Matemáticas Discretas

Análisis de algoritmos

Gabriel Diéguez
gsdieguez@ing.puc.cl

Fernando Suárez
fsuarez1@ing.puc.cl

Departamento de Ciencia de la Computación
Escuela de Ingeniería
Pontificia Universidad Católica de Chile

9 de octubre de 2019

Objetivos

1. Aplicar inducción como técnica para demostración de propiedades en conjuntos discretos y como técnica de definición formal de objetos discretos.
2. Demostrar formalmente que un algoritmo simple funciona correctamente, y determinar la eficiencia de un algoritmo, desarrollando una notación asintótica para estimar el tiempo de ejecución.

Contenidos

Introducción

¿Qué es un algoritmo?

- Es difícil dar una definición formal...
- Método o conjunto de instrucciones que sirven para resolver un problema.
  - Un poco amplio, ¿no?

Nos interesan los métodos que resuelven problemas *computacionales*.

- Reciben un INPUT (representación de datos de entrada).
- Entregan un OUTPUT que depende del INPUT y cumple ciertas condiciones.

Introducción

Diremos entonces que un **algoritmo** es un método para convertir un INPUT válido en un OUTPUT. A estos métodos les exigiremos ciertas propiedades:

- Precisión: cada instrucción debe ser planteada de forma precisa y no ambigua.
- Determinismo: cada instrucción tiene un único comportamiento que depende sólo del input.
- Finitud: el algoritmo está compuesto por un conjunto finito de instrucciones.

Este tipo de formalismo es el que estudiaremos en este capítulo.

Introducción

El análisis de algoritmos es una disciplina de la Ciencia de la Computación que tiene dos objetivos:

- Estudiar cuándo y por qué los algoritmos son **correctos** (es decir, hacen lo que dicen que hacen).
- Estimar la cantidad de **recursos** computacionales que un algoritmo necesita para su ejecución.

De esta manera, podemos, por ejemplo:

- Entender bien los algoritmos, para luego reutilizarlos total o parcialmente.
- Determinar qué mejorar de un algoritmo para que sea más eficiente.

Introducción

Usaremos pseudo-código para escribir algoritmos.

- Instrucciones usuales como **if**, **while**, **return**...
- Notaciones cómodas para arreglos, conjuntos, propiedades lógicas, etc.

Consideraremos que los algoritmos tienen:

- **Precondiciones:** representan el input del programa.
- **Postcondiciones:** representan el output del programa, lo que hace el algoritmo con el input.

Corrección de algoritmos

Queremos determinar cuándo un algoritmo es correcto; es decir, hace lo que dice que hace. ¿Qué significa esto más formalmente?

Un algoritmo es **correcto** si para todo INPUT válido, el algoritmo se detiene y produce un OUTPUT correcto.

Entonces, ¿cuándo es incorrecto?

Un algoritmo es **incorrecto** si existe un INPUT válido para el cual el algoritmo no se detiene o produce un OUTPUT incorrecto.

Algoritmos iterativos

Debemos demostrar dos cosas:

- **Corrección parcial:** si el algoritmo se detiene, se cumplen las postcondiciones.
- **Terminación:** el algoritmo se detiene.

Nos preocupamos sólo de los *loops* de los algoritmos (¿por qué?).

Estos loops tienen una condición G que determina si se siguen ejecutando:

while(G)
...
end

Algoritmos iterativos

Para demostrar corrección parcial, buscamos un **invariante** $I(k)$ para los loops:

- Una propiedad I que sea verdadera en cada paso k de la iteración.
- Debe relacionar a las variables presentes en el algoritmo.
- Al finalizar la ejecución, debe asegurar que las postcondiciones se cumplan.

Algoritmos iterativos

Una vez que encontramos un invariante, demostramos la corrección del loop inductivamente:

- **Base:** las precondiciones deben implicar que $I(0)$ es verdadero.
- **Inducción:** para todo natural $k > 0$, si G e $I(k)$ son verdaderos antes de la iteración, entonces $I(k+1)$ es verdadero después de la iteración.
- **Corrección:** inmediatamente después de terminado el loop (i.e. cuando G es falso), si $k = N$ e $I(N)$ es verdadero, entonces la postcondiciones se cumplen.

Y para demostrar terminación, debemos mostrar que existe un k para el cual G es falso.

