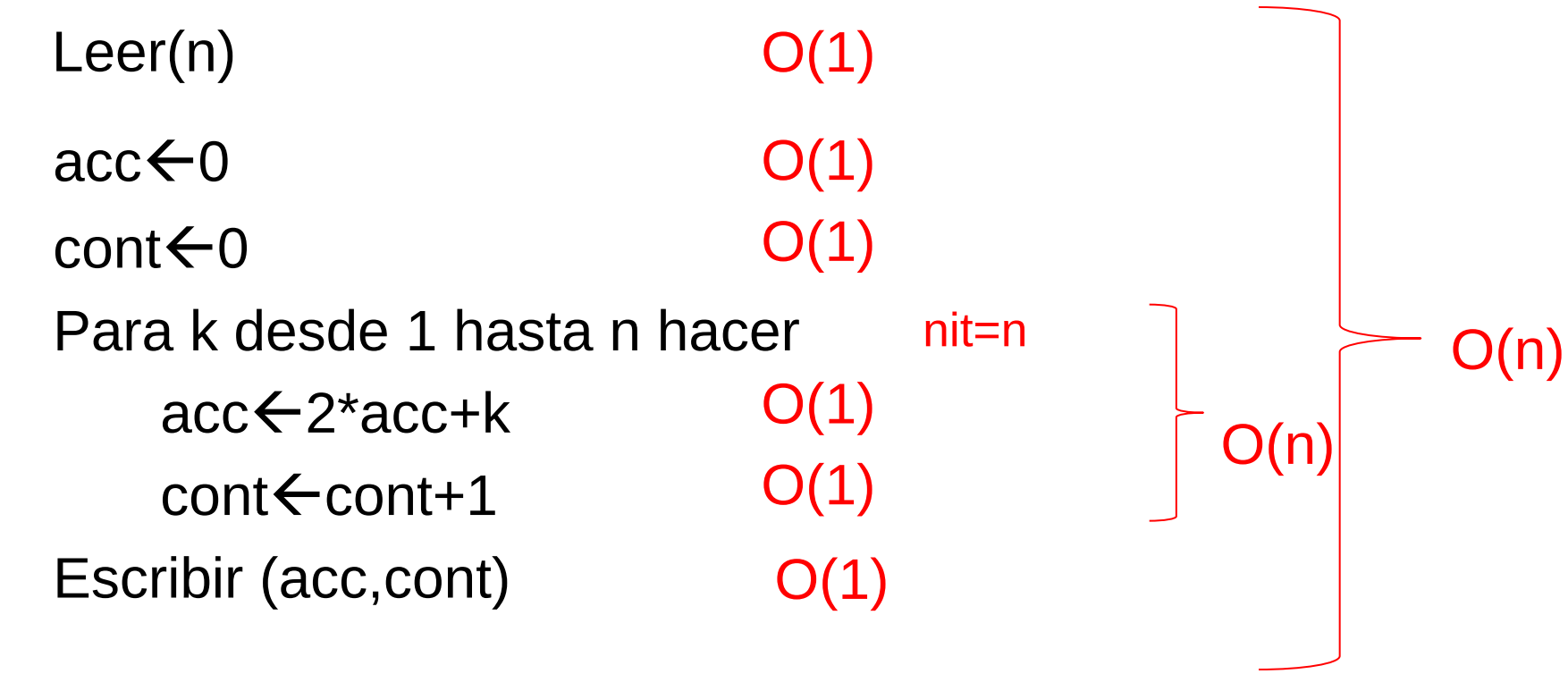
Ejemplo

```
…
acc←0                                          O(1)
cont←0                                         O(1)
Para k desde 1 hasta 1000 hacer                nit=1000
      acc←2*acc+k                              O(1)
      cont←cont+1                              O(1)
Escribir (acc,cont)                            O(1)
…
```

O(1) (bloque Para)

O(1) (todo)

Ejemplo


Leer(n) O(1)
acc←0 O(1)
cont←0 O(1)
Para k desde 1 hasta n hacer nit=n
acc←2*acc+k O(1)
cont←cont+1 O(1)
Escribir (acc,cont) O(1)
O(n)
O(n)
…

Ejemplo

…

Leer (n) O(1)

k←1 O(1)

