Estructuras de Datos y Algoritmos 1

Análisis de Algoritmos: Tiempo de Ejecución

http://www.dc.exa.unrc.edu.ar

Carlos Luna
cluna@fing.edu.uy

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 *tiempo de ejecución*

Tiempo de ejecución de un programa

Factores que intervienen:

- **Los datos de entrada al programa**
- La calidad del código generado por el compilador
- La naturaleza y rapidez de las instrucciones de máq.
- **La complejidad de tiempo del algoritmo base**

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 **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 - $\Omega(n)$

- 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 $\Omega(n^3)$

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)

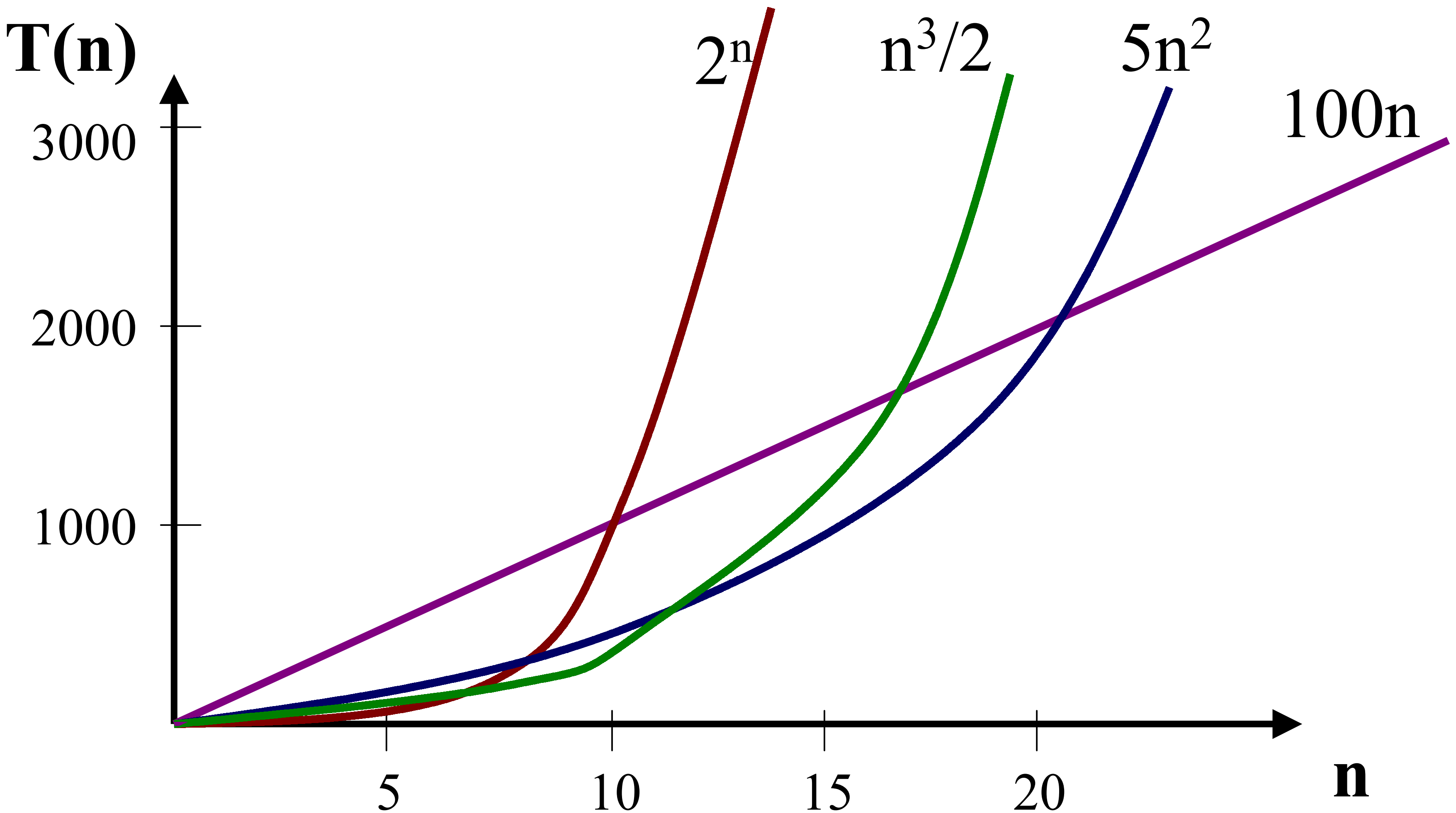
  – Si $n > 20$, P1 es más rápido que P2

  (para entradas "grandes" es mejor P1)

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

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)))**

⇒ 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 $O(n^3)$.

