Análisis de la eficiencia de un algoritmo

IIC2283 – Diseño y Análisis de Algoritmos

Departmento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

2023-2

Una noción general de algoritmo

Sea $\mathcal{P}$ un problema a resolver, $\mathcal{I}_\mathcal{P}$ su conjunto de instancias (casos posibles del problema) y $\mathcal{S}_\mathcal{P}$ el conjunto de soluciones a las instancias

Vamos a pensar en un algoritmo $\mathcal{A}$ como una función $\mathcal{A} : \mathcal{I}_\mathcal{P} \to \mathcal{S}_\mathcal{P}$

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

Tiempo de ejecución de un algoritmo

A cada algoritmo $\mathcal{A}$ asociamos una función $\text{tiempo}_{\mathcal{A}} : \mathcal{I}_{\mathcal{P}} \to \mathbb{N}$ tal que:

$\text{tiempo}_{\mathcal{A}}(I)$: número de pasos realizados por $\mathcal{A}$ con entrada $I \in \mathcal{I}_{\mathcal{P}}$

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

Es importante definir el modelo de computación sobre el que vamos a trabajar

Modelos de Computación

Antes de diseñar y analizar algoritmos, es importante definir el *modelo de computación* subyacente.

Esto significa definir el tipo de computador hipotético sobre el cual se ejecutarán nuestros algoritmos.

Cada modelo de computación especifica las operaciones elementales que el computador hipotético puede ejecutar.

Esto afecta la manera en que uno diseña y analiza algoritmos sobre esos modelos, entonces es importante aclararlo desde un principio.

Dos modelos típicos: las Máquinas de Turing y las Máquinas de Acceso Aleatorio (RAM, por sus siglas en inglés).

