Programación 2

Introducción al Análisis de Algoritmos

Análisis de Algoritmos: Introducción

- Qué algoritmos elegir para resolver un problema?
  - Que sean fáciles de entender, codificar y depurar
  - Que usen eficientemente los recursos del sistema: que usen poca memoria y que se ejecuten con la mayor rapidez posible
- Ambos factores en general se contraponen…
- Nos concentraremos ahora en el segundo factor y en particular en el análisis del <u>*tiempo de ejecución*</u>

Tiempo de ejecución de un programa

Factores que intervienen:

- La calidad del código generado por el compilador
- La naturaleza y rapidez de las instrucciones de máq.
- Aspectos físicos del PC
- **Los datos de entrada al programa**
- **La complejidad de tiempo del algoritmo base (lógica algorítmica)**.

El tiempo de ejecución de un programa depende de la entrada y en general, del tamaño de la misma

T(n)

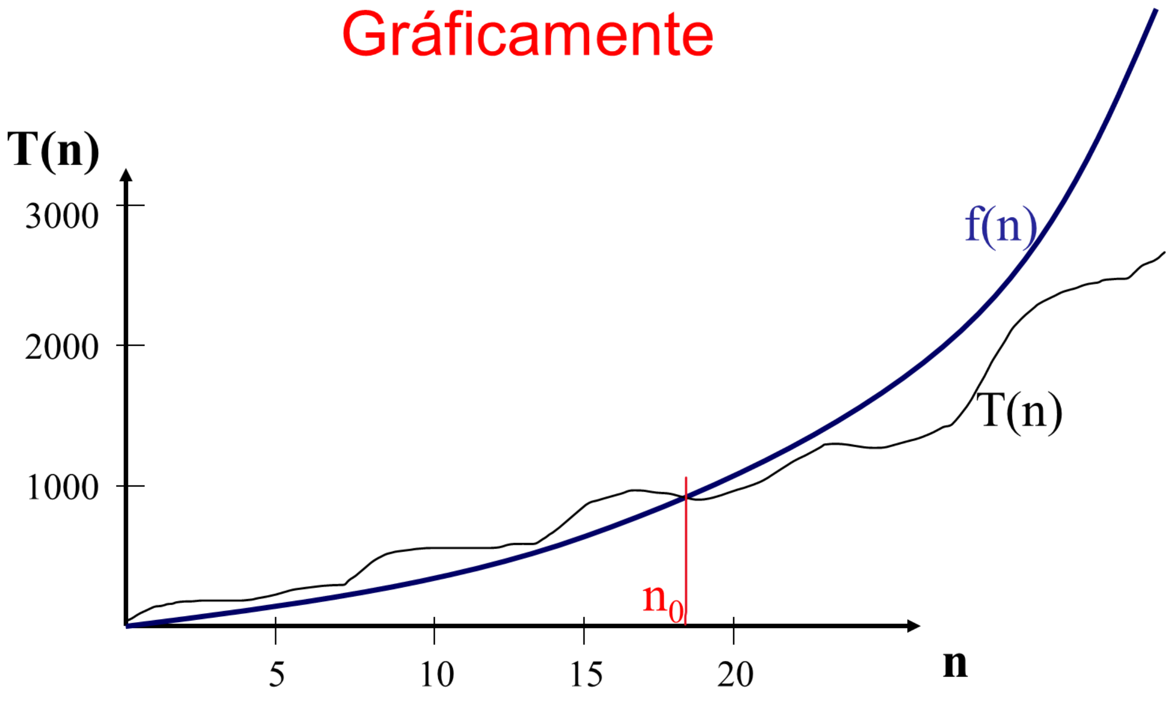
- **T(n)** = tiempo de ejecución de un programa con una entrada de tamaño **n**
  = número de instrucciones ejecutadas en un computador idealizado con una entrada de tamaño **n**
- Para el problema de ordenar una secuencia de elementos, n sería la cantidad de elementos

  Ejemplo: $T(n) = c.n^2$, donde c es una constante
- $T^{peor}(n)$ = tiempo de ejecución para el peor caso
  $T^{prom}(n)$ = tiempo de ejecución del caso promedio
  Nos centraremos en $T^{peor}(n)$ y lo llamaremos simplemente T(n).