Algoritmos iterativos

Ejercicio

Escriba un algoritmo que multiplique dos números naturales (sin usar la multiplicación):

- **Pre:** $n, m \in \mathbb{N}$.
- **Post:** $p = n \cdot m$.

Demuestre que su algoritmo es correcto.

Algoritmos recursivos

En el caso de los algoritmos recursivos, no necesitamos dividir la demostración en corrección parcial y terminación (¿por qué?).

- Basta demostrar por inducción la propiedad (corrección) deseada.
- En general, la inducción se realiza sobre el tamaño del input.

Ejercicio

Escriba un algoritmo recursivo que encuentre el máximo elemento de un arreglo:

- **Pre:** un arreglo $A = [a_0, a_1, \ldots, a_{n-1}]$, y un natural n (largo del arreglo).
- **Post:** $m = \max(A)$.

Demuestre que el algoritmo es correcto.

Complejidad de algoritmos

Ya vimos cómo determinar cuando un algoritmo era correcto.

- Esto no nos asegura que el algoritmo sea útil en la práctica.
- Necesitamos estimar su tiempo de ejecución.
  - En función del tamaño del input.
  - Independiente de: lenguaje, compilador, hardware...

Lo que nos interesa entonces no es el tiempo *exacto* de ejecución de un algoritmo, sino que su comportamiento a medida que crece el input.

- Introduciremos notación que nos permitirá hablar de esto.

Complejidad de algoritmos

Lo que nos interesa entonces no es el tiempo *exacto* de ejecución de un algoritmo, sino que su comportamiento a medida que crece el input.

- Introduciremos notación que nos permitirá hablar de esto.

Vamos a ocupar funciones de dominio natural ($\mathbb{N}$) y recorrido real postivo ($\mathbb{R}^+$).

- El dominio será el tamaño del input de un algoritmo.
- El recorrido será el tiempo necesario para ejecutar el algoritmo.

Notación asintótica

Sea $f : \mathbb{N} \to \mathbb{R}^+$.

Definición

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(g(n) \leq c \cdot f(n))\}$$

Diremos que $g \in O(f)$ es a lo más de orden f o que es $O(f)$.

Sea $f : \mathbb{N} \to \mathbb{R}^+$.

Definición

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(g(n) \geq c \cdot f(n))\}$$

Diremos que $g \in \Omega(f)$ es al menos de orden f o que es $\Omega(f)$.

Notación asintótica

Sea $f : \mathbb{N} \to \mathbb{R}^+$.

Definición

$\Theta(f) = O(f) \cap \Omega(f)$

Diremos que $g \in \Theta(f)$ es exactamente de orden f o que es $\Theta(f)$.

Ejercicio

Demuestre que $g \in \Theta(f)$ si y sólo si existen $c, d \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $\forall n \geq n_0$: $c \cdot f(n) \leq g(n) \leq d \cdot f(n)$.

Notación asintótica

Ejercicios

Demuestre que:

1. $f(n) = 60n^2$ es $\Theta(n^2)$.
2. $f(n) = 60n^2 + 5n + 1$ es $\Theta(n^2)$.

¿Qué podemos concluir de estos dos ejemplos?

- Las constantes no influyen.
- En funciones polinomiales, el mayor exponente "manda".

Notación asintótica

Ejercicio

Demuestre que $f(n) = \log_2(n)$ es $\Theta(\log_3(n))$.

¿Qué podemos concluir de este ejemplo?

- Nos podemos independizar de la base del logaritmo.

Notación asintótica

Podemos formalizar las conclusiones anteriores:

Teorema

Si $f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_2 \cdot n^2 + a_1 \cdot n + a_0$, con $a_i \in \mathbb{R}$ y $a_k > 0$, entonces f es $\Theta(n^k)$.

Teorema

Si $f(n) = \log_a(n)$ con $a > 1$, entonces para todo $b > 1$ se cumple que f es $\Theta(\log_b(n))$.

Ejercicio

Demuestre los teoremas.

