Diseño de Algoritmos - Algoritmos II

Nazareno Aguirre, Sonia Permigiani,
Gastón Scilingo, Simón Gutiérrez
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 1(c): Revisión de conceptos previos

Análisis de Algoritmos

En la construcción de algoritmos es necesario realizar tareas de análisis para:

- corrección
- eficiencia en tiempo
- eficiencia en espacio ocupado
- ...

Enfoques

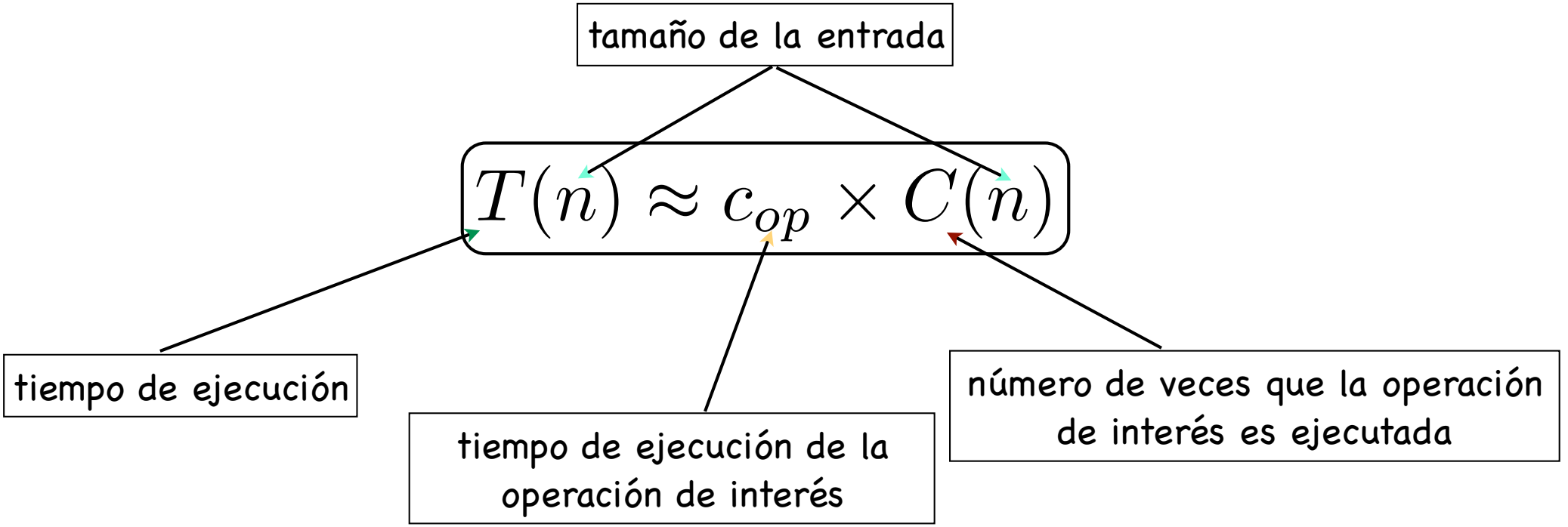
- analítico
- empírico

Ya sabemos cómo realizar algunas de estas tareas analíticamente. Repasémoslas.

Análisis de Eficiencia en Tiempo

El análisis de eficiencia en tiempo se realiza determinando el número de veces que la operación de interés es ejecutada, como una función sobre el tamaño de la entrada.

La operación de interés es aquella que contribuye en mayor medida al tiempo de ejecución del algoritmo estudiado.