Para i desde n hasta n+999 hacer nit=1000 O(1)

    k←k*i O(1)

Escribir (k) O(1)

…

O(1)

Ejemplo

```
…
Leer (n)                                  O(1)
suma←10                                   O(1)
Para i desde 1 hasta n hacer              nit = n  } O(n)        } O(n)
      suma←suma+i*i+i                     O(1)
Escribir (suma)                           O(1)
…
```

Ejemplo

…

Leer (n) O(1)

k←0 O(1)

Para i desde **1** hasta **n** hacer nit = n

    Para j desde **1** hasta **n** hacer nit = n

        k←k+1 O(1)

Escribir (k) O(1)

…

O(n) O(n²) O(n²)

Ejemplo

…

Leer (n) O(1)

k←0 O(1)

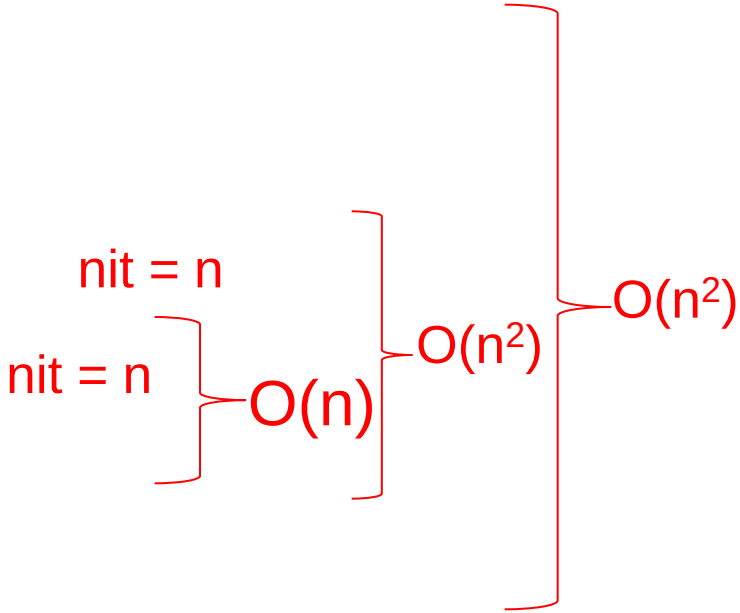
Para i desde **1** hasta **n** hacer nit = n

    Para j desde **1** hasta **i** hacer nit = n

        k←k+1 O(1)

Escribir (k) O(1)

…



Por qué en la iteración de índice j se considera nit = n ????

Ejemplo

Para i desde 1 hasta n hacer

Para j desde 1 hasta i hacer

Por qué en la iteración de índice j se considera nit = n ?

Dos posibilidades:

1) Se supone siempre el peor caso para el valor de i, que es i=n entonces: nit=n en el lazo j
2) Se puede calcular el valor exacto de iteraciones de los 2 lazos anidados:

$1+2+3+\ldots+n = \Sigma^{n}_{i=1}\ i = (1+n).\ n/2\ = n/2 + n^2/2 \in O(n^2)$

Ejemplo

…

Leer (n,B,C) $O(n^2)$

Para **i** desde **1** hasta **n** hacer — nit = n

  Para **j** desde **1** hasta **n** hacer — nit = n

    A (i,j) ← 0 $O(1)$

    Para **k** desde **1** hasta **n** hacer — nit = n

      A (i,j) ← A (i,j) + B (i,k) * C (k,j) $O(1)$

$O(n)$, $O(n^2)$, $O(n^3)$, $O(n^3)$

Escribir (A) $O(1)$

…

Ejemplo

…

Leer (n) O(1)

suma←0 O(1)

Para **i** desde **1** hasta n hacer nit = n

  Para **j** desde **1** hasta **n^2** hacer nit = n^2

    suma←suma + j O(1)

Escribir (suma) O(1)

…

$O(n^2)$ $O(n^3)$ $O(n^3)$

Ejemplo

…

Leer (n) O(1)

cuad←1 O(1)

Mientras cuad < n hacer nit = ?

    cuad←cuad*2 O(1)

Escribir (cuad) O(1)

…

Cómo se calcula el número de iteraciones de la iteración condicional ?

