

Complejidad computacional (1/2)

Clase 22

IIC 1253

Pedro Bahamondes

Outline

Introducción

Complejidad de algoritmos iterativos

Ecuaciones de recurrencia

Epílogo

Algoritmos

El análisis de algoritmos consta de dos partes

- 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.

Hoy estudiaremos cómo medir el segundo punto,
y lo aplicaremos a algoritmos iterativos y recursivos



Objetivos de la clase

- Estudiar la complejidad de algoritmos iterativos
- Aplicar notación asintótica a algoritmos iterativos
- Definir ecuaciones de recurrencia en algoritmos recursivos
- Estudiar la complejidad de algoritmos recursivos

Outline

Introducción

Complejidad de algoritmos iterativos

Ecuaciones de recurrencia

Epílogo

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.

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.

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.

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  $x \leftarrow 0$   
2 for  $i = 1$  to  $n$  do  
3   for  $j = 1$  to  $i$  do  
4      $x \leftarrow x + 1$ 
```

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

Solución: Apuntes Jorge Pérez, Sección 3.1.3, páginas 104 y 105.

Contando instrucciones

Ejercicio

Considere el siguiente trozo de código:

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

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

Solución: Apuntes Jorge Pérez, Sección 3.1.3, página 105.

Algoritmos iterativos

Consideremos el siguiente algoritmo de búsqueda en arreglos:

input : arreglo de enteros $A = [a_0, \dots, a_{n-1}]$, un natural $n > 0$
correspondiente al largo del arreglo y un entero k

output: índice de k en A , -1 si no está.

Búsqueda(A, n, k):

```
1   for  $i = 0$  to  $n - 1$  do  
2       if  $a_i = k$  then  
3           return  $i$   
4   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?).

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 sí lo sería, y más específicamente la comparación.

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 sí 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 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:

input : arreglo $A = [a_0, \dots, a_{n-1}]$ y su largo $n > 0$

output: arreglo está ordenado al terminar el algoritmo.

InsertionSort(A, n):

```
1   for  $i = 1$  to  $n - 1$  do
2        $j \leftarrow i$ 
3       while  $a_{j-1} > a_j \wedge j > 0$  do
4            $t \leftarrow a_{j-1}$ 
5            $a_{j-1} \leftarrow a_j$ 
6            $a_j \leftarrow t$ 
7            $j \leftarrow j - 1$ 
```

Solución: Apuntes Jorge Pérez, Sección 3.1.3, página 106.

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.

Outline

Introducción

Complejidad de algoritmos iterativos

Ecuaciones de recurrencia

Epílogo

Algoritmos recursivos

Algoritmos recursivos

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

Algoritmos recursivos

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

- Buscamos alguna(s) instrucción(es) representativa.

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.

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?

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?

Ahora tenemos que considerar llamados recursivos

Algoritmos recursivos: un ejemplo

Algoritmos recursivos: un ejemplo

input : Arreglo ordenado $A[0, \dots, n-1]$, elemento x , índices i, f

output: Índice $m \in \{0, \dots, n-1\}$ tq $A[m] = x$ si x está en A , o -1

BinarySearch($A, x, i = 0, f = n-1$):

```
1  if  $i > f$  then
2      return  $-1$ 
3  else if  $i = f$  then
4      if  $A[i] = x$  then
5          return  $i$ 
6      else
7          return  $-1$ 
8  else
9       $m \leftarrow \lfloor (i + f)/2 \rfloor$ 
10     if  $A[m] < x$  then
11         return BinarySearch( $A, x, m + 1, f$ )
12     else if  $A[m] > x$  then
13         return BinarySearch( $A, x, i, m - 1$ )
14     else
15         if  $A[m] = x$  then return  $m$ 
```

Algoritmos recursivos: un ejemplo

- ¿Qué operaciones contamos?

Algoritmos recursivos: un ejemplo

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

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 `BinarySearch` en el peor caso, en función del tamaño del arreglo.

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 `BinarySearch` en el peor caso, en función del tamaño del arreglo.

Respuesta:

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

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 `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: 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 `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**.

¿Cómo obtenemos una fórmula explícita?

Algoritmos recursivos: un ejemplo

Ejercicio

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

Contaremos las comparaciones. Dividiremos el análisis del peor caso:

- Si el arreglo tiene largo 1, entramos en la instrucción 3 y luego hay una comparación $\Rightarrow T(n) = 3$, con $n = 1$.
- Si el arreglo tiene largo mayor a 1, el peor caso es entrar en el `else` de 8 y luego en la segunda llamada recursiva. En tal caso, se hacen las comparaciones de las líneas 1, 3, 10, 12 a lo que sumamos las comparaciones que haga la llamada recursiva, que serán $T(\lfloor \frac{n}{2} \rfloor)$.

Entonces, nuestra función $T(n)$ será:

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

Algoritmos recursivos: ecuaciones de recurrencia

Necesitamos *resolver* esta 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 .