Velocidad de crecimiento - O(n)

T(n) es **O(*f*(n))** "orden *f(n)*" si existen constantes positivas *c* y *n0* tales que T(n) $\leq$ *c.f*(n) cuando n$\geq$ *n0.*
f(n) es una **<u>cota superior</u>** para la <u>velocidad (taza) de crecimiento</u> de un programa con tiempo de ejecución T(n)

Gráficamente



Velocidad de crecimiento - O(f(n))

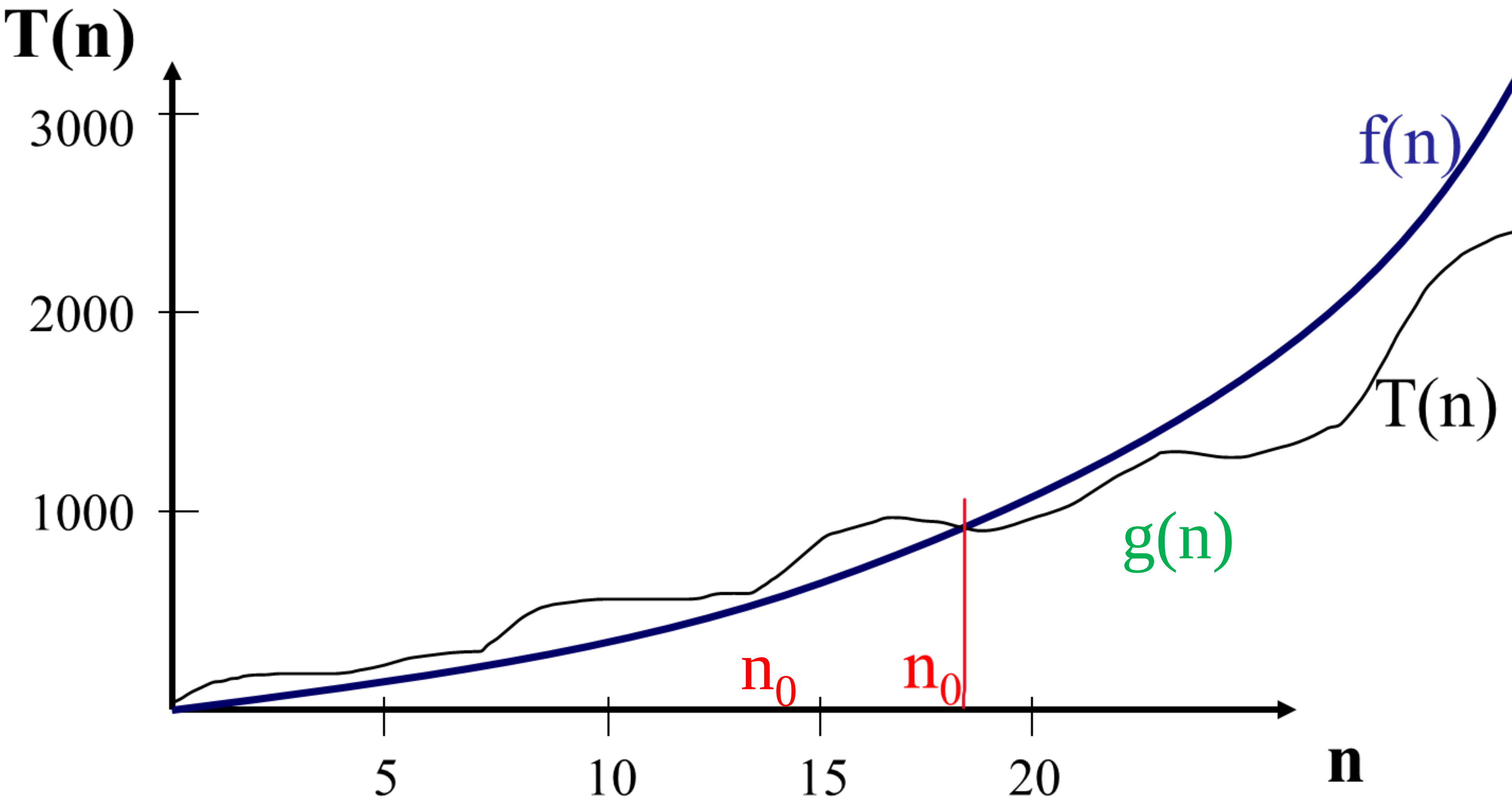
- T(n) es **O(*f*(n))** "orden *f(n)*" si existen constantes positivas *c* y *n0* tales que $T(n) \leq c.f(n)$ cuando $n \geq n0$. *f*(n) es una **cota superior** para la velocidad (taza) de crecimiento de un programa con tiempo de ejecución T(n)
- Ejemplo:
  - $T(n) = 3n^3 + 2n^2$ es $O(n^3)$
    - Sean *n0* = 0 y *c* = 5, $3n^3 + 2n^2 \leq 5\, n^3$, para $n \geq 0$
    - También T(n) es $O(n^4)$, pero sería una aseveración más débil que decir que es $O(n^3)$