Ejemplo

```
cuad←1
Mientras cuad < n hacer
        cuad←cuad*2
```

Cómo se calcula el número de iteraciones de la iteración condicional Mientras …?

Analizar los valores que va tomando cuad:

cuad=1,2,4,8,… cuad=2^0 , 2^1 , 2^2 , 2^3 , … $\Rightarrow$ cuad=2^x ,

Si itera x veces mientras (cuad<n), entonces la iteracion finaliza cuando (cuad $\geq$ n) , pero cuad=2^x, entonces $2^x \geq n$,

de alli se deduce que: $x \geq \log_2 n$

Ejemplo

…

Leer (n) O(1)

cuad←1 O(1)

Mientras cuad < n hacer nit = $\log_2 n+1$ $O(\log_2 n)$

    cuad←cuad*2 O(1)

Escribir (cuad) O(1)

$O(\log_2 n)$

…

Ejemplo

```
...
Leer (n)                                    O(1)
r←0                                         O(1)
Si n =100 Entonces
        r ← r+n                             O(1)
Sino Si n=200  Entonces
        r ← r+2*n
Sino Si n=300 Entonces
        Para i desde 1 a n  hacer           nit = n ?
                r ← r+i                     O(1)
Sino
        r ← -999                            O(1)
Escribir (r)                                O(1)
...
```

O(1) u O(n) ?

O(1) u O(n) ?

Complejidad Práctica

La complejidad de un Programa es generalmente función de las características de la instancia.

La funcion de complejidad se puede usar para comparar dos programas P y Q que hagan la misma tarea.

Se presentan tablas con los valores aproximados para distintas funciones de crecimiento.

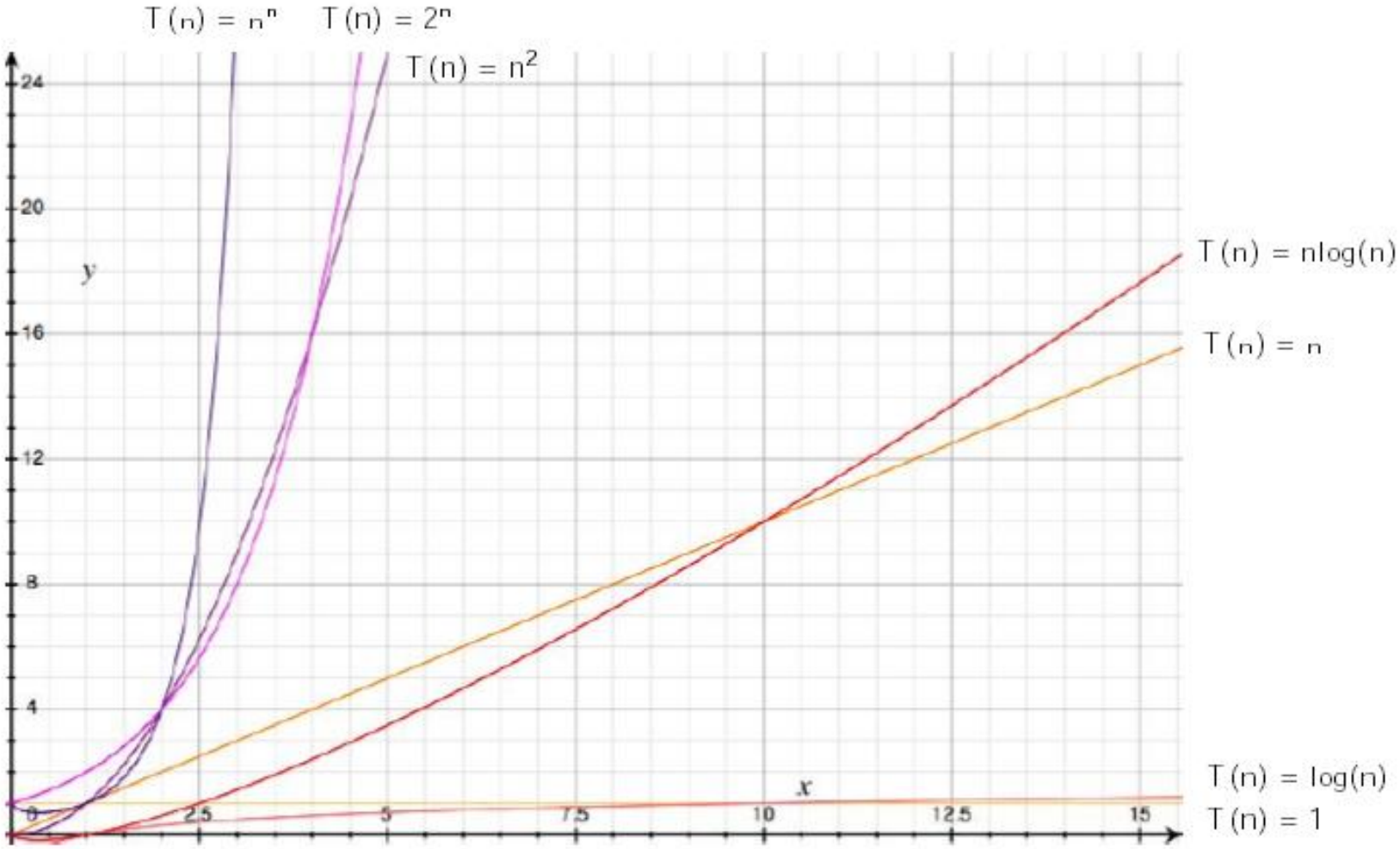
Crecimiento de distintas funciones de costo

