AA2021a-03-Analisis-Algoritmos 2

Abdiel E. Cáceres González

20 de febrero de 2021

Índice

Código de Honor

La UJAT espera que sus estudiantes muestren respeto por el orden, la moral y el honor de sus compañeros, sus maestros y su persona. Por ello, se establece este Código de Honor, con el propósito de guiar la vida escolar de los alumnos. Estos lineamientos no permiten todo acto que deshonre los ámbitos académicos. Las siguientes acciones son consideradas como violatorias al Código de Honor de:

1. Usar, proporcionar o recibir apoyo o ayuda no autorizada al presentar exámenes y al elaborar o presentar reportes, tareas y en general en cualquier otra forma por la que el maestro evalúe el desempeño académico del estudiante.
2. Presentar trabajos o exámenes por otra persona, o permitir a otro que lo haga por uno mismo.

Las sanciones podrán ser desde la reprobación con calificación 0 (cero) en la tarea por evaluar, hasta una calificación reprobatoria de 0 (cero) en la materia.

PARTE DOS
Análisis de algoritmos

3. Iteraciones

3.1. Ciclos simples

A veces el tiempo de ejecución es proporcional al número de veces que se ejecuta un ciclo `while` u otros constructores similares. De esto se sigue que el número de iteraciones es un buen indicativo del tiempo de ejecución de un algoritmo. Este es el caso para muchos algoritmos incluyendo algunos de búsqueda, ordenación, multiplicación de matrices y otros.

Se puede contar el número de iteraciones viendo una o más instrucciones [ordenes, comandos o sentencias] en el algoritmo que se ejecutan, y luego haciendo una estimación del número de veces que se ejecutan. Suponiendo que el costo de cada instrucción es constante, el costo estimado es asintóticamente proporcional al costo total del algoritmo y se puede expresar en términos de alguna notación de complejidad.

Un modo de calcular el número de iteraciones es asociar el ciclo con una fórmula de suma. La variable iteradora del ciclo se puede llamar «simple» si tiene incrementos unitarios. Un ciclo también se llama «simple» si su variable iteradora es simple. En su forma más simple un ciclo `for` luce como:

```
1  cuenta = 1
2  for i = in range(inf,sup)
3    cuenta = cuenta + 1
4  end
```

que se puede asociar a la suma

$$\texttt{cuenta} = \sum_{i=\texttt{inf}}^{\texttt{sup}} 1.$$

Entonces un ciclo simple se puede asociar a una suma haciendo lo siguiente:

1. Usar la variable iteradora en el ciclo como el índice de la suma
2. Usar el valor inicial de la variable iteradora como el límite inferior de la suma, y el último valor del iterador como el límite superior de la suma. Considerando que el límite superior no se considera como parte del ciclo. En el ciclo de ejemplo, la línea `cuenta = cuenta + 1`, se ejecuta $\texttt{sup} - \texttt{inf}$ veces.
3. Cada ciclo anidado se asocia con una suma anidada.

Tomemos por ejemplo un algoritmo que llamaremos `Iter1` y que sirve para coleccionar todas las sumas de cada cuadrado perfecto desde `1` hasta n, donde n es un número cuadrado perfecto [un número entero cuyas raices cuadradas también son números enteros]. Es decir, que el algoritmo `Iter1` calcula $\sum_{i=1}^{j} i$, haciendo variar j desde 1 hasta n. Aunque hay maneras más eficientes de hacer el cálculo, el algoritmo `Iter1` tal como está, servirá para ilustrar el modo de contar las iteraciones en algoritmos con ciclos simple.

```
PROBLEMA: Colecciona el resultado de todas las sumas de los numeros enteros desde 1 hasta el cuadrado
     de j [inclusive], para j desde 1 hasta un la raiz cuadrada de n [inclusive].
ENTRADA : n = k^2 ; para algun entero k
SALIDA  : lista de numero_entero
```

```
1  Iter1(n: entero):
2    k ⟵ √n
3    sumas ⟵ []
4    for j desde 1 hasta k + 1
5      s ⟵ 0
6      for i desde 1 hasta j^2 + 1
7        s ⟵ s + i
8      sumas_j ⟵ suma
9    return sumas
```

```
import math

def Iter1(n):
    """
    Colecciona el resultado de todas las sumas de los numeros enteros desde 1 hasta el cuadrado de j,
        para j desde 1 hasta un la raiz cuadrada de n.
    ENTRADA
        n : número entero tal que sqrt(n) es entero
    """
    k = int(math.sqrt(n))
    sumas = []
    for j in range(1,k+1):
        s = 0
        for i in range(1,j**2+1):
            s = s + i
        sumas.append(s)
    return sumas

x = int(input())
print(Iter1(x))
```