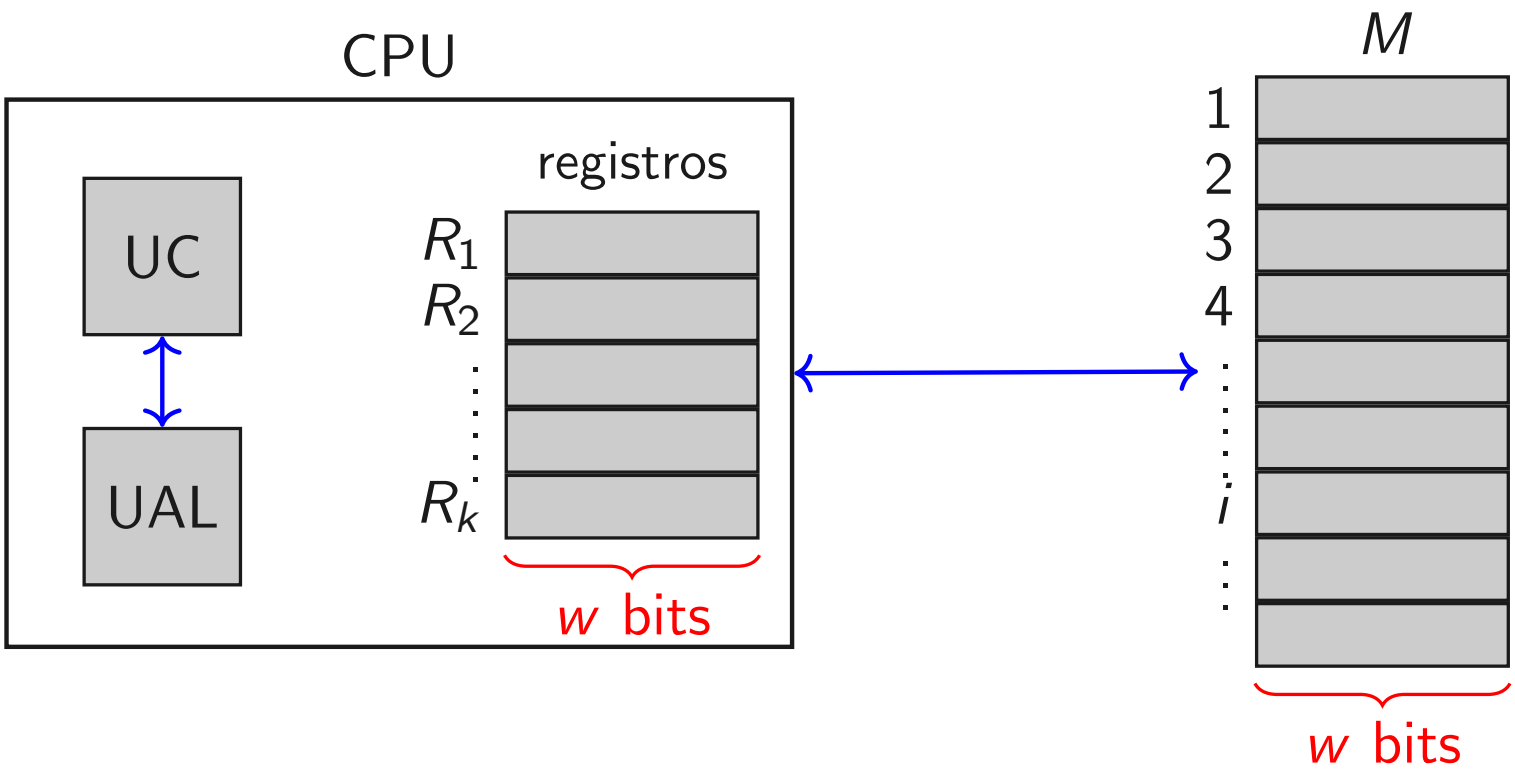
El Modelo de Computación Word RAM

Word RAM (simplemente RAM) es uno de los modelos más usados en la literatura por su similitud con los computadores actuales

Es una máquina hipotética que consiste de:

- **CPU**: la cual está formada por
  - **Registros**: k celdas de w bits cada una (el parámetro más importante del modelo)
  - **Instrucciones**: un conjunto de instrucciones que la CPU es capaz de ejecutar ($\leftarrow$, $+$, $\times$, $-$, $/$, **and**, **or**, **not**, $=$, $\leq$, $\geq$, $<$, $>$, $\neq$, saltos implícitos o explícitos, y ops. a nivel de bits).
  - **Unidad de control**: ejecuta las instrucciones de un algoritmo por *pasos* o *ciclos del procesador* (una instrucción por ciclo)
- **Memoria**: una cantidad infinita de celdas de w bits cada una, $M[1]$, $M[2]$, ..., que pueden ser accedidas de manera aleatoria (i.e., en el orden que un algoritmo necesite) para lectura y escritura y cada acceso toma un ciclo del procesador

El Modelo de Computación Word RAM



Recordatorio:

A cada algoritmo $\mathcal{A}$ asociamos una función $\text{tiempo}_{\mathcal{A}} : \mathcal{I}_{\mathcal{P}} \to \mathbb{N}$ tal que:

$\text{tiempo}_{\mathcal{A}}(w)$: número de pasos realizados por $\mathcal{A}$ con entrada $w \in \mathcal{I}_{\mathcal{P}}$

Tiempo de ejecución de un algoritmo

Tiempo de Ejecución

Sea $\mathcal{A}$ un algoritmo que resuelve un problema abstracto $\mathcal{P}$, y sea $I \in \mathcal{I}_{\mathcal{P}}$ una instancia particular de $\mathcal{P}$. El *tiempo de ejecución* del algoritmo $\mathcal{A}$ para la instancia I, denotado $\text{tiempo}_{\mathcal{A}}(I)$, es la cantidad de instrucciones que ejecuta $\mathcal{A}$ para resolver I.