Ejemplos de Tamaños de Entrada y Operaciones de Interés

Problema	**Medida del tamaño de la entrada**	**Operación de interés**
Búsqueda de una clave en una lista de n elementos	Número de elementos de la lista, i.e., n	Comparación de claves
Multiplicación de matrices	Dimensiones de las matrices, o el número total de elementos	Multiplicación de números
Chequeo de primalidad de un entero n	el tamaño de n, i.e., el número de dígitos de n (en representación binaria)	División
Algún problema típico de grafos	número de vértices y/o número de arcos	Visitar un vértice o atravesar un arco

Análisis Empírico de Eficiencia en Tiempo

En muchos casos, realizar una evaluación analítica del tiempo de ejecución de un algoritmo puede ser demasiado complicado. En estos casos, una alternativa útil es realizar un análisis empírico. Algunos de los puntos a tener en cuenta en el análisis empírico de eficiencia son los siguientes:

- seleccionar una muestra específica (típica) de entradas para el algoritmo
- utilizar unidades de tiempo para la medida, o contar el número de operaciones de interés realizadas
- analizar los datos obtenidos

El diseño de un análisis empírico debe evitar cuidadosamente diferentes tipos de amenazas de validez (ver http://web.pdx.edu/~stipakb/download/PA555/ResearchDesign.html)

Peor caso, caso promedio y mejor caso

En muchos casos, la eficiencia de un algoritmo no sólo depende del tamaño de la entrada, sino también de su forma. En estos casos, en el análisis del tiempo de ejecución de un algoritmo puede distinguirse entre:

- análisis de peor caso. $C_{worst}(n)$: máximo entre las entradas de tamaño n
- análisis de mejor caso. $C_{best}(n)$: mínimo entre las entradas de tamaño n
- análisis de caso promedio. $C_{avg}(n)$: promedio entre las entradas de tamaño n
  - el número de veces que la operación de interés será ejecutada para una entrada "típica"
  - NO es el promedio entre el mejor caso y el peor caso
  - el número esperado de operaciones de interés es interpretado como una variable aleatoria (con alguna suposición de la distribución de las posibles entradas)

Tasa de Crecimiento

La tasa de crecimiento de la función asociada al tiempo de ejecución, entendida como la clase de funciones a la cual pertenece (o por la cual está acotada) la función de eficiencia de un algoritmo, es de gran importancia.

Esta tasa de crecimiento nos permite predecir respuestas a algunas preguntas importantes, del estilo de las siguientes:

- Cuánto más rápido correrá este algoritmo en una computadora el doble de rápida de la que tengo?

- Cuánto tiempo más me insumirá resolver un problema del doble del tamaño?

Algunas tasas de crecimiento importantes

ent.	$log_2 n$	n	$n log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20	10^6	2×10^7	10^{12}	10^{18}		

Asintóticas de Tasas de Crecimiento

Una forma útil de comparar funciones de eficiencia ignorando factores constantes (y tamaños pequeños en las entradas) es a través de la consideración de asintóticas de la tasa de crecimiento de las funciones de eficiencia.

La notación usual para clases de funciones de eficiencia, relacionada a asintóticas, es la siguiente:

- $O(g(n))$: Clase de funciones que no crecen más rápido que g
- $\Omega(g(n))$: Clase de funciones que no crecen menos rápido que g
- $\Theta(g(n))$: Clase de funciones que tienen exactamente la misma tasa de crecimiento que g

Clases Básicas de Eficiencia

1	constante
log n	logarítmica
n	lineal
n log n	n-log-n
n^2	cuadrática
n^3	cúbica
2^n	exponencial
n!	factorial

Análisis de Eficiencia de Algoritmos Iterativos

Las actividades usuales asociadas al análisis de eficiencia de algoritmos iterativos son las siguientes:

- Decidir cuál es el tamaño de la entrada para el problema en cuestión
- Identificar la operación de interés para el algoritmo
- Determinar cuáles son los casos que corresponden a las configuraciones peor, mejor o promedio para el algoritmo en cuestión (de acuerdo al análisis que se quiera realizar)
- Obtener una expresión que denote el número de veces que la operación de interés es ejecutada (para cada uno de los casos anteriores)
- Simplificar la expresión anterior (que suele involucrar sumatorias)

Análisis de Eficiencia de Algoritmos Recursivos

Las actividades usuales asociadas al análisis de eficiencia de algoritmos recursivos son las siguientes:

- Decidir cuál es el tamaño de la entrada para el problema en cuestión
- Identificar la operación de interés para el algoritmo
- Determinar cuáles son los casos que corresponden a las configuraciones peor, mejor o promedio para el algoritmo en cuestión (de acuerdo al análisis a realizar)
- Obtener una ecuación de recurrencia que denote el número de veces que la operación de interés es ejecutada (para cada uno de los casos anteriores)
- Resolver la expresión anterior hasta obtener una forma "cerrada"

Análisis de Corrección de Algoritmos

Uno de los aspectos fundamentales asociados a la calidad de software es la corrección del mismo, entendida como el grado con el cual el software se ajusta a los requisitos del mismo.