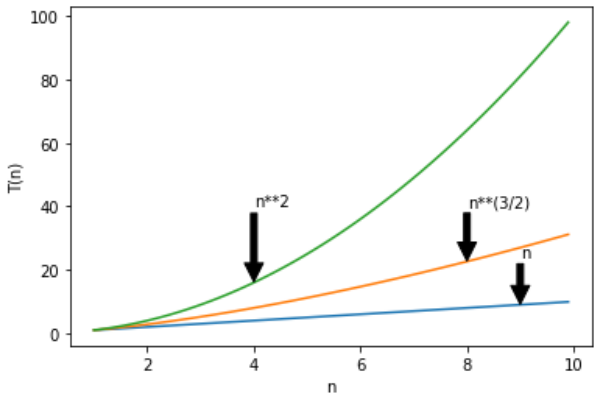
Antes de hacer el análisis, asegurémonos de entender cómo funciona el algoritmo. Así que hagamos algunas corridas con diferentes valores:

entrada:			salida:
`x`	`n`	`k`	`Iter1(x)`$\hookleftarrow$`sumas`
1	1	1	`[1]`
4	4	2	`[1,10]`
9	9	3	`[1,10,45]`
16	16	4	`[1,10,45,136]`
25	25	5	`[1,10,45,136,325]`
36	36	6	`[1,10,45,136,325,666]`
49	49	7	`[1,10,45,136,325,666,1225]`

Por ahora supongamos que $\sqrt{n}$ se puede calcular en un tiempo constante, es decir es de complejidad $\mathbf{\Theta}\left(1\right)$. Obviamente el costo del algoritmo está dominado por el número de veces que se ejecuta la línea 7 del algoritmo. Como tenemos dos ciclos simples anidados, inmediatamente podemos relacionarlos con una doble suma

$$\begin{aligned}
\sum_{j=1}^{k+1}\sum_{i=1}^{j^2+1} 1 &= \sum_{j=1}^{k+1}(j^2+1)-1 \\
&= \sum_{j=1}^{k+1} j^2 \\
&= \frac{(k+1)(k+1+1)(2(k+1)+1)}{6} \\
&= \frac{(k+1)(k+2)(2k+3)}{6} \\
&= \frac{2k^3+9k^2+13k+6}{6} \\
&= \mathbf{\Theta}\left(k^3\right) \\
&= \mathbf{\Theta}\left(n\sqrt{n}\right)
\end{aligned}$$

Así el tiempo de ejecución del algoritmo es $\mathbf{\Theta}\left(n\sqrt{n}\right)$, o de manera equivalente $\mathbf{\Theta}\left(n^{\frac{3}{2}}\right)$.