Notación asintótica

Las funciones más usadas para los órdenes de notación asintótica tienen nombres típicos:

Notación	Nombre
$\Theta(1)$	Constante
$\Theta(\log n)$	Logarítmico
$\Theta(n)$	Lineal
$\Theta(n \log n)$	$n \log n$
$\Theta(n^2)$	Cuadrático
$\Theta(n^3)$	Cúbico
$\Theta(n^k)$	Polinomial
$\Theta(m^n)$	Exponencial
$\Theta(n!)$	Factorial

con $k \geq 0, m \geq 2$.

Volviendo a complejidad...

Queremos encontrar una función $T(n)$ que modele el tiempo de ejecución de un algoritmo.

- Donde n es el tamaño del input.
- No queremos valores exactos de T para cada n, sino que una notación asintótica para ella.
- Para encontrar T, contamos las instrucciones ejecutadas por el algoritmo.
- A veces contaremos cierto tipo de instrucciones que son relevantes para un algoritmo particular.

Contando instrucciones

Ejercicio

Considere el siguiente trozo de código:

```
1: x = 0
2: for i = 1 to n  do
3:     for j = 1 to i do
4:         x = x + 1
```

Encuentre una notación asintótica para la cantidad de veces que se ejecuta la instrucción 4 en función de n.

Contando instrucciones

Ejercicio

Considere el siguiente trozo de código:

1: $x = 0$
2: $j = n$
3: **while** $j \geq 1$ **do**
4:     **for** $i = 1$ **to** j **do**
5:         $x = x + 1$
6:     $j = \lfloor \frac{j}{2} \rfloor$

Encuentre una notación asintótica para la cantidad de veces que se ejecuta la instrucción 5 en función de n.

Algoritmos iterativos

Consideremos el siguiente algoritmo de búsqueda en arreglos:

$\textsc{Búsqueda}(A, n, k)$

Input: un arreglo de enteros $A = [a_0, \ldots, a_{n-1}]$, un natural $n > 0$ correspondiente al largo del arreglo y un entero k.
Output: el índice de k en A, -1 si no está.

for $i = 0$ **to** $n - 1$ **do**
    if $a_i = k$ **then**
        return i
return -1

Algoritmos iterativos

¿Qué instrucción(es) contamos?

- Deben ser representativas de lo que hace el problema.
- En este caso, por ejemplo 3 y 4 no lo son (¿por qué?).
- La instrucción 2 si lo sería, y más específicamente la comparación.
  - Las comparaciones están entre las instrucciones que se cuentan típicamente, sobre todo en búsqueda y ordenación.

¿Respecto a qué parámetro buscamos la notación asintótica?

- En el ejemplo, es natural pensar en el tamaño del arreglo n.

En conclusión: queremos encontrar una notación asintótica (ojalá Θ) para la cantidad de veces que se ejecuta la comparación de la línea 2 en función de n. Llamaremos a esta cantidad $T(n)$.

Algoritmos iterativos

Ahora, ¿$T(n)$ depende sólo de n?

- El contenido del arreglo influye en la ejecución del algoritmo.
- Estimaremos entonces el tiempo para el **peor caso** (cuando el input hace que el algoritmo se demore la mayor cantidad de tiempo posible) y el **mejor caso** (lo contrario) para un tamaño de input n.

En nuestro ejemplo:

- **Mejor caso:** $a_0 = k$. Aquí la línea 2 se ejecuta una vez, y luego $T(n)$ es $\Theta(1)$.
- **Peor caso:** k no está en A. La línea 2 se ejecutará tantas veces como elementos en A, y entonces $T(n)$ es $\Theta(n)$.
- Diremos entonces que el algoritmo $\textsc{Búsqueda}$ es de **complejidad** $\Theta(n)$ o lineal en el peor caso, y $\Theta(1)$ o constante en el mejor caso.

Algoritmos iterativos

Ejercicio

Determine la complejidad en el mejor y peor caso:

INSERT-SORT(A, n)
Input: un arreglo $A = [a_0, \ldots, a_{n-1}]$ y su largo $n > 0$.
Output: el arreglo está ordenado al terminar el algoritmo.

for $i = 1$ **to** $n - 1$ **do**
    $j = i$
    while $a_{j-1} > a_j \wedge j > 0$ **do**
        $t = a_{j-1}$
        $a_{j-1} = a_j$
        $a_j = t$
        $j = j - 1$

Algoritmos iterativos

En general, nos conformaremos con encontrar la complejidad del peor caso.

