Análisis de eficiencia

Nociones básicas

Segundo semestre 2022

IIC2283

Prof. Nicolás Van Sint Jan

Outline

Notación asintótica

Ecuaciones de recurrencia: búsqueda binaria

Outline

Una noción general de algoritmo

Sea Σ un alfabeto.

Vamos a pensar en un **algoritmo** $\mathcal{A}$ como una función $\mathcal{A} : \Sigma^* \to \Sigma^*$.

Esta es una representación general que incluye tanto a **problemas de decisión** como a **problemas de computación**.

¿ Cómo se representa un **problema de decisión** ?

Tiempo de ejecución de un algoritmo

Sea $\mathcal{A}$ un algoritmo. Asociamos a $\mathcal{A}$ una **función de tiempo de ejecución**

$$\text{tiempo}_{\mathcal{A}} : \Sigma^* \to \mathbb{N}$$

tal que:

$$\text{tiempo}_{\mathcal{A}}(w) \quad := \quad \text{número de pasos realizados por } \mathcal{A} \text{ con entrada } w \in \Sigma^*.$$

Para definir esta función tenemos que definir qué **operaciones** vamos a contar, y qué **costo** les asignamos.

Tiempo de ejecución de un algoritmo en el peor caso

El primer tipo de análisis que vamos a realizar de la complejidad de un algoritmo va a estar basado en el **peor caso**.

Definición

Para cada algoritmo $\mathcal{A}$ asociamos una **función peor-caso** $t_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ tal que

$$t_{\mathcal{A}}(n) \;=\; \max\{\text{tiempo}_{\mathcal{A}}(w) \mid w \in \Sigma^* \text{ y } |w| = n\},$$

donde $|w|$ es el largo de w.

Orden de un algoritmo

En muchos casos, nos interesa conocer el **"orden"** de un algoritmo en lugar de su complejidad exacta.

- Queremos decir que un algoritmo es **lineal** o **cuadrático**, en lugar de decir que su complejidad es $3n^2 + 17n + 22$

¿ Que noción matemática nos puede servir para esto ?

Supuesto

La **complejidad de un algoritmo** va a ser medida en términos de funciones de la forma $f : \mathbb{N} \to \mathbb{R}_0^+$, donde $\mathbb{R}^+ = \{r \in \mathbb{R} \mid r > 0\}$ y $\mathbb{R}_0^+ = \mathbb{R}^+ \cup \{0\}$

Estas funciones incluyen a las funciones definidas en las transparencias anteriores, y también sirven para modelar el tiempo de ejecución de un algoritmo.

La notación $\mathcal{O}(f)$

Definición

Sea $f : \mathbb{N} \to \mathbb{R}_0^+$ una función. Se define el **conjunto** $\mathcal{O}(f)$ tal que

$$\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ g(n) \leq c \cdot f(n)\}$$

Decimos entonces que $g \in \mathcal{O}(f)$.

- También usamos la notación g es $\mathcal{O}(f)$, lo cual es formalizado como $g \in \mathcal{O}(f)$

Ejemplo

Demuestre que $3n^2 + 17n + 22 \in O(n^2)$

Las notaciones $\Omega(f)$ y $\Theta(f)$

Definición

Sea $f : \mathbb{N} \to \mathbb{R}_0^+$ una función. Se definen los **conjuntos $\Omega(f)$ y $\Theta(f)$** tal que

$$\begin{aligned} \Omega(f) &= \{g : \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ c \cdot f(n) \leq g(n)\} \\ \Theta(f) &= O(f) \cap \Omega(f) \end{aligned}$$

Ejercicios

1. Demuestre que $3n^2 + 17n + 22 \in \Theta(n^2)$
2. Demuestre que $g \in \Theta(f)$ si y sólo si existen $c, d \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tal que para todo $n \geq n_0$: $c \cdot f(n) \leq g(n) \leq d \cdot f(n)$

Ejercicios

1. Sea $p(n)$ un polinomio de grado $k \geq 0$ con coeficientes en los números enteros. Demuestre que $p(n) \in O(n^k)$.

2. ¿Cuáles de las siguientes afirmaciones son ciertas?
   - $n^2 \in O(n)$
   - Si $f(n) \in O(n)$, entonces $f(n)^2 \in O(n^2)$
   - Si $f(n) \in O(n)$, entonces $2^{f(n)} \in O(2^n)$

3. Suponga que $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ existe y es igual a ℓ. Demuestre lo siguiente:
   - Si $\ell = 0$, entonces $f \in O(g)$ y $g \notin O(f)$
   - Si $\ell = \infty$, entonces $g \in O(f)$ y $f \notin O(g)$
   - Si $\ell \in \mathbb{R}^+$, entonces $f \in \Theta(g)$

4. Encuentre funciones f y g tales que $f \in \Theta(g)$ y $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ no existe

Outline

Búsqueda binaria

Suponga que tiene una lista ordenada (de menor a mayor) $L[1 \ldots n]$ de números enteros con $n \geq 1$

¿Cómo podemos verificar si un número a está en L?

Búsqueda binaria

El siguiente es un posible pseudo-código para el algoritmo de **búsqueda binaria**:

BúsquedaBinaria(a, L, i, j)
    if $i > j$ **then return** no
    else if $i = j$ **then**
        if $L[i] = a$ **then return** i
        else return no
    else
        $p := \lfloor \frac{i+j}{2} \rfloor$
        if $L[p] < a$ **then return BúsquedaBinaria**(a, L, $p+1$, j)
        else if $L[p] > a$ **then return BúsquedaBinaria**(a, L, i, $p-1$)
        else return p

Llamada inicial al algoritmo: **BúsquedaBinaria**(a, L, 1, n)

Tiempo de ejecución de búsqueda binaria

¿Cuál es la complejidad del algoritmo?