Velocidad de crecimiento - $\Omega(g(n))$

- T(n) es **Ω(*g*(n))** si existen constantes positivas *c* y *n0* tales que $T(n) \geq c.g(n)$ cuando $n \geq n0$. *g*(n) es una **cota inferior** para la velocidad (taza) de crecimiento de un programa con tiempo de ejecución T(n)

- Ejemplo:
  - $T(n) = 3n^3 + 2n^2$ es $\Omega(n^3)$
    - Tomemos *n0* = 0 y *c* = 1, $3n^3 + 2n^2 \geq n^3$, para $n \geq 0$
    - También T(n) es $\Omega(n^2)$, pero sería una aseveración más débil que decir que es $\Omega(n^3)$

Gráficamente



$\Theta(h(n))$

- T(n) es $\boldsymbol{\Theta(h(n))} \Leftrightarrow$ T(n) es $O(h(n))$ y $\Omega(h(n))$

- <u>Ejemplo</u>:
  - $T(n) = 3n^3 + 2n^2$ es $\Theta(n^3)$

Mejor caso

También existe el **mejor caso**, que corresponde al costo mínimo, en tiempo de ejecución, de un algoritmo para datos de entrada de tamaño n.

Su aplicabilidad es menos relevante en la práctica, en general.

Por ejemplo, en un algoritmo iterativo típico de búsqueda de un elemento en un arreglo (no ordenado) de tamaño n (que itera desde la posición 0 a la n-1), el *peor caso* será de orden n (el elemento está al final o no está, por ejemplo), mientras que el *mejor caso* será de tiempo constante (es el primer elemento).

Evaluación de programas

- Un programa con tiempo de ejecución $O(n^2)$ es mejor que uno con $O(n^3)$ para resolver un mismo problema.
- Supongamos dos programas P1 y P2 con $T1(n) = 100n^2$ y $T2(n) = 5n^3$

¿Cuál programa es preferible?

- Si $n < 20$, P2 es más rápido que P1

  (para entradas "pequeñas" es mejor P2)
- Si $n > 20$, P1 es más rápido que P2

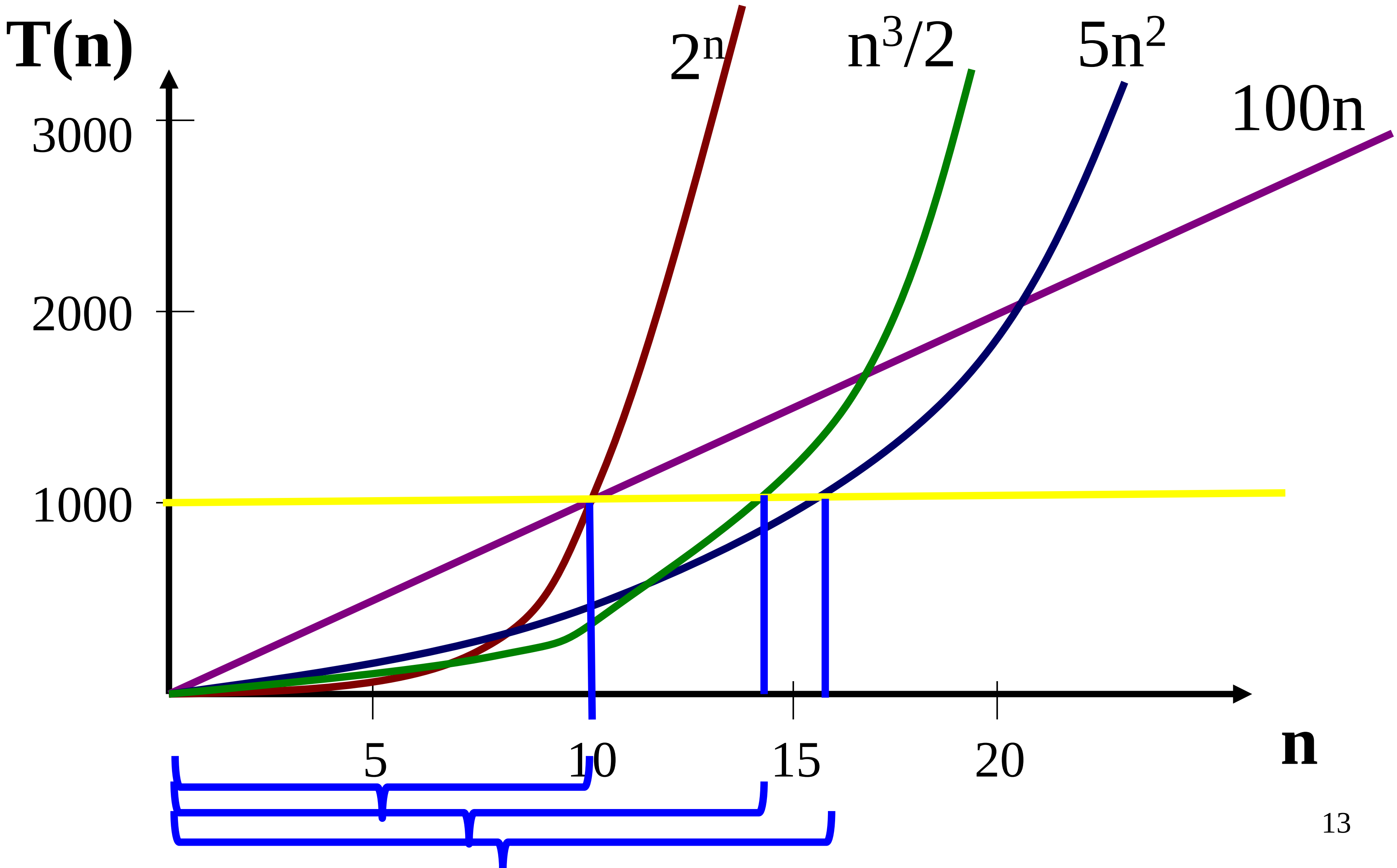
  (para entradas "grandes" es mejor P1)