El tiempo de ejecución es clave para comparar algoritmos que resuelven un mismo problema

Medir el tiempo en base a la cantidad de operaciones elementales permite independizarnos de las implementaciones particulares de un algoritmo

Tiempo de ejecución de un algoritmo

Algoritmo 1: MÁXIMO($S[1..n]$)

```
1 m ← 1
2 for i ← 2 to n do
3 |   if S[m] ≤ S[i] then
4 |   |   m ← i
5 |   end
6 end
7 return m
```

Este algoritmo realiza $n - 1$ comparaciones entre elementos de S para encontrar el máximo

Análisis de Mejor y Peor Caso

Es común que el tiempo de ejecución de un algoritmo dependa de la instancia que se está resolviendo

Por ejemplo:

Algoritmo 2: PERTENENCIA(x, $S[1..n]$)

Entrada: Un conjunto no necesariamente ordenado representado por un arreglo $S[1..n]$, y un elemento x.
Salida: la posición i tal que $S[i] = x$, o $n+1$ si $x \notin S$.

```
1 for i ← 1 to n do
2 |   if x = S[i] then
3 |   |   return i
4 |   end
5 end
6 return n + 1                                  // Indica que x ∉ S
```

Tiempo de ejecución de un algoritmo en el mejor caso

Tiempo de Ejecución de Mejor Caso

Sea $\mathcal{A}$ un algoritmo que resuelve un problema abstracto $\mathcal{P}$. El tiempo de ejecución de mejor caso para $\mathcal{A}$ sobre todas las instancias de tamaño n se define como:

$$t_{\mathcal{A}}(n) \equiv \text{mín}\,\{\text{tiempo}_{\mathcal{A}}(I) \mid I \in \mathcal{I}_{\mathcal{P}},\ |I| = n\}.$$

- El mejor caso del algoritmo anterior consiste en buscar el elemento $x = S[1]$ (ejecuta una cantidad constante de instrucciones)
- Este tipo de análisis es optimista y no suele aportar demasiada información
- Al analizar algoritmos, siempre es conveniente ser "relativamente" pesimista

Tiempo de ejecución de un algoritmo en el peor caso

Tiempo de Ejecución de Peor Caso

Sea $\mathcal{A}$ un algoritmo que resuelve el problema $\mathcal{P}$. El tiempo de ejecución de peor caso para $\mathcal{A}$ sobre todas las instancias de tamaño n se define como:

$$T_{\mathcal{A}}(n) \equiv \text{máx}\{\text{tiempo}_{\mathcal{A}}(I) \mid I \in \mathcal{I}_{\mathcal{P}},\ |I| = n\}.$$

- El peor caso del algoritmo anterior es buscar $x = S[n]$ (o $x \notin S$), ejecutando n ciclos
- Este tipo de análisis es pesimista
  - Previene al usuario sobre el peor escenario y permite tomar recaudos
  - Puede penalizar demasiado a ciertos algoritmos con peores casos cuya probabilidad de ocurrencia es baja

Notaciones asintóticas

Es importante distinguir la complejidad de mejor y peor caso de un algoritmo

Sin embargo, a veces es preferible denotar el tiempo de ejecución de un algoritmo de tal manera que se incluyan todas las instancias del problema

Las notaciones asintóticas permiten **acotar** el tiempo de ejecución de todas las instancias

Permiten clasificar funciones que modelan la complejidad de un algoritmo para compararlas

Estudiamos a continuación el concepto de cota asintótica

Notación asintótica

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 $3n^2 + 17n + 22$

Vamos a desarrollar notación para hablar del orden de un algoritmo.