¿ Qué operaciones vamos a considerar ? ¿ Cuál es el peor caso ?

Si contamos sólo las **comparaciones**, entonces la siguiente expresión define la complejidad del algoritmo:

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

donde $c \in \mathbb{N}$ y $d \in \mathbb{N}$ son constantes tales que $c \geq 1$ y $d \geq 1$.

Esta es una **ecuación de recurrencia**.

¿Cómo podemos solucionar la ecuación anterior?

Solucionando una ecuación de recurrencia

Técnica básica: **sustitución de variables**.

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

Para la ecuación anterior usamos la sustitución $n = 2^k$

- Vamos a resolver la ecuación suponiendo que n es una potencia de 2.
- Vamos a utilizar inducción para demostrar que la solución obtenida nos da el orden del algoritmo.

Ecuaciones de recurrencia: sustitución de variables

Si realizamos la sustitución $n = 2^k$ en la ecuación:

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

obtenemos:

$$T(2^k) = \begin{cases} c & k = 0 \\ T(2^{k-1}) + d & k > 0 \end{cases}$$

Ecuaciones de recurrencia: sustitución de variables

Extendiendo la expresión anterior obtenemos:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + d \\ &= (T(2^{k-2}) + d) + d \\ &= T(2^{k-2}) + 2d \\ &= (T(2^{k-3}) + d) + 2d \\ &= T(2^{k-3}) + 3d \\ &= \cdots \end{aligned}$$

Deducimos la expresión general para $k - i \geq 0$:

$$T(2^k) = T(2^{k-i}) + i \cdot d$$

Ecuaciones de recurrencia: sustitución de variables

Considerando $i = k$ obtenemos:

$$\begin{aligned} T(2^k) &= T(1) + k \cdot d \\ &= c + k \cdot d \end{aligned}$$

Dado que $k = \log_2(n)$, obtenemos que

$$T(n) = c + d \cdot \log_2(n)$$

para n potencia de 2.

Usando **inducción** vamos a extender esta solución y demostrar que $T(n) \in O(\log_2(n))$.

Inducción constructiva

Sea $T(n)$ definida como:

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

Queremos demostrar que $T(n) \in O(\log_2(n))$

- Vale decir, queremos demostrar que existen $e \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $T(n) \leq e \cdot \log_2(n)$ para todo $n \geq n_0$

Inducción nos va servir tanto para demostrar la propiedad y como para determinar valores adecuados para e y n_0.

Nos referiremos a esta técnica como **inducción constructiva**.

Inducción constructiva

Dado que $T(1) = c$ y $\log_2(1) = 0$ no es posible encontrar una valor para e tal que $T(1) \leq e \cdot \log_2(1)$

Dado que $T(2) = (c + d)$, si consideramos $e = (c + d)$ tenemos que $T(2) \leq e \cdot \log_2(2)$

- Definimos entonces $e = (c + d)$ y $n_0 = 2$

Tenemos entonces que demostrar lo siguiente:

$$\forall n \geq 2.\ T(n) \leq e \cdot \log_2(n)$$

¿Cuál es el principio de inducción adecuado para este problema?

Inducción constructiva y fuerte

Hasta ahora:

- Tenemos n_0 como punto de partida.
- n_0 es un caso base, pero podemos tener otros.
- Dado $n > n_0$ tal que n no es un caso base, suponemos que la propiedad se cumple para todo $k \in \{n_0, \ldots, n-1\}$.

Inducción constructiva y fuerte

Queremos demostrar que $\forall n \geq 2.\ T(n) \leq e \cdot \log_2(n)$

- $n = 2$ es el punto de partida y el primer caso base
- $n = 3$ también es un caso base ya que $T(3) = T(1) + d$ y para $T(1)$ no se cumple la propiedad
- Para $n \geq 4$ tenemos que $T(n) = T(\lfloor \frac{n}{2} \rfloor) + d$ y $\lfloor \frac{n}{2} \rfloor \geq 2$, por lo que resolvemos este caso de manera inductiva
  - Suponemos que la propiedad se cumple para todo $k \in \{2, \ldots, n-1\}$

La demostración por inducción fuerte

Vamos a demostrar:

$$\forall n \geq 2.\ T(n) \leq e \cdot \log_2(n)$$

Demostración

Casos base:

$$\begin{array}{lcll} T(2) & = & c+d & = & e \cdot \log_2(2) \\ T(3) & = & c+d & < & e \cdot \log_2(3) \end{array}$$

Caso inductivo:

Suponemos que $n \geq 4$ y para todo $k \in \{2, \ldots, n-1\}$ se tiene que $T(k) \leq e \cdot \log_2(k)$

La demostración por inducción fuerte

Demostración

Usando la definición de $T(n)$ y la hipótesis de inducción concluimos que:

$$\begin{aligned}
T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + d \\
&\leq e \cdot \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + d \\
&\leq e \cdot \log_2\left(\frac{n}{2}\right) + d \\
&= e \cdot \log_2(n) - e \cdot \log_2(2) + d \\
&= e \cdot \log_2(n) - (c + d) + d \\
&= e \cdot \log_2(n) - c \\
&< e \cdot \log_2(n)
\end{aligned}$$

□