¿Entonces? Lo relevante es el orden (O), cuando *n* crece...

Evaluación de programas (cont)

- La velocidad de crecimiento de un programa determina el tamaño de los problemas que se pueden resolver en un computador.
- Si bien las computadoras son cada vez más veloces, también aumentan los deseos de resolver problemas más grandes.
- Salvo que los programas tengan una velocidad de crecimiento baja, ej: O(n) u O(n.log(n)), un incremento en la rapidez del computador no influye significativamente en el tamaño de los problemas que pueden resolverse en una cantidad fija de tiempo.

Tiempos de ejecución de 4 programas



Efecto de multiplicar por 10 la velocidad de un computador

T(n)	Tamaño del max. problema para 10^3	Tamaño del max. problema para 10^4	Incremento en el tamaño del max. problema
$100n$	10	100	10
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Aunque la velocidad de un computador aumente 1000%, un algoritmo ineficiente no permitirá resolver problemas mucho más grandes.

La idea es entonces: <u>desarrollar algoritmos eficientes.</u>

Algunas velocidades de crecimiento típicas

Para *n* grande ↓

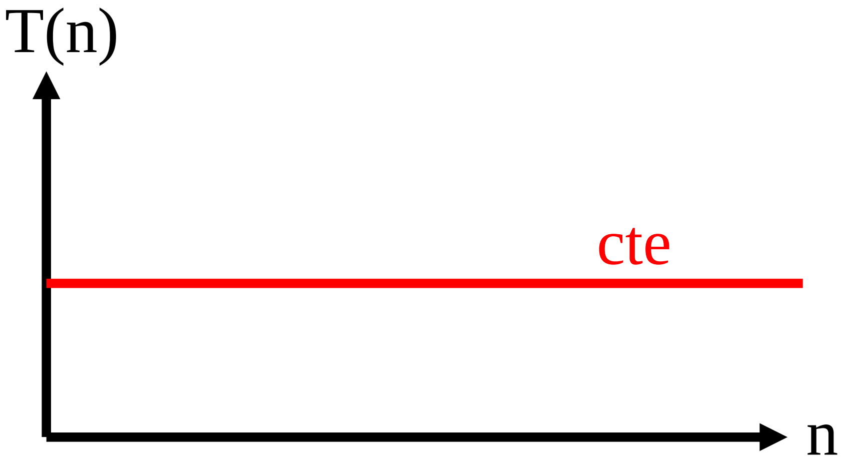
Función	Nombre
c	constante
log(n)	logarítmica
$\log^2(n)$	log-cuadrado
n	lineal
n.log(n)	
n^2	cuadrática
n^3	cúbica
2^n	exponencial

Tiempo constante – O(1)

T(n) es ***O(f(n))*** *"orden f(n)" si existen constantes positivas c y n0 tales que* $T(n) \leq c.f(n)$ *cuando* $n \geq n0$.

Si T(n) = cte, T(n) es O(1), ya que existe *c* (c=cte) y *n0* (n0=0), tales que:

$$cte \leq cte.1 \text{ cuando } n \geq 0.$$



Cálculo del tiempo de ejecución

- **Regla de la Suma**:

Si T1(n) es O(f1(n)) y T2(n) es O(f2(n)) entonces **T1(n)+T2(n)** es **O(max (f1(n), f2(n)))**

$\Rightarrow$ Puede usarse para calcular el tiempo de ejecución de una secuencia de pasos de programa.