Ahora consideremos este otro ejemplo que llamaremos `Iter2`, que nos servirá para contar el número de iteraciones. Consiste de dos ciclos anidados y una variable `cuenta` que almacena el número de iteraciones hechas por el algoritmo cuya entrada es un número entero positivo `n`.

```
PROBLEMA: Calcula el numero de iteraciones en un ciclo anidado.
ENTRADA : n  # un entero > 0
SALIDA  : numero entero
```

```
1  Iter2(n:entero):
2    cuenta ⟵ 0
3    for i desde 1 hasta n+1
4      m ⟵ ⌊n / i⌋
5      for j desde 1 hasta m+1
6        cuenta ⟵ cuenta + 1
7    return cuenta
```

```
import math

def Iter2(n):
  """
  Cuenta el numero veces que entra en el ciclo j
  """
  cuenta = 0
  for i in range(1,n+1):
    m = math.floor(n/i)
    for j in range(1,m+1):
      cuenta = cuenta + 1
  return cuenta

x = int(input())
Iter2(x)
```

Antes de hacer el análisis, vamos a hacer algunas ejecuciones para estar seguros de comprender cómo funciona este algoritmo.

Entrada	Salida
1	1
2	3
3	5
4	8
5	10
10	27

Observemos que en el ciclo externo se asignan valores para `i` desde 1 hasta `n`, mientras que en el ciclo interno los valores para `j` varían desde 1 hasta $\lfloor \frac{n}{i} \rfloor$. Observa que los valores posibles para $\lfloor \frac{n}{i} \rfloor$ son números enteros que van desde `n` hasta 1.

De nuevo vemos dos ciclos anidados que son simples aunque el límite superior del ciclo interno no siempre cambia de uno en uno, el cambio de la variable iteradora sí cambia en pasos unitarios.

Para encontrar el número de veces que se actualiza la variable `cuenta`, hay que notar que tal línea se ejecuta $\sum_{i=1}^{n+1} \sum_{j=1}^{m+1} 1$ veces:

$$\sum_{i=1}^{n+1}\sum_{j=1}^{m+1} 1 = \sum_{i=1}^{n+1}(m+1-1)$$

$$= \sum_{i=1}^{n+1} m$$

$$= \sum_{i=1}^{n+1} \left\lfloor \frac{n}{i} \right\rfloor.$$

Por la definición de la función suelo, sabemos que

$$\frac{n}{i} - 1 < \left\lfloor \frac{n}{i} \right\rfloor \leq \frac{n}{i}.$$

Entonces

$$\sum_{i=1}^{n} \left(\left\lfloor \frac{n}{i} \right\rfloor - 1 \right) < \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \leq \sum_{i=1}^{n} \frac{n}{i} \approx n \ln n$$

Entonces decimos que el paso 6 se hace $\mathbf{\Theta}(n \log n)$ veces. Como el tiempo de ejecución es proporcional a `cuenta`, concluimos que el algoritmo es $\mathbf{\Theta}(n \log n)$.

3.2. Ciclos no simples

Como vemos, la relación entre sumas y ciclos simples es directa. Si al menos un ciclo no es simple, entonces necesitamos crear un nuevo iterador [una nueva variable iteradora] que sea simple para incluirlo en la suma. Esta variable simple depende del iterador original, y al hacer el cambio es importante asegurarse de preservar la dependencia al evaluar la nueva suma.

Para comprender este tema, consideremos ahora el algoritmo `Iter3` que consiste de dos ciclos anidados y una variable `cuenta` que guarda el número de iteraciones hechas por el algoritmo. La entrada del algoritmo es un número entero n que es una potencia de 2, esto es $n = 2^k$ para algún entero no negativo k.

```
PROBLEMA: Cuenta el número de veces que se hace un ciclo cuando el contador depende de otro.
ENTRADA : n ; una potencia de 2.
SALIDA  : un número entero.
```

```
1  Iter3(n : entero):
2    cuenta ←- 0
3    i ←- 1
4    while i ≤ n
5      for j desde 1 hasta i
6        cuenta ←- cuenta + 1
7      i ←- 2 * i
8    return cuenta
```

```
def Iter3(n):
  """
  Cuenta el numero de veces que se hace un ciclo cuando el contador depende de otro
  donde:
  n : entero-potencia-de-2
  """
  cuenta = 0
  i = 1
  while i <= n:
    for j in range(1,i+1):
      cuenta = cuenta + 1
    i = 2 * i
  return cuenta

print(Iter3(int(input())))
```