- Es la que más interesa, al decirnos qué tan mal se puede comportar un algoritmo en la práctica.

Además, a veces puede ser difícil encontrar una notación Θ.

- ¿Con qué nos basta?
- Es suficiente con una buena estimación O, tanto para el mejor y el peor caso.
- Nos da una cota superior para el tiempo de ejecución del algoritmo.

Algoritmos recursivos

En el caso de los algoritmos recursivos, el principio es el mismo: contar instrucciones.

- Buscamos alguna(s) instrucción(es) representativa.
- Contamos cuántas veces se ejecuta en cada ejecución del algoritmo.
- ¿Cuál es la diferencia?

Tenemos que considerar las llamadas recursivas al algoritmo.

- Esto hará que aparezcan fórmulas recursivas que deberemos resolver.

Algoritmos recursivos: un ejemplo

¿Cómo buscamos un elemento en un arreglo previamente ordenado?

```
BinarySearch(a, A, i, j)
if i > j then
    return −1
else if i = j then
    if A[i] = a then
        return i
    else
        return −1
else
    p = ⌊(i+j)/2⌋
    if A[p] < a then
        return BinarySearch(a, A, p + 1, j)
    else if A[p] > a then
        return BinarySearch(a, A, i, p − 1)
    else
        return p
```

Algoritmos recursivos: un ejemplo

- ¿Qué operaciones contamos?
- ¿Cuál es el peor caso?

Ejercicio

Encuentre una función $T(n)$ para la cantidad de comparaciones que realiza el algoritmo $\textsc{BinarySearch}$ en el peor caso, en función del tamaño del arreglo.

Respuesta:

$$T(n) = \begin{cases} 3 & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + 4 & n > 1 \end{cases}$$

Esta es una **ecuación de recurrencia**.

Algoritmos recursivos: ecuaciones de recurrencia

Necesitamos *resolver* esta ecuación de recurrencia.

- Es decir, encontrar una expresión que no dependa de T, sólo de n.
- Técnica básica: sustitución de variables.

¿Cuál sustitución para n nos serviría en el caso anterior?

$$T(n) = \begin{cases} 3 & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + 4 & n > 1 \end{cases}$$

Ejercicio

Resuelva la ecuación ocupando la sustitución $n = 2^k$.

Respuesta: $T(n) = 4 \cdot \log_2(n) + 3$, con n potencia de 2.

Notación asintótica condicional

Sea $P \subseteq \mathbb{N}$.

Definición

$$O(f \mid P) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)$$
$$(n \in P \to g(n) \leq c \cdot f(n))\}$$

Las notaciones $\Omega(f \mid P)$ y $\Theta(f \mid P)$ se definen análogamente.

Volviendo al ejemplo...

Tenemos que $T(n) = 4 \cdot \log_2(n) + 3$, con n potencia de 2. ¿Qué podemos decir sobre la complejidad de T?

Sea $POTENCIA_2 = \{2^i \mid i \in \mathbb{N}\}$. Entonces:

$$T \in \Theta(\log_2(n) \mid POTENCIA_2)$$

Pero queremos concluir que $T \in \Theta(\log_2(n))$...

- O al menos $O(\log_2(n))$.
- ¿Ideas? ¡Inducción! :D

Para el ejemplo anterior:

Ejercicio

Demuestre que si $T \in O(\log_2(n) \mid POTENCIA_2)$, entonces $T \in O(\log n)$.

Algunas observaciones:

- Demostraremos que $(\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(T(n) \leq c \cdot \log_2(n))$.
- Primero, debemos estimar n_0 y c (expandiendo T por ejemplo).
- ¿Cuál principio de inducción usamos?

Definición

Una función $f : \mathbb{N} \to \mathbb{R}^+$ es **asintóticamente no decreciente** si:

$$(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(f(n) \leq f(n+1))$$

Ejemplos

Las funciones $\log_2 n$, n, n^k y 2^n son asintóticamente no decrecientes.

Ecuaciones de recurrencia: otra técnica

Definición

Dado un natural $b > 0$, una función $f : \mathbb{N} \to \mathbb{R}^+$ es b-**armónica** si $f(b \cdot n) \in O(f)$.

Ejemplos

Las funciones $\log_2 n$, n y n^k son b-armónicas para cualquier b.
La función 2^n no es 2-armónica (porque $2^{2n} \notin O(2^n)$).