Ejemplo: supongamos 3 procesos secuenciales con tiempos de ejecución $O(n^2)$, $O(n^3)$ y O(n.log(n)). El tiempo de ejecución de la composición es $O(n^3)$.

¿Cómo se demuestra la regla de la Suma?

Teorema de la Regla de la Suma

Si T1(n) es O(f1(n)) y T2(n) es O(f2(n)) entonces **T1(n)+T2(n)** es **O(max (f1(n), f2(n)))**

- H1: T1(n) es O(f1(n)) $\leftrightarrow$ $\exists$ *c1*,*n1:* T1(n) $\leq$ *c1*.*f1*(n), para n$\geq$*n1*
- H2: T2(n) es O(f2(n)) $\leftrightarrow$ $\exists$ *c2*,*n2:* T2(n) $\leq$ *c2*.*f2*(n), para n$\geq$*n2*
- Tesis: T1(n)+T2(n) es O(max(f1(n),f2(n))) $\leftrightarrow$ $\exists$ *c*, *n0:*

  T1(n)+T2(n) $\leq$ *c*. max(f1(n),f2(n)), para n$\geq$*n0.*

<u>Demostración (bosquejo):</u>

T1(n)+T2(n) $\leq$ c1.f1(n) + c2.f2(n), por H1 y H2 para n$\geq$n1, n$\geq$n2

$\leq \underbrace{2.\ \max(c1,c2)}_{c}.$ max(f1(n),f2(n)). Notar que: x+y $\leq$ 2. max(x,y)

n0 = max(n1,n2)

Cálculo del tiempo de ejecución (cont)

Si para todo $n \geq n0$ (n0 cte) $f1(n) \geq f2(n)$ entonces $O(f1(n)+f2(n))$ es lo mismo que $O(f1(n))$.

Ejemplo: $O(n^2+n)$ es lo mismo que $O(n^2)$

- **Regla del Producto**:

Si $T1(n)$ es $O(f1(n))$ y $T2(n)$ es $O(f2(n))$ entonces **$T1(n).T2(n)$** es **$O(f1(n).f2(n))$**

$O(c.f(n))$ es lo mismo que $O(f(n))$ (c es una cte postiva)

Ejemplo: $O(n^2/2)$ es lo mismo que $O(n^2)$

Cálculo de T(n) - Algunas reglas

- Para una **asignación** (lectura/escritura e instrucciones básicas) es en general O(1) (tiempo constante)
- Para una **secuencia** de pasos se determina por la regla de la suma (dentro de un factor cte, el "máximo")
- Para un " if (***Cond***) ***Sent*** " es el tiempo para ***Sent*** más el tiempo para evaluar ***Cond*** (este último en general O(1) para condiciones simples)
- Para un " if (***Cond***) $\boldsymbol{Sent_1}$ else $\boldsymbol{Sent_2}$ " es el tiempo para evaluar ***Cond*** más el máximo entre los tiempos para $\boldsymbol{Sent_1}$ y $\boldsymbol{Sent_2}$

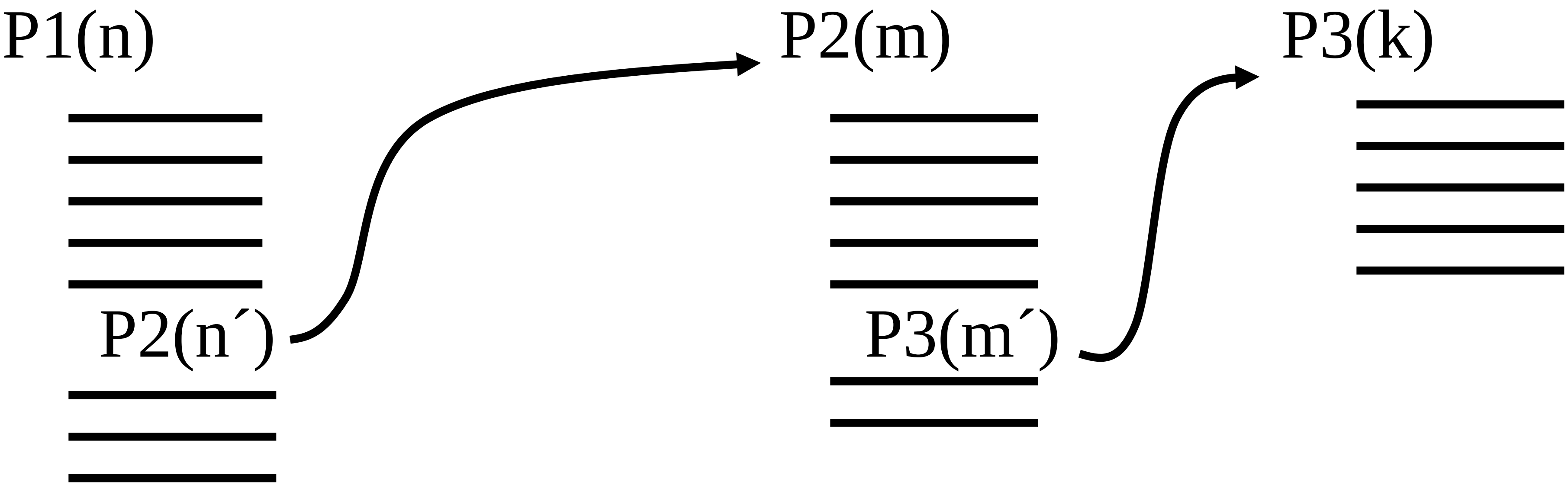
Cálculo de T(n) - Algunas reglas (cont)