Vamos a hacer algunas corridas para observar qué sucede en los ciclos:

n	i	j	cuenta
1	1	1	1
2	1	1	3
	2	1,2	
4	1	1	7
	2	1,2	
	4	1,2,3,4	
8	1	1	15
	2	1,2	
	4	1,2,3,4	
	8	1,2,3,4,5,6,7,8	
16	1	1	31
	2	1,2	
	4	1,2,3,4	
	8	1,2,3,4,5,6,7,8	
	16	1,2,3,4,5,6,7,8,9,10,...,16	
32	1	1	63
	2	1,2	
	4	1,2,3,4	
	8	1,2,3,4,5,6,7,8	
	16	1,2,3,4,5,6,7,8,9,10,...,16	
	32	1,2,3,4,5,6,7,8,9,10,...,32	

En este caso, es obvio que el ciclo `for` es simple, pero el ciclo `while` no lo es, porque depende del valor de entrada `n`. El iterador del ciclo `while` es la variable `i` que no es simple ya que su valor no se modifica de uno en uno, sino que se duplica en cada iteración. El iterador adquiere sus valores en la sucesión

$$\texttt{i} = 1, 2, 4, 8, \ldots, n,$$

que como son potencias de 2, se puede escribir como

$$\texttt{i} = 2^0, 2^1, 2^2, 2^3, \ldots, 2^k.$$

Ahora ya es más claro que el nuevo iterador, que debe ser un iterador simple, debe ser el exponente de la base 2. Elegimos ahora un nombre para la variable que no se utilice en el algoritmo como nuevo índice de la fórmula de la suma. Digamos que es `r`. Notemos la siguiente relación entre el nuevo iterador y el iterador original

$$\texttt{i} = 2^{\texttt{r}} \text{ o bien } \texttt{r} = \log \texttt{i}.$$

Ahora para contar cuántas veces se hace la línea 6 del algoritmo `Iter3`, podemos hacer

$$\begin{aligned}
\sum_{r=0}^{k} \sum_{j=1}^{i} 1 &= \sum_{r=0}^{k} i \\
&= \sum_{r=0}^{k} 2^{\texttt{r}} \\
&= 2^{k+1} - 1 \\
&= n + 1 - 1 \\
&= \boldsymbol{\Theta}(n)
\end{aligned}$$

Y concluimos que el algoritmo es de orden lineal $\boldsymbol{\Theta}(n)$.

Este nuevo ejemplo es el algoritmo `Iter4`, que consiste de dos ciclos anidados y una variable `cuenta` que cuenta el número de iteraciones hechas por el algoritmo cuando se ejecuta con una entrada $n = 2^k$, para algún número entero positivo k.

```
"""
Cuenta el numero de veces que se hace un ciclo cuando el contador
depende de otro
"""
ENTRADA : n = 2^k ; un numero entero positivo
SALIDA  : un numero entero


Iter4(n: entero):
cuenta ←- 0
while n ≥ 1
  for j desde 1 hasta n
    cuenta ←- cuenta + 1
  n ←- n/2
return count
```

```
def Iter4(n):
  """
  Cuenta el numero de veces que se hace un ciclo cuando el contador depende de otro
  donde:
  n : entero potencia de 2
  """
  cuenta = 0
  while n >= 1:
    for j in range(1,n+1):
      cuenta = cuenta + 1
    n = n // 2
  return cuenta

print(Iter4(int(input())))
```

Vamos a hacer algunas corridas para observar qué sucede en los ciclos:

n	j	cuenta
1	1	1
2	1,2 1	3
4	1,2,3,4 1,2 1	7
8	1,2,3,4,5,6,7,8 1,2,3,4 1,2 1	15
16	1,2,3,4,5,6,7,8,9,10,...,16 1,2,3,4,5,6,7,8 1,2,3,4 1,2 1	31
32	1,2,3,4,5,6,7,8,9,10,...,32 1,2,3,4,5,6,7,8,9,10,...,16 1,2,3,4,5,6,7,8 1,2,3,4 1,2 1	63