Al igual que para el análisis de eficiencia, podemos evaluar la corrección de programas de al menos dos maneras:

- Analíticamente (verificación de programas, cálculo de precondición más débil, inducción, etc.)
- Empíricamente (testing)

Análisis Empírico de Corrección de Algoritmos

El testing es en la actualidad el mecanismo fundamental para la evaluación de corrección de software, y constituye un área de investigación muy rica y activa.

El proceso básico de testing consiste en:

- elegir una muestra de entradas para el programa
- determinar, para cada una de las entradas elegidas, cuáles deberían ser los resultados obtenidos (de acuerdo a la especificación del programa)
- ejecutar el programa para las entradas elegidas
- contrastar los resultados obtenidos con los resultados esperados

Análisis Empírico de Corrección de Algoritmos

Existe una variedad de técnicas, metodologías y herramientas para soporte al testing. Muchas de éstas ayudan en la decisión o producción de elementos ligados a los siguientes problemas:

- elección de una muestra de entradas para el programa
  - el análisis de cobertura ayuda a evaluar cuán representativa es la muestra elegida, de acuerdo a diferentes criterios
  - las técnicas de generación de casos de test ayudan a producir casos de test para programas (automáticamente, en algunos casos)
- contrastar los resultados obtenidos con los esperados
  - las herramientas de soporte al testing suelen permitir automatizar en parte esta tarea (véase JUnit para el caso de Java, HSpec para Haskell)

Verificación de Programas

Si bien el testing es en la actualidad el mecanismo de más amplia adopción para la evaluación de corrección de software, éste tiene serias limitaciones. En particular, en general no nos permite garantizar que un programa no tiene errores.

Un mecanismo más poderoso para la evaluación de la corrección de programas consiste en verificar (usando herramientas matemático-lógicas) la corrección del mismo.

El proceso básico de verificación consiste en:

- formalizar la especificación asociada al programa
- verificar que el programa respeta la especificación. Esta tarea suele contar de las siguientes:
  - demostrar que, para toda entrada admisible, si el programa termina entonces retorna el resultado esperado (corrección parcial).
  - demostrar que, para toda entrada admisible, el programa termina.

Verificación de Programas Funcionales

La técnica fundamental para la verificación de programas funcionales es la inducción. Consideremos por ejemplo la siguiente especificación y definición para la función f:

$$\langle \forall i \ : i \in Nat : \ f.i = i^2 \rangle$$

$$\begin{array}{rcl} f.0 & \doteq & 0 \\ f.(i+1) & \doteq & f.i + 2 * i + 1 \end{array}$$

Debemos comprobar (por inducción) que la función f satisface su especificación.

Verificación de Programas Funcionales

Caso base

$$
\begin{aligned}
& f.0 \\
= & \ \{ \text{ definición de } f \ \} \\
& 0 \\
= & \ \{ \text{ álgebra } \} \\
& 0^2
\end{aligned}
$$

Paso inductivo

$$
\begin{aligned}
& f.(i+1) \\
= & \ \{ \text{ definición de } f \ \} \\
& f.i + 2 * i + 1 \\
= & \ \{ \text{ hipótesis inductiva } \} \\
& i^2 + 2 * i + 1 \\
= & \ \{ \text{ álgebra } \} \\
& (i+1)^2
\end{aligned}
$$

Verificación de Programas Imperativos

La técnica fundamental para la verificación de programas imperativos es la asociada a triplas de Hoare, y cálculo de precondición más débil.