- Para un **ciclo** es la suma, sobre todas las iteraciones del ciclo ($\sum$), del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación (este último suele ser O(1)).

  $\Rightarrow$ A menudo este tiempo es, despreciando factores constantes, el producto del número de iteraciones del ciclo y el mayor tiempo posible para una ejecución del cuerpo.

Cálculo de T(n) - Algunas reglas (cont)

- **Llamada a procedimientos** (funciones) no recursivos



Se calcula el tiempo de ejecución del procedimiento que no depende de otro: P3.
Luego el de P2 y entonces el de P1.

Cálculo de T(n) - Ejemplos (cont)

Considere los siguientes fragmentos de programas:

- ```
  for (i=0; i<n; i++) printf("%d\n", A[i][i]);
  ```

- ```
  for (i=0; i<n; i++)
       for (j=0; j<n; j++) if (i== j) printf("%d\n", A[i][j]);
  ```

¿Qué hacen?

¿Cuál es más eficiente y por qué?

Cálculo de T(n) - Ejemplos

Escribir algoritmos en pseudocódigo para los siguientes problemas y determinar el orden de tiempo de ejecución en el peor caso (<u>O(n)</u>):

- Calcular el máximo entre dos números.
- Calcular la sumatoria de los elementos de un arreglo de tamaño n de números enteros.
- Imprimir los elementos de una matriz cuadrada ($n \times n$) de números enteros.
- Calcular la sumatoria de los elementos de una matriz de tamaño $n \times m$ de números naturales.

Cálculo de T(n) - Ejemplos (cont)

Considere el siguiente fragmento de programa:

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

¿Qué hace?, ¿Cuál es su tiempo de ejecución?

Despreciando algunos factores constantes, podemos calcular T(n) y luego O(n) para: T(n) = $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} cte$

Nota: Para este problema existen algoritmos O(n.log(n))

Cálculo de T(n) - Ejercicios

Considere el siguiente procedimiento:

```
void P (int n)
{     int i,j;
      if (n%2==0)
            for (i=n-1; i>-1; i--) printf("%d\n", A[i][i]);
      else  for (i=0; i<n; i++)
                  for (j=n-1; j>-1; j--)
                        if (i==j) printf("%d\n", A[i][j]);
}
```

Donde, par es una función booleana de O(1) y A es una matriz definida como una variable global al procedimiento de tamaño $n \times n$.

a) ¿Qué hace el procedimiento P?

b) Calcule el Orden (O) de tiempo de ejecución del procedimiento P.

Cálculo de T(n) - Ejercicios

Considere la siguiente función:

```
int F (int n, int * A)
{     int m=A[0];
      for (int i=n-1; 0<i; i--)
            if (A[i]<m) m=A[i];
      return m;
};
```

Donde, A es una arreglo de enteros de tamaño n.

a) ¿Qué calcula la función F?

b) Calcule el Orden (O) de tiempo de ejecución de la función F.

Cálculo de T(n) - Ejercicios

Considere el siguiente procedimiento:

```
void P (int k, int n, int * A)
{     int i,j; bool esta=false;
      for (i=0; i<n && !esta; i++)
            if (A[i]==k)
                  { esta=true;
                    for (j=0; j<n; j++) printf(…);
                  }
}
```

Donde, A es una arreglo de enteros de tamaño n. ¿Cuál es el Orden (O) de tiempo de ejecución del procedimiento P?

¿Es O(n)? ¿Por qué?