Aquí el ciclo `for` es simple, mientras que el iterador `while` inicia con el valor `n` y se decrementa a la mitad en cada iteración hasta que alcanza el valor de 1, inclusive. Observemos los valores que toma `n` cada vez que inicia el ciclo `while`:

$$\mathtt{n}, \frac{\mathtt{n}}{2}, \frac{\frac{\mathtt{n}}{2}}{2}, \frac{\frac{\mathtt{n}}{4}}{2}, \frac{\frac{\mathtt{n}}{8}}{2}, \ldots, 2, 1$$

Como `n` es una potencia de 2, es decir $\texttt{n} = 2^k$ para algún entero no negativo k, podemos reescribir la suceción como:

$$2^k, 2^{k-1}, 2^{k-2}, \ldots, 2^1, 2^0.$$

Y ahora es el momento de introducir una variable simple `r` que tome todos los valores desde 0 hasta k. Así, haciendo $i = 2^r$ tenemos $r = \log i$ y calculamos el valor para `cuenta`:

$$\sum_{r=0}^{k} \sum_{j=1}^{i} 1$$

Que es la misma que en el ejemplo anterior. Como el tiempo de ejecución es proporcional a `cuenta`, concluimos que es $\boldsymbol{\Theta}(n)$.

Ahora estudiemos este otro ejemplo. Consideremos el algoritmo `Iter5` que consiste también de dos ciclos anidados y una variable `cuenta` que cuenta el número de iteraciones hechas dentro del ciclo `while` con una entrada n que es de la forma 2^{2^k} para algún número entero positivo k ($k = \log\log n$).

```
PROBLEMA: Cuenta el numero de veces que se hace un ciclo cuando
          el contador depende de otro
ENTRADA : n = 2^(2^k)  ; un numero entero positivo
SALIDA  : un numero entero
```

```
1  Iter5(n: entero):
2  cuenta ←- 0
3  for i desde 1 hasta n
4    j ←- 2
5    while j ≤ n
6      j ←- j^2
7      cuenta ←- cuenta + 1
8  return count
```

En este caso, el ciclo `for` es simple, pero el ciclo `while` no lo es. Así que veamos los valores que debe tener j.

$$\begin{aligned} j &= 2, 2^2, 4^2, 16^2, \ldots, n \\ &= 2^1, (2^1)^2, (2^2)^2, (2^4)^2, \ldots, n \\ &= 2^1, 2^2, 2^4, 2^8, \ldots, n \\ &= 2^{2^0}, 2^{2^1}, 2^{2^2}, \ldots, 2^{2^k} \end{aligned}$$

Ahora nos fijamos en que el exponente más externo, si cambia de uno en uno, por lo que se puede tratar como un iterador simple. De modo que introduciremos la nueva variable r de tal modo que $j = 2^{2^r}$ y en consecuencia $r = \log\log j$. El valor de `cuenta` se vuelve

$$\begin{aligned} \sum_{i=1}^{n} \sum_{r=0}^{k} 1 &= \sum_{i=0}^{n} (k+1) \\ &= \sum_{i=1}^{n} (\log\log n + 1) \\ &= (\log\log n + 1) \sum_{i=1}^{n} 1 \\ &= n(\log\log n + 1). \end{aligned}$$