Esta técnica consiste en acompañar los programas con aserciones (en lógica de primer orden, por ejemplo), para la precondición y postcondición asociadas al programa:

$$\left.\begin{array}{l}\{\texttt{Precondición}\}\\ \texttt{Programa}\\ \{\texttt{Postcondición}\}\end{array}\right\} \text{Corrección total}$$

Este programa anotado es visto como una fórmula, cuya validez establece que bajo todas las entradas que satisfacen la precondición, el programa termina, y lo hace en un estado que satisface la postcondición.

Ejemplos de Programas Anotados

$\{x = X \ \& \ y = Y\}$

```
aux := x;
x := y;
y := aux;
```

$\{x = Y \ \& \ y = X\}$

$\{true\}$

```
i := 1;
while (i<=length(A)) do
   j := i;
   while (j>1) do
      if (A[j-1]>A[j]) then
         swap(j-1,j)
      j := j-1
   end
   i := i+1
end
```

$\{\forall x \in [1..\, length(A) - 1] : A[x] \leq A[x + 1]\}$

Programas Anotados como Aserciones Lógicas: Reglas de Inferencia

Asignación

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}$$

Composición en secuencia

$$\frac{\{\phi\}\ p_1\ \{\eta\} \qquad \{\eta\}\ p_2\ \{\psi\}}{\{\phi\}\ p_1; p_2\ \{\psi\}}$$

Ejecución condicional

$$\frac{\{\phi \wedge B\}\ p_1\ \{\psi\} \qquad \{\phi \wedge \neg B\}\ p_2\ \{\psi\}}{\{\phi\}\ \text{if } B \text{ then } p_1 \text{ else } p_2 \text{ fi}\ \{\psi\}}$$

Iteración

$$\frac{\{\phi \wedge B\}\ p\ \{\phi\}}{\{\phi\}\ \text{while } B \text{ do } p \text{ od}\ \{\phi \wedge \neg B\}}$$

Consecuencia

$$\frac{\phi' \Rightarrow \phi \qquad \{\phi\}\ p\ \{\psi\} \qquad \psi \Rightarrow \psi'}{\{\phi'\}\ p\ \{\psi'\}}$$

Programas Anotados como Aserciones Lógicas: Iteración

Regla de inferencia para iteración:

La regla de inferencia para razonar sobre programas con iteración es lo que hace interesante al sistema de inferencia. Esta regla involucra nociones importantes, como las nociones de invariante de ciclo Inv y función cota t:

Si se cumple:

(1) $\{\phi\}$ Init $\{Inv\}$

(2) $\{C \wedge Inv\}$ CC $\{Inv\}$

(3) $\neg C \wedge Inv \Rightarrow \psi$

(4) $\{t = T\}$ CC $\{t < T\}$

(5) $C \Rightarrow t > 0$

entonces:

$\{ \phi \}$
Init
while C do
CC
od
$\{ \psi \}$

Características de la verificación usando lógica de Hoare

- Puede extenderse a programas concurrentes (Owicki-Gries).
- No puede automatizarse (el problema de la verificación es indecidible).
- Introduce conceptos de gran importancia (aserciones, especificaciones, invariantes, etc.) que influyen en lenguajes y metodologías de programación.

Ejemplo 1: Búsqueda Secuencial

Realicemos para este ejemplo el análisis de corrección y tiempo de ejecución:

```
Algoritmo int BusquedaSecuencial(int A[0..n-1], int K)
   // Busca un valor dado en un arreglo, mediante búsqueda secuencial.
   // Entrada: un arreglo A[0..n-1] y una clave K
   // Salida: el índice de la primera posición de A en que está
   // almacenado K, o -1 si K no pertenece a A
   i ← 0
   while i<n and A[i] ≠ K do
      i ← i+1
   if i<n then return i
          else return -1
```

Ejemplo 2: Selection Sort

Realicemos para este segundo ejemplo el análisis de corrección y tiempo de ejecución:

```
Algoritmo SelectionSort(A[0..n-1])
   // Ordena un arreglo mediante el proceso de selección.
   // Entrada: un arreglo A[0..n-1] de elementos comparables
   // Salida: el arreglo A[0..n-1], ordenado de manera ascendente.
   for i ← 0 to n-2 do
      min ← i
      for j ← i+1 to n-1 do
         if A[j]<A[min] then min ← j
      swap(A[i],A[min])
```