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.

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?

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?

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

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 $n = 2^k$.

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.

Algoritmos recursivos: ecuaciones de recurrencia

Ejercicio

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

$$T(2^k) = \begin{cases} 3 & k = 0 \\ T(2^{k-1}) + 4 & k > 0 \end{cases}$$

Expandiendo el caso recursivo:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 4 \\ &= (T(2^{k-2}) + 4) + 4 \\ &= T(2^{k-2}) + 8 \\ &= (T(2^{k-3}) + 4) + 8 \\ &= T(2^{k-3}) + 12 \\ &\vdots \end{aligned}$$

Algoritmos recursivos: ecuaciones de recurrencia

Ejercicio

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

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

$$T(2^k) = T(2^{k-i}) + 4i$$

Tomamos $i = k$:

$$T(2^k) = T(1) + 4k = 3 + 4k$$

Como $k = \log_2(n)$:

$$T(n) = 4 \cdot \log_2(n) + 3, \text{ con } n \text{ potencia de } 2$$

Algoritmos recursivos: ecuaciones de recurrencia

Ejercicio

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

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

$$T(2^k) = T(2^{k-i}) + 4i$$

Tomamos $i = k$:

$$T(2^k) = T(1) + 4k = 3 + 4k$$

Como $k = \log_2(n)$:

$$T(n) = 4 \cdot \log_2(n) + 3, \text{ con } n \text{ potencia de } 2$$

Problema: esto solo es válido cuando $n = 2^k$

Notación asintótica condicional

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

Definición

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

Notación asintótica condicional

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

Definición

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

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

Estamos restringiendo a un tipo de n particular

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 ?

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:

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

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 $T \in \Theta(\log_2(n)) \dots$

Usaremos inducción

Generalización de soluciones usando inducción

Para el ejemplo anterior:

Ejercicio

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

Generalización de soluciones usando inducción

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?

Generalización de soluciones usando inducción

Ejercicio

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

Veamos los primeros valores de $T(n)$ para estimar c y n_0 :

$$T(1) = 3$$

$$T(2) = T(1) + 4 = 7$$

$$T(3) = T(1) + 4 = 7$$

$$T(4) = T(2) + 4 = 11$$

Podríamos tomar $c = 7$ y $n_0 = 2$, pues con $n = 1$:

$$T(1) = 3 \not\leq 7 \cdot \log_2(1) = 0$$

y con $n = 2$

$$T(2) = 7 \leq 7 \cdot \log_2(2) = 7$$

Generalización de soluciones usando inducción

Ejercicio

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

Veamos los primeros valores de $T(n)$ para estimar c y n_0 :

$$T(1) = 3$$

$$T(2) = T(1) + 4 = 7$$

$$T(3) = T(1) + 4 = 7$$

$$T(4) = T(2) + 4 = 11$$

Podríamos tomar $c = 7$ y $n_0 = 2$, pues con $n = 1$:

$$T(1) = 3 \not\leq 7 \cdot \log_2(1) = 0$$

y con $n = 2$

$$T(2) = 7 \leq 7 \cdot \log_2(2) = 7$$

La intuición nos dice $n_0 = 2$ y $c = 7$. . . lo demostraremos

Generalización de soluciones usando inducción

Ejercicio

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

PD: $\forall n \geq 2, T(n) \leq 7 \cdot \log_2(n)$. Por inducción fuerte:

BI: Además de $n = 2$, debemos mostrar la base para $n = 3$, puesto que depende de $T(1)$ que no está incluido en el resultado que estamos mostrando.

$$T(2) = 7 = 7 \cdot \log_2(2)$$

$$T(3) = 7 < 7 \cdot \log_2(3) \text{ pues el logaritmo es creciente}$$

HI: Supongamos que con $n \geq 4, \forall k \in \{2, \dots, n-1\}$ se cumple que $T(k) \leq 7 \cdot \log_2(k)$.

Generalización de soluciones usando inducción

Ejercicio

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

TI: Como $n \geq 4$:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4 && / \text{ HI} \\ &\leq 7 \cdot \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4 && / \log \text{ es creciente, sacamos el piso} \\ &\leq 7 \cdot \log_2\left(\frac{n}{2}\right) + 4 && / \log \text{ de división} \\ &= 7(\log_2(n) - \log_2(2)) + 4 \\ &= 7 \cdot \log_2(n) - 7 + 4 \\ &= 7 \cdot \log_2(n) - 3 \\ &< 7 \cdot \log_2(n) \square \end{aligned}$$

Outline

Introducción

Complejidad de algoritmos iterativos

Ecuaciones de recurrencia

Epílogo

Objetivos de la clase

- Estudiar la complejidad de algoritmos iterativos
- Aplicar notación asintótica a algoritmos iterativos
- Definir ecuaciones de recurrencia en algoritmos recursivos
- Estudiar la complejidad de algoritmos recursivos