n	log n	n log n	n^2	n^3	2^n
1	0	0	1	1	2
5	0.7	3	25	125	32
10	1.0	10	100	1000	1024
20	1.3	26	400	8000	1048576
50	1.7	85	2500	25000	1125900000×10^6
100	2.0	200	10000	1000000	$1267651000 \times 10^{21}$
200	2.3	460	40000	8000000	-
500	2.7	1349	250000	125000000	-
1000	3.0	3000	1000000	1000000000	-

Crecimiento de distintas funciones de costo

n	log n	n log n	n^2	n^3	2^n
1	0	0	1	1	2
5	0.7	3	25	125	32
10	1.0	10	100	1000	1024
20	1.3	26	400	8000	1048576
50	1.7	85	2500	25000	1125900000×10^6
100	2.0	200	10000	1000000	$1267651000 \times 10^{21}$
200	2.3	460	40000	8000000	-
500	2.7	1349	250000	125000000	-
1000	3.0	3000	1000000	1000000000	-

Gráfica de los valores de las funciones T(n)



Tiempo T(n)
(si se ejecutan en promedio 10^6 instrucciones/seg)

n \ T(n)	n	n LOG_2n	n^2	n^3	1.5^n	2^n	n!
10	<1s	<1s	<1s	<1s	<1s	<1s	10^{25} a
30	<1s	<1s	<1s	<1s	<1s	18 min	-
50	<1s	<1s	<1s	<1s	11 min	36 a	-
100	<1s	<1s	<1s	1 s	12892 a	$3\ 10^7$ a	-
1.000	<1s	<1s	1s	17 min	-	-	-
10.000	<1s	<1s	2 min	12 d	-	-	-
100.000	<1s	2s	3 h	32 a	-	-	-
1.000.000	1s	20s	12 d	31710 a	-	-	-

s=segundo min=minuto h=hora d=día a=año

Tiempo T(n)
(si se ejecutan en promedio 10^9 instrucciones/seg)

n \ T(n)	n	n LOG$_2$n	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121.36 d	18.3 min
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 a	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 a	$4\ 10^{13}$ a
1.000	1. μs	9.96 μs	1 ms	1 s	16.67 min	$3.17\ 10^7$ a	$32\ 10^{283}$ a
10.000	10. μs	130.03 μs	100 ms	16.67 min	115.7 d	$3.17\ 10^{23}$ a	-
100.000	100 μs	1.66 ms	10 s	11.57 d	3171 a	$3.17\ 10^{33}$ a	-
1.000.000	1.ms	19.92 ms	16.67 min	31.71 a	$3.17\ 10^7$ a	$3.17\ 10^{43}$ a	-

μs= microsegundo = 10^{-6} segundo ms= milisegundo = 10^{-3} segundo

s=segundo min=minuto h=hora d=día a=año

Tamaño del problema más grande que se puede resolver en 1 hora

complejidad del algoritmo	**con una computadora actual**	**con una computadora 100 veces más rápida**	**con una computadora 1000 veces más rápida.**
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$312.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

Algoritmos eficientes

Un algoritmo es eficiente si existe un polinomio P(n) tal que el algoritmo puede resolver cualquier caso de tamaño de entrada *n* en un tiempo:

$$T(n) \in O(P(n))$$

Algoritmo eficiente: Algoritmo de complejidad polinomial

- Un algoritmo es eficiente si tiene un *tiempo polinómico* de ejecución.
- Si un algoritmo ***no*** es eficiente se lo denomina algoritmo de *tiempo exponencial*.