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) 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 $\boldsymbol{Sent_2}$

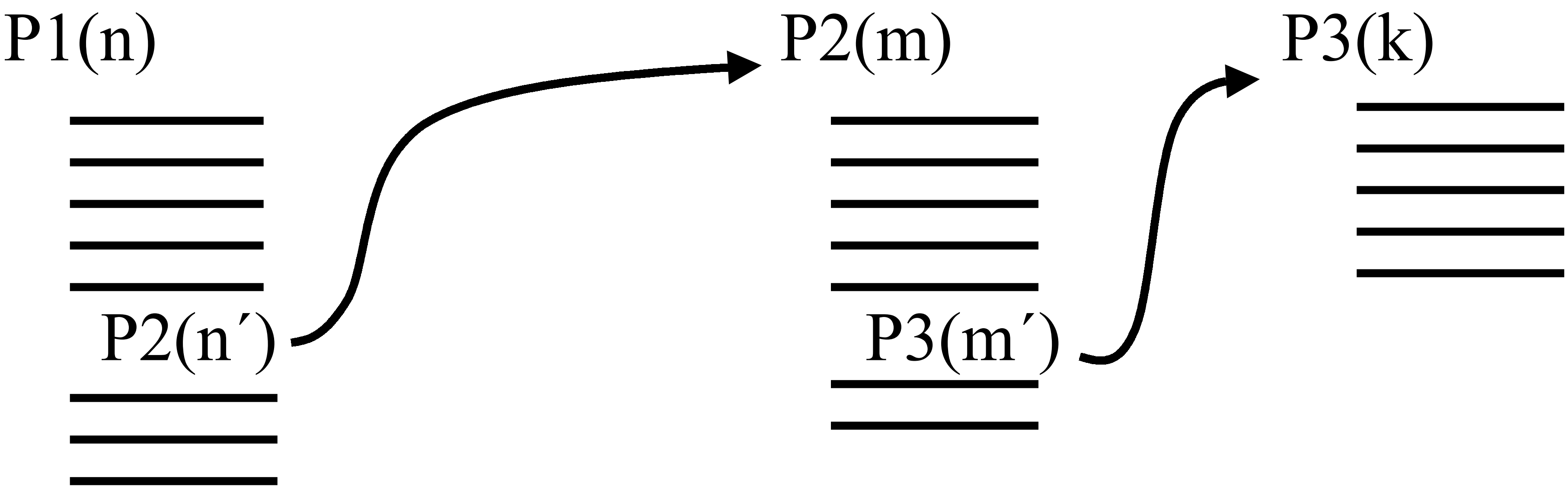
Cálculo de T(n) - Algunas reglas (cont)

- Para un **ciclo** es la suma, sobre todas las iteraciones del ciclo, 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

Escribir algoritmos en pseudocódigo para los siguientes problemas y determinar el tiempo de ejecución (<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 los siguientes fragmentos de programas:

- `for (i=0; i<n; i++) cout << A[i][i];`

- ```
  for (i=0; i<n; i++)
        for (j=0; j<n; j++) if (i== j) cout << A[i][j];
  ```

¿Qué hacen?, ¿Cuál es más eficiente y por qué?
  ```

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?

Refine la acción *intercambiar*.

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

T(n) para programas recursivos

- **Un ejemplo**: "factorial"

```
int fact (int n)
  { if (n>1) return n*fact(n-1);
    else return 1; }
```

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n > 1$) c es una constante

$\Rightarrow$ **T(n) es O(n)**. *Probarlo !!!*

- **Otro ejemplo**: Analizar que el tiempo de ejecución de un algoritmo de búsqueda dicotómica sobre una secuencia ordenada de n elementos es $O(\log_2(n))$

T(n) para programas recursivos
Métodos generales de resolución

Resolución de ecuaciones de recurrencia:

- Suposición de una solución (*guess*)
- <u>Expansión de recurrencias</u>
- Soluciones generales para clases de recurrencias
  - Soluciones homogéneas y particulares
  - Funciones Multiplicativas

....

Aspectos importantes además de O(n)

- Si un algoritmo se va a utilizar sólo algunas veces, el costo de 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 el mantenimiento del mismo.
- 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 en Pascal.**
  Mark Allen Weiss; Benjamin/Cummings Inc., 1993.
  (Capítulo 2)

- **Estructuras de Datos y Algoritmos.**
  A. Aho, J. E. Hopcroft & J. D. Ullman; Addison-Wesley, 1983.
  (Capítulo 1, secciones 1.4 y 1.5**)**
  (Capítulo 9: recurrencias**)**

Ejercicios adicionales propuestos

- **Los del final del capítulo 2 del libro:**
  Estructuras de Datos y Análisis de Algoritmos en Pascal.
  Mark Allen Weiss; Benjamin/Cummings Inc., 1993.
- **Los del final del capítulo 1 del libro:**
  Estructuras de Datos y Algoritmos.
  A. Aho, J. E. Hopcroft & J. D. Ullman; Addison-Wesley, 1983.
  (Capítulo 9: recurrencias**)**
- Calcular el tiempo de ejecución y el orden (O) de algoritmos vistos en los cursos previos de programción.
- Calcular el tiempo de ejecución y el orden (O) de los algoritmos de ordenación: merge-sort, insert-sort, quicksort y select-sort. Compararlos.