Notación asintótica

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 $O(f)$

Sea $f : \mathbb{N} \to \mathbb{R}_0^+$

Definición

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

Decimos entonces que $g \in O(f)$

- También usamos la notación g es $O(f)$, lo cual es formalizado como $g \in O(f)$
- $O(f)$ son todas las funciones g que crecen más lentamente o igual que f
- Para algoritmo $\mathcal{A}$ se tiene que el tiempo de ejecución de cualquier instancia es $O(T_{\mathcal{A}}(n))$

Ejercicio

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

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

Definición

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

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

- Si para un algoritmo $\mathcal{A}$ se tiene que $t_{\mathcal{A}}(n) \in \Theta(T_{\mathcal{A}}(n))$, entonces el tiempo de ejecución de cualquier instancia es $\Theta(T_{\mathcal{A}}(n))$
- Siempre que se pueda, usar Θ en lugar de O para acotar el tiempo de ejecución de un algoritmo

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 $n \geq n_0$: $c \cdot f(n) \leq g(n) \leq d \cdot f(n)$

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

Ejemplo:

$$O(n) = \{1, 2, \ldots, \lg\lg n, \ldots, \lg n, \ldots, \sqrt{n}, n^{1/3}, \ldots, \underbrace{n, n+1, 2n, \ldots}_{\Theta(n)}\}$$

$$\Omega(n) = \{\underbrace{n, n+1, 2n, \ldots}_{\Theta(n)}, n\lg n, \ldots, n\sqrt{n}, \ldots, n^2, \ldots, 2^n, \ldots, n!, n^n, \ldots\}$$

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

Ejercicios

Para el siguiente algoritmo clásico para ordenar una lista L (de menor a mayor, asumiendo una relación de orden total sobre los elementos de L).

InsertionSort($L[1 \ldots n]$: lista de elementos)
    for $i := 2$ **to** n **do**
        $j := i - 1$
        while $j \geq 1$ **and** $L[j] > L[j+1]$ **do**
            $aux := L[j]$
            $L[j] := L[j+1]$
            $L[j+1] := aux$
            $j := j - 1$
    return L

1. Encuentre el tiempo de ejecución de mejor y peor caso
2. Acote asintóticamente el tiempo de ejecución para todas las posibles entradas de n elementos que puede recibir el algoritmo (use la notación adecuada)

Ecuaciones de recurrencia

Aparecen naturalmente al analizar algoritmos recursivos, por ejemplo:

Algoritmo 3: $\textsc{Fib}(n)$

```
1 if n ≤ 1 then
2 |   return n
3 else
4 |   return Fib(n − 1) + Fib(n − 2)
5 end
```

El tiempo de ejecución de $\textsc{Fib}(n)$ puede expresarse como la ecuación de recurrencia:

$$T(n) = \begin{cases} T(n-1) + T(n-2) + b, & n \geq 2; \\ a, & n = 0; \\ a, & n = 1. \end{cases}$$

para constantes $a \in \mathbb{N}$, $b \in \mathbb{N}$, ambas > 0.

Ecuaciones de recurrencia

A pesar de que $T(n)$ puede definirse de forma simple y directa desde el algoritmo, no sabemos mucho respecto de la función que representa

Nos gustaría encontrar la forma cerrada de T para poder estudiarla (e.g., entender su velocidad de crecimiento)

Para el caso $\textsc{Fib}(n)$, ¿Cuál es la función que acota a $T(n)$?

A continuación demostraremos que $T(n) \in O(2^n)$

Eso significa que hay que demostrar que $\exists c \in \mathbb{R}^+$, $\exists n_0 \in \mathbb{N}$, tal que

$$T(n) \leq c \cdot 2^n, \ \forall n \geq n_0.$$

Vamos a demostrarlo por inducción

Ecuaciones de recurrencia

Casos base: Primero chequeamos que la propiedad se cumpla para los casos base

- Para $n = 0$, $T(0) = a$, entonces se debe cumplir $a \leq c \cdot 2^0$, lo cual es cierto para $c \geq a$.
- Para $n = 1$, $T(1) = a$, entonces se debe cumplir $a \leq c \cdot 2^1$, lo cual es cierto para $c \geq a/2$.