Algoritmos no eficientes

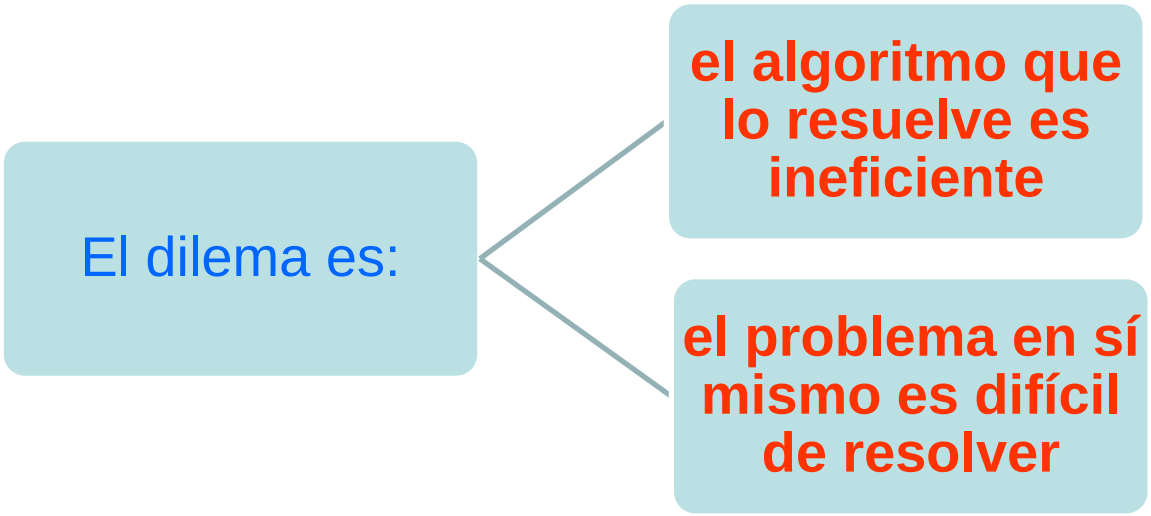
Los ejemplos de problemas más populares para los que *no se conocen algoritmos eficientes* son:

- viajante de comercio
- ciclo hamiltoniano
- satisfactibilidad de una fórmula booleana
- coloreado óptimo de grafos

Todos los algoritmos conocidos para estos problemas son *no eficientes*, de modo que son algoritmos inútiles cuando el tamaño de la entrada *n* crece mucho.