$\neq$k	$\neq$k	$\neq$k	$\neq$k	…	…	$\neq$k	$\neq$k	$\neq$k	k

Cálculo de T(n) - Ejercicios

Considere el siguiente procedimiento:

```
void P (int k, int n, int * A) // se omite el uso de la variable "esta"
{     int i,j; bool esta=false;
      for (i=0; i<n && !esta; i++)
            if (A[i]==k)
                  { esta=true;
                    for (j=0; j<n; j++) printf(…);
                  }
}
```

Donde, A es una arreglo de enteros de tamaño n. ¿Cuál es el Orden (O) de tiempo de ejecución del procedimiento P?

¿Es O(n)? ¿Por qué?

k	k	k	k	k	k	k	k	k	k

Cálculo de T(n) - Ejercicios

Considere el siguiente procedimiento:

```
void P (int n)
{     for (int i=1; i<n+1; i++)
            for (int j=i; j<n+1; j++)
                  for (int k=j; k<i+1; k++)
                        cout << i+j+k;
};
```

Calcule el Orden (O) de tiempo de ejecución del procedimiento P.

Nota: $\sum_{i=1}^{n} i^2$ = n(n+1)(2n+1)/6.

Resolver: T(n) = $\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=j}^{i} cte$, y comprobar que el término más grande positivo justifica que T(n) es O(n^2).

Aspectos importantes además de O(n)

- Si un algoritmo se va a utilizar sólo algunas veces, el costo de escritura y depuración puede ser el dominante.
- Si un programa se va a ejecutar sólo con entradas "pequeñas", el orden O(n) puede ser menos importante que el factor constante de la fórmula de tiempo de ejec.
- Un algoritmo eficiente pero complicado puede dificultar su mantenimiento.
- Un algoritmo eficiente en tiempo de ejecución pero que ocupa demasiado espacio de almacenamiento puede ser inadecuado.

Bibliografía

- **Estructuras de Datos y Análisis de Algoritmos**
  Mark Allen Weiss

  (Capítulo 2)

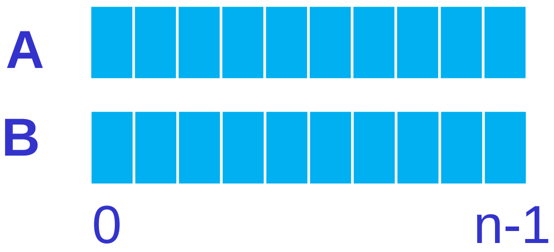
- **Estructuras de Datos y Algoritmos.**
  A. Aho, J. E. Hopcroft & J. D. Ullman

  (Capítulo 1, secciones 1.4 y 1.5**)**

Más ejercicios...

Considere la siguiente función en C/C++:

```
bool F (int *A, int *B, int n) {
    bool b=true;
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            b = b && (A[i] < B[j]);
    return b;
}
```



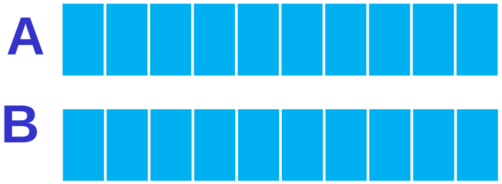
a) ¿Qué calcula la función F?

b) Determine el tiempo de ejecución para el peor caso de la función F y el Orden (O).

c) Desarrolle una función que calcule lo mismo que F en un menor orden de tiempo de ejecución en el peor caso.

Más ejercicios...

```
bool F (int *A, int *B, int n) {
    int minB = mínimo(B, n);
    int maxA = máximo(A, n);
    return (maxA < minB);
}
```



Determine el tiempo de ejecución para el peor caso de la función F y el Orden (O).

Más ejercicios...

Considere la siguiente función en C/C++:

```
bool F (int *A, int n) {
    bool b=true;
    for (int i=0; i<n; i++)
        for (int j=n-1; j>-1; j--)
            b = b && (A[i] == A[j]);
    return b;
}
```

a) ¿Qué calcula F?

b) Calcule el tiempo de ejecución para el peor caso de la función F y el Orden (O).

c) El problema que resuelve F podría resolverse en un menor orden (O) de tiempo de ejecución en el peor caso? Justifique.

Más ejercicios...

Considere la siguiente función en C/C++:

```
bool F (int *A, int n) {
    int i;
    for (i=0; i<n && A[0]==A[i]; i++) ;
    return i==n;
}
```

a) ¿Qué calcula F?

b) Calcule el tiempo de ejecución para el peor caso de la función F y el Orden (O).

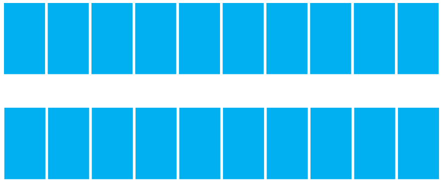
Más ejercicios...