Esto significa que la propiedad es cierta para los casos base

Ecuaciones de recurrencia

Hipótesis inductiva: Asumimos $T(n') \leq c \cdot 2^{n'}, \ \forall n' \in \{0, 1, \ldots, n-1\}$

Estamos usando inducción fuerte, por lo que es necesario chequear que la propiedad se cumple para todos los casos base

Paso inductivo: Sea $n \geq 2$. Entonces $T(n)$ corresponde al caso recurrente:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + b \\ &\leq c \cdot 2^{n-1} + c \cdot 2^{n-2} + b \\ &< c \cdot 2^{n-1} + c \cdot 2^{n-1} + b \\ &= c \cdot 2^{n} + b \end{aligned}$$

Esto prueba la propiedad para todo $n \geq 0$ (b es una constante, por lo que no influye en el resultado)

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

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:

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

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

Solucionando una ecuación de recurrencia

¿Cómo podemos solucionar una ecuación de recurrencia?

- Técnica básica: sustitución de variables

Para la ecuación anterior usamos la sustitución $n = 2^k$, por lo que $k = \log_2(n)$

- 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:

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

obtenemos:

$$T(2^k) = \begin{cases} b, & k = 0 \quad [2^k = 1] \\ T(2^{k-1}) + d, & k > 0 \quad [2^k > 1] \end{cases}$$

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:

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

Dado que $k = \log_2(n)$, obtenemos que $T(n) = b + d \cdot \log_2(n)$ para n potencia de 2

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

Inducción constructiva

Sea $T(n)$ definida como:

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

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

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

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

- Por esto usamos el término inducción constructiva

Inducción constructiva

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

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

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

Tenemos entonces que demostrar lo siguiente:

$$\forall n \geq 2,\ T(n) \leq c \cdot \log_2(n)$$

Inducción constructiva y fuerte

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

- 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 $n' \in \{n_0, \ldots, n-1\}$

Inducción constructiva y fuerte

Queremos demostrar que $\forall n \geq 2,\ T(n) \leq c \cdot \log_2(n)$

- 2 es el punto de partida y el primer caso base
- También 3 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 $n' \in \{2, \ldots, n-1\}$

La demostración por inducción fuerte

Casos base:

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

Caso inductivo:

Suponemos que $n \geq 4$ y para todo $n' \in \{2, \ldots, n-1\}$ se tiene que $T(n') \leq c \cdot \log_2(n')$

La demostración por inducción fuerte

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

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

Un segundo ejemplo de inducción constructiva

Considere la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 0 & n = 0 \\ n^2 + n \cdot T(n-1) & n > 0 \end{cases}$$

Queremos determinar una función $f(n)$ para la cual se tiene que $T(n) \in O(f(n))$

- ¿Alguna conjetura sobre quién podría ser $f(n)$?

Una posible solución para la ecuación de recurrencia

Dada la forma de la ecuación de recurrencia, podríamos intentar primero con $f(n) = n!$

Tenemos entonces que determinar $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $T(n) \leq c \cdot n!$ para todo $n \geq n_0$

- Pero nos vamos a encontrar con un problema al tratar de usar la hipótesis de inducción

Una posible solución para la ecuación de recurrencia

Supongamos que la propiedad se cumple para n:

$$T(n) \leq c \cdot n!$$

Tenemos que:

$$\begin{aligned} T(n+1) &= (n+1)^2 + (n+1) \cdot T(n) \\ &\leq (n+1)^2 + (n+1) \cdot (c \cdot n!) \\ &= (n+1)^2 + c \cdot (n+1)! \end{aligned}$$

Pero no existe una constante c para la cual $(n+1)^2 + c \cdot (n+1)! \leq c \cdot (n+1)!$

- Dado que $n \in \mathbb{N}$

¿Cómo solucionamos el problema con la demostración?

Una demostración por inducción puede hacerse más simple considerando una propiedad más fuerte.