Problemas-Algoritmos

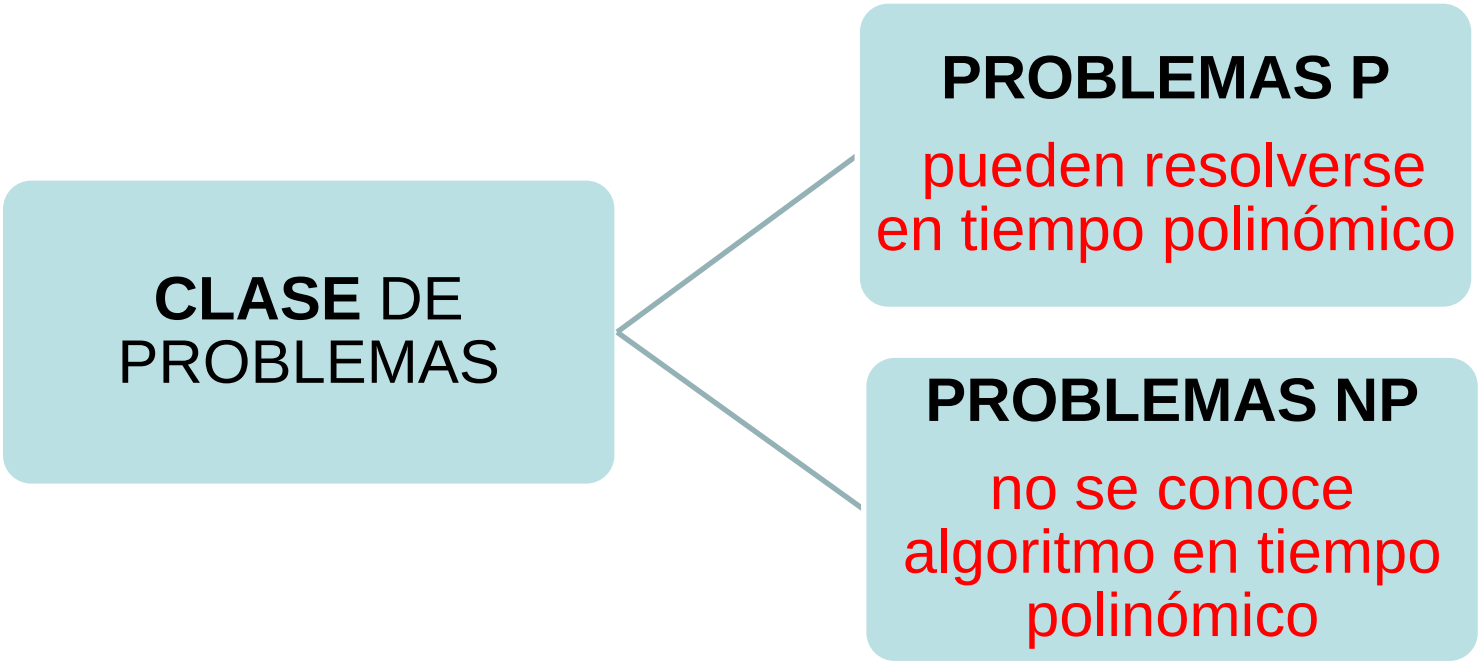
Entre los problemas que se resuelven con una computadora existen algunos para los cuales no se conocen ningún algoritmo eficiente que los resuelva, tampoco se ha demostrado que tienen una dificultad intrínseca.



- Podría ser que todavía no se ha encontrado un algoritmo eficiente para resolverlo…
- Podría ser que el problema sea intrínsecamente difícil, pero todavía no se lo ha podido demostrar…

Problemas

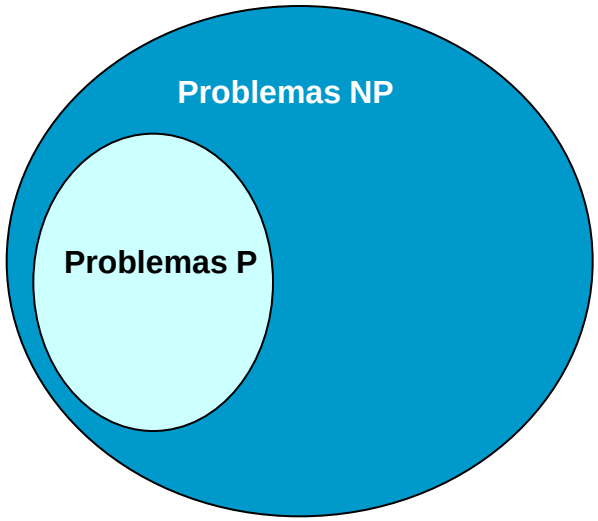
Los problemas se pueden dividir en 2 grandes grupos:



Problemas P y NP

- Es fácil de ver que*:*

$$P \subset NP$$



- **Problema abierto:**

$$\text{es } P \neq NP?$$

*Este es **El Problema** abierto*
más importante de teoría de la computación.

Problemas NP completos

- Fueron caracterizados en 1971 por Stephen Cook. Todos los *problemas NP completos son equivalentes desde el punto de vista computacional* . Esto significa que se puede pasar de un problema a otro problema de NP con por una transformación polinómica..

- Ejemplo del problemas de este tipo son: SAT, agente viajero, ciclo hamiltoniano en un grafo, coloreado de un mapa con 3 colores.