Considere la siguiente función en C/C++:

```
bool F (int *A, int *B, int n){
  bool b=false;
  for (int i=0; i<n; i++)
     for (int j=0; j<n; j++)
           b = b || (A[i] < B[j]);
  return b;
}
```

A

B



a) ¿Qué calcula la función F?

b) Calcule el tiempo de ejecución para el peor caso de la función F y el Orden (O).

c) Escriba una Función G que resuelva el mismo problema que la función F pero en O(n) peor caso. Justifique.

Más ejercicios...

Considere la siguiente función en C/C++:

```
bool F (int *A, int n)
{  bool b=false;
   for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
              b = b || (A[i]==A[j] && i!=j);
   return b;
}
```

a) ¿Qué computa la función F?

b) Calcule el tiempo de ejecución para el peor caso de F y el Orden (O).

c) Escriba una Función G que resuelva el mismo problema que la función F pero en O(n) peor caso, asumiendo que el arreglo parámetro está ordenado de menor a mayor. Justifique.

d) ¿Puede resolverse el problema anterior en O(n) peor caso pero sin asumir que el arreglo parámetro está ordenado? Justifique.

Más ejemplos...

Ver tres problemas con soluciones en: [link]

Más ejercicios... (*sorting*)

Escriba un procedimiento *Sort012* en C/C++ que dado un arreglo de valores en el conjunto {0,1,2} de tamaño *n* lo ordene de menor a mayor en O(*n*) peor caso. Justifique.

Por ejemplo, si n=6, *Sort012* del arreglo [0,2,2,1,0,0] deja el arreglo [0,0,0,1,2,2].

Generalice el procedimiento sabiendo que los valores están en rango [0:k], siendo k una constante. Este algoritmo es un ejemplo del algoritmo de ordenación conocido como *bucket sort*.

Si k fuera una variable, ¿cuál sería el orden (O) del procedimiento anterior?

Más ejercicios... (*sorting*)

```
void sort012 (int * A, int n)
{  int cont0=0;
   int cont1=0;
   int i;
   for (i=0; i<n; i++){
       if (A[i]==0) cont0++;
       else if (A[i]==1) cont1++;
   }
   for (i=0; i<cont0; i++){ A[i]=0; }
   for (i=cont0; i<cont0+cont1; i++){ A[i]=1; }
   for (i=cont0+cont1; i<n; i++){ A[i]=2; }
}
```

conteo

ordenación

Es un algoritmo de ordenación por conteo.

¿Es O(n)?

Más ejercicios...

Escriba una función *esPermutacion* en C/C++ que dado un arreglo de enteros de tamaño n, retorne true si, y sólo si, el arreglo almacena una permutación de los enteros 0,...,n-1.

Se requiere que *esPermutacion* tenga O(n) en el peor caso. Justifique.

Por ejemplo, si n =5, *esPermutacion* retorna true para el arreglo [1,3,0,4,2] y false en los siguientes casos: [1,1,0,4,2] y [1,3,0,8,2].

Más ejercicios...

```
bool esPermutacion(int * A, int n){
   int i;
   bool * ESTA = new bool[n];
   for (i=0; i<n; i++){ ESTA[i] = false;}
   for (i=0; i<n && A[i]>=0 && A[i]<n && !ESTA[A[i]]; i++){
       ESTA[A[i]] = true;
   }
   delete [] ESTA;
   return (i==n);
}
```

¿Es O(n)? Justifique.

Más ejercicios...

Escriba una función *sonPermutaciones* en C/C++ que dados dos arreglos de tamaño n de valores enteros en el rango [0,K] retorne true si, y sólo si, los arreglos son permutaciones. K es un valor constante.

Se requiere que *sonPermutaciones* tenga O(n) en el peor caso. Justifique.

Por ejemplo, si n=5 y K=3, *sonPermutaciones* retorna true para los arreglos: [1,1,0,3,2], [0,1,2,3,1]; y, false para los siguientes arreglos: [1,1,0,3,2] y [0,3,0,1,2].

Más ejercicios... (*sorting*)

Desarrolle algoritmos de ordenación sobre arreglos de enteros de tamaño *n* siguiendo las siguientes estrategias:

- Busca el mínimo elemento, lo pone al inicio y luego aplica la misma estrategia a los restantes elementos (todos salvo el primero).
- Recorre el arreglo e inserta de manera ordenada cada elemento en un nuevo arreglo originalmente vacío.

Calcule el tiempo de ejecución y el orden (O) del primer algoritmo, conocido como *select sort*, y del segundo algoritmo, conocido como *insert sort*.