- Dado que la hipótesis de inducción se va a volver más fuerte

Vamos a seguir tratando de demostrar que $T(n) \in O(n!)$ pero ahora considerando una propiedad más fuerte.

Vamos a demostrar lo siguiente:

$$(\exists c \in \mathbb{R}^+)(\exists d \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(T(n) \leq c \cdot n! - d \cdot n)$$

Inducción constructiva sobre una propiedad más fuerte

Para tener una mejor idea de los posible valores para c, d y n_0 vamos a considerar primero el paso inductivo en la demostración.

Supongamos que la propiedad se cumple para n:

$$T(n) \quad \leq \quad c \cdot n! - d \cdot n$$

Tenemos que:

$$\begin{aligned} T(n+1) &= (n+1)^2 + (n+1) \cdot T(n) \\ &\leq (n+1)^2 + (n+1) \cdot (c \cdot n! - d \cdot n) \\ &= c \cdot (n+1)! + (n+1)^2 - d \cdot n \cdot (n+1) \\ &= c \cdot (n+1)! + ((n+1) - d \cdot n) \cdot (n+1) \end{aligned}$$

Inducción constructiva sobre una propiedad más fuerte

Para poder demostrar que la propiedad se cumple para $n+1$ necesitamos que lo siguiente sea cierto:

$$(n+1) - d \cdot n \quad \leq \quad -d$$

De lo cual concluimos la siguiente restricción para d:

$$\frac{(n+1)}{(n-1)} \quad \leq \quad d$$

Si consideramos $n \geq 2$ concluimos que $d \geq 3$

- Consideramos entonces $n_0 = 2$ y $d = 3$

Inducción constructiva sobre una propiedad más fuerte

Para concluir la demostración debemos considerar el caso base $n_0 = 2$

Tenemos que:

$$\begin{array}{lcll} T(0) & = & 0 & \\ T(1) & = & 1^2 + 1 \cdot T(0) & = & 1 \\ T(2) & = & 2^2 + 2 \cdot T(1) & = & 6 \end{array}$$

Entonces se debe cumplir que $T(2) \leq c \cdot 2! - 3 \cdot 2$, vale decir,

$$6 \quad \leq \quad c \cdot 2 - 6$$

Concluimos que $c \geq 6$, por lo que consideramos $c = 6$

- Tenemos entonces que $(\forall n \geq 2)(T(n) \leq 6 \cdot n! - 3 \cdot n)$, de lo cual concluimos que $T(n) \in O(n!)$

El Teorema Maestro

Muchas de las ecuaciones de recurrencia que vamos a usar en este curso tienen la siguiente forma:

$$T(n) = \begin{cases} c & n = 0 \\ a \cdot T(\lfloor \frac{n}{b} \rfloor) + f(n) & n \geq 1 \end{cases}$$

donde a, b y c son constantes, y $f(n)$ es una función arbitraria.

El Teorema Maestro nos dice cuál es el orden de $T(n)$ dependiendo de ciertas condiciones sobre a, b y $f(n)$

El Teorema Maestro

El Teorema Maestro también se puede utilizar cuando $\lfloor \frac{n}{b} \rfloor$ es reemplazado por $\lceil \frac{n}{b} \rceil$

Antes de dar el enunciado del Teorema Maestro necesitamos definir una condición de regularidad sobre la función $f(n)$

Una condición de regularidad sobre funciones

Dado: función $f : \mathbb{N} \to \mathbb{R}_0^+$ y constantes $a, b \in \mathbb{R}$ tales que $a \geq 1$ y $b > 1$

Definición

f es (a, b)-regular si existen constantes $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $c < 1$ y

$$(\forall n \geq n_0)(a \cdot f(\lfloor \frac{n}{b} \rfloor) \leq c \cdot f(n))$$

Ejercicio

1. Demuestre que las funciones n, n^2 y 2^n son (a, b)-regulares si $a < b$.
2. Demuestre que la función $\log_2(n)$ no es (1,2)-regular.