Ecuaciones de recurrencia: otra técnica

Sean $f, g : \mathbb{N} \to \mathbb{R}^+$, un natural $b > 1$ y

$$POTENCIA_b = \{b^i \mid i \in \mathbb{N}\}$$

Teorema

Si f, g son asintóticamente no decrecientes, g es b-armónica y $f \in O(g \mid POTENCIA_b)$, entonces $f \in O(g)$.

Ejercicio

Demuestre el teorema.

Ecuaciones de recurrencia: otra técnica

Volviendo al ejemplo de BinarySearch...

Ejercicio

Demuestre que $T \in O(\log n)$ usando el teorema anterior.

Algunas observaciones:

- Ya sabemos que $T \in O(\log_2(n) \mid POTENCIA_2)$.
- Ya sabemos que $\log_2(n)$ es asintóticamente no decreciente y 2-armónica.
- Debemos demostrar entonces que T es asintóticamente no decreciente. ¿Alguien adivina cómo?

Dividir para conquistar

- Muchos algoritmos conocidos y usados en la práctica se basan en dividir el input en instancias más pequeñas para resolverlas recursivamente.
- Típicamente, existe un umbral n_0 desde el cual se resuelve recursivamente el problema (es decir, para inputs de tamaño $n \geq n_0$).
- Se divide el input por una constante b y se aproxima a un entero (usando $\lfloor\rfloor$ o $\lceil\rceil$), haciendo a_1 y a_2 llamadas recursivas para cada caso.
- Además, en general se hace un procesamiento adicional antes o después de las llamadas recursivas, que llamaremos $f(n)$.

Dividir para conquistar: un ejemplo

Ejercicio

¿Cómo ordenamos dos listas ya ordenadas en una?

$$L_1 = \{4, 7, 17, 23\}$$
$$L_2 = \{1, 9, 10, 15\}$$

¿Cómo podemos ocupar esta técnica para ordenar una lista?
¿Cuál es la complejidad de este algoritmo?

Teorema Maestro

Teorema

Si $a_1, a_2, b, c, c_0, d \in \mathbb{R}^+$ y $b > 1$, entonces para una recurrencia de la forma

$$T(n) = \begin{cases} c_0 & 0 \leq n < n_0 \\ a_1 \cdot T\left(\lceil \frac{n}{b} \rceil\right) + a_2 \cdot T\left(\lfloor \frac{n}{b} \rfloor\right) + c \cdot n^d & n \geq n_0 \end{cases}$$

se cumple que

$$T(n) \in \begin{cases} \Theta(n^d) & a_1 + a_2 < b^d \\ \Theta(n^d \cdot log(n)) & a_1 + a_2 = b^d \\ \Theta(n^{log_b(a_1+a_2)}) & a_1 + a_2 > b^d \end{cases}.$$

Ejercicio

¿Qué valor debe cumplir n_0?

Teorema Maestro

Teorema

Si $a_1, a_2, b, c, c_0, d \in \mathbb{R}^+$ y $b > 1$, entonces para una recurrencia de la forma

$$T(n) = \begin{cases} c_0 & 0 \leq n < \frac{b}{b-1} \\ a_1 \cdot T\left(\lceil \frac{n}{b} \rceil\right) + a_2 \cdot T\left(\lfloor \frac{n}{b} \rfloor\right) + c \cdot n^d & n \geq \frac{b}{b-1} \end{cases}$$

se cumple que

$$T(n) \in \begin{cases} \Theta(n^d) & a_1 + a_2 < b^d \\ \Theta(n^d \cdot log(n)) & a_1 + a_2 = b^d \\ \Theta(n^{log_b(a_1+a_2)}) & a_1 + a_2 > b^d \end{cases}.$$

Ejercicio

Demuestre el teorema.

Volviendo al ejemplo...

Ejercicio

¿Cuál es la complejidad de MergeSort?

Matemáticas Discretas

Análisis de algoritmos

Gabriel Diéguez
gsdieguez@ing.puc.cl

Fernando Suárez
fsuarez1@ing.puc.cl

Departamento de Ciencia de la Computación
Escuela de Ingeniería
Pontificia Universidad Católica de Chile

9 de octubre de 2019