Entonces el tiempo de ejecución es $\boldsymbol{\Theta}(n \log\log n)$

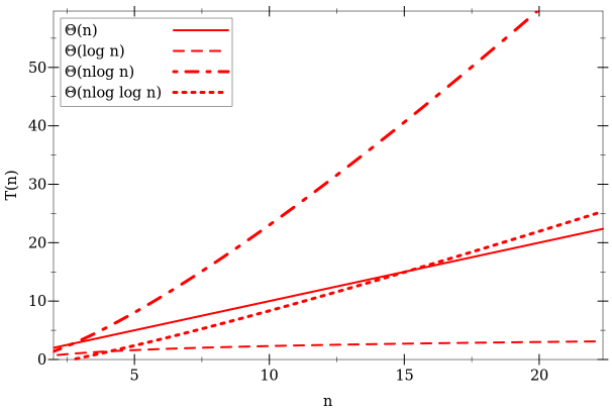


Figura 1: Comparativa de diferentes clases relacionadas con $\log n$

Ejercicios

Criterio de evaluación

Para que seas conciente de cómo se evalúan y califican estos ejercicios, considera el siguiente criterio que otorga cierto número de puntos por cada ejercicio:

- 3: Un ejercicio entregado a tiempo y sin alguna falla. Un ejercicio de programación corre a la perfección; sigue fielmente las instrucciones y su notación, y tiene su contrato correctamente escrito.
- 2: Un ejercicio mayormente correcto entregado a tiempo. Un ejercicio de programación o bien no tiene su contrato; o bien ha cambiado identificadores o es parcialmente correcto.
- 1: Un ejercicio entregado a destiempo o mayormente equivocado. Un ejercicio de programación no tiene su contrato y ha cambiado identificadores y es parcialmente correcto.
- 0: Un ejercicio completamente equivocado o no se entrega.

La calificación de la lista de ejercicios se obtiene al ponderar el número de puntos alzanzado en una escala de 0 a 10.

1. Observa el siguiente problema computacional y el algoritmo que lo resuelve:

```
PROBLEMA: Ordenar una lista de números enteros de menor a mayor
ENTRADA: Una lista A de n números enteros
SALIDA: Una permutación de la lista, de modo que los números estén ordenados de menor a mayor.
```

```
1  burbuja(A):
2    for i desde 1 hasta |A| − 1:
3      for j desde 0 hasta |A| − 2:
4        si A_j > A_{j+1}:
5          aux ⟵ A_j
6          A_j ⟵ A_{j+1}
7          A_{j+1} ⟵ aux
8    return A
```

Análiza la complejidad para este algoritmo y proporciona una cota superior para el tiempo de ejecución.

2. Escribe un programa para resolver el problema de ordenar una lista de números mediante el método de burbuja [ejercicio anterior]. Debes proporcionar un programa que se llame `burbuja.py` con la entrada dada en el momento de la ejecución como en los siguientes ejemplos:

Entrada	Salida
`3 9 1 0 7 2 9 23 12`	`[0, 1, 2, 3, 7, 9, 9, 12, 23]`
lista – vacia	`[]`
`1 2 3 4`	`[1, 2, 3, 4]`

3. Escribe un programa basado en cíclos llamad `palindromos.py`, que verifique si todas las palabras de una colección de palabras son palíndromos.

```
PROBLEMA: Determina si todas las palabras de una colecci\'on de palabras son palíndromos
ENTRADA: Pals : Un arreglo de palabras.
SALIDA: Un valor booleano. True si todas las palabras en el arreglo son palíndromos, False si no
     todas las palabras lo son.
```

Entrada	Salida
lista – vacia	`True`
`ana oso reconocer radar aerea`	`True`
`ana oso reconocer radare aerea`	`False`

4. Analiza la complejidad del programa que escribiste en el ejercicio anterior y dibuja una gráfica con la función asintótica que obtviste.

5. Estudia el siguiente algoritmo:

```
1  Iter6(n):
2    cuenta ⟵ 0
3    for i desde 1 hasta n:
4      for j desde i+1 hasta n:
5        for k desde j+1 hasta n:
6          cuenta ⟵ cuenta + 1
7    return cuenta
```

Donde la función `agregar(e, lst)`, toma como entrada un elemento `e` y una lista `lst`. La salida es una arreglo con todos los elementos de `lst` y además un nuevo elemento `e` que está en el extremo derecho de la lista `lst`.

Realiza y reporta el análisis de complejidad de este algoritmo.