Una solución al segundo problema

Por contradicción, supongamos que $\log_2(n)$ es (1,2)-regular.

Entonces existen constantes $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $c < 1$ y

$$(\forall n \geq n_0)(\log_2(\lfloor \frac{n}{2} \rfloor) \leq c \cdot \log_2(n))$$

De esto concluimos que:

$$(\forall k \geq n_0)(\log_2(\lfloor \frac{2 \cdot k}{2} \rfloor) \leq c \cdot \log_2(2 \cdot k))$$

Una solución al segundo problema

Vale decir:

$$(\forall k \geq n_0)(\log_2(k) \leq c \cdot (\log_2(k) + 1))$$

Dado que $0 < c < 1$, concluimos que:

$$(\forall k \geq n_0)(\log_2(k) \leq \frac{c}{1-c})$$

Lo cual nos lleva a una contradicción. □

El enunciado del Teorema Maestro

Teorema

Sea $f : \mathbb{N} \to \mathbb{R}_0^+$, $a, b, c \in \mathbb{R}_0^+$ *tales que* $a \geq 1$ *y* $b > 1$, *y* $T(n)$ *una función definida por la siguiente ecuación de recurrencia:*

$$T(n) = \begin{cases} c & n = 0 \\ a \cdot T(\lfloor \frac{n}{b} \rfloor) + f(n) & n \geq 1 \end{cases}$$

Se tiene que:

1. *Si* $f(n) \in O(n^{\log_b(a)-\varepsilon})$ *para* $\varepsilon > 0$, *entonces* $T(n) \in \Theta(n^{\log_b(a)})$
2. *Si* $f(n) \in \Theta(n^{\log_b(a)})$, *entonces* $T(n) \in \Theta(n^{\log_b(a)} \cdot \log_2(n))$
3. *Si* $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$ *para* $\varepsilon > 0$ *y* f *es* (a, b)*-regular, entonces* $T(n) \in \Theta(f(n))$

Usando el Teorema Maestro

Considere la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & n = 0 \\ 3 \cdot T(\lfloor \frac{n}{2} \rfloor) + c \cdot n & n \geq 1 \end{cases}$$

Dado que $\log_2(3) > 1.5$, tenemos que $\log_2(3) - 0.5 > 1$

Deducimos que $c \cdot n \in O(n^{\log_2(3)-0.5})$, por lo que usando el Teorema Maestro concluimos que $T(n) \in \Theta(n^{\log_2(3)})$

El Teorema Maestro y la función $\lceil x \rceil$

Suponga que cambiamos $\lfloor \frac{n}{b} \rfloor$ por $\lceil \frac{n}{b} \rceil$ en la definición de (a, b)-regularidad.

Entonces el Teorema Maestro sigue siendo válido pero con $T(\lfloor \frac{n}{b} \rfloor) + f(n)$ reemplazado por $T(\lceil \frac{n}{b} \rceil) + f(n)$

Analizando la complejidad de un algoritmo

Sea $\mathcal{A} : \Sigma^* \to \Sigma^*$ un algoritmo

- Recuerde que $t_{\mathcal{A}}(n)$ es el mayor número de pasos realizados por $\mathcal{A}$ sobre las entradas $w \in \Sigma^*$ de largo n

Definición (Complejidad en el peor caso)

Decimos que $\mathcal{A}$ en el peor caso es $O(f(n))$ si $t_{\mathcal{A}}(n) \in O(f(n))$

Analizando la complejidad de un algoritmo

Notación

Las definición de peor caso puede ser modificada para considerar las notaciones Θ y Ω

- Simplemente reemplazando $O(f(n))$ por $\Theta(f(n))$ u $\Omega(f(n))$, respectivamente

Por ejemplo, decimos que $\mathcal{A}$ en peor caso es $\Omega(f(n))$ si $t_{\mathcal{A}}(n) \in \Omega(f(n))$