Complejidad

E. Rivas

Mayo de 2014

Introducción

¿Cuándo un algoritmo es mejor que otro?

Introducción

¿Cuándo un algoritmo es mejor que otro? Cuando tarda menos en ejecutarse.

Introducción

¿Cuándo un algoritmo es mejor que otro? Cuando tarda menos en ejecutarse.

¿Y cuánto tarda una función en ejecutarse?

Introducción

¿Cuándo un algoritmo es mejor que otro? Cuando tarda menos en ejecutarse.

¿Y cuánto tarda una función en ejecutarse?

Una opción para responder esta pregunta es compilar nuestro programa y ejecutarlo. Sin embargo, esta respuesta depende de muchos factores, entre ellos:

- Hardware.
- Software.
- Actividad del usuario en la PC.

Introducción

¿Cuándo un algoritmo es mejor que otro? Cuando tarda menos en ejecutarse.

¿Y cuánto tarda una función en ejecutarse?

Una opción para responder esta pregunta es compilar nuestro programa y ejecutarlo. Sin embargo, esta respuesta depende de muchos factores, entre ellos:

- Hardware.
- Software.
- Actividad del usuario en la PC.

Debemos buscar una manera más "neutra" de calcular el tiempo que tarda en ejecutarse una función.

Contando instrucciones

Una opción para contar cuánto tiempo tarda en ejecutarse una función, es contar la cantidad de instrucciones que se ejecutan cuando se llama a esa función.

```
void f() {
  int x, y, z;

  x = 3;
  y = 4;
  z = x + y;
}
```

Contando instrucciones

Una opción para contar cuánto tiempo tarda en ejecutarse una función, es contar la cantidad de instrucciones que se ejecutan cuando se llama a esa función.

```
 0:    push   %ebp
 1:    mov    %esp,%ebp
 3:    sub    $0x10,%esp
 6:    movl   $0x3,-0xc(%ebp)
 d:    movl   $0x4,-0x8(%ebp)
14:    mov    -0x8(%ebp),%eax
...
```

Contando instrucciones

Una opción para contar cuánto tiempo tarda en ejecutarse una función, es contar la cantidad de instrucciones que se ejecutan cuando se llama a esa función.

```
 0:    push   %ebp
 1:    mov    %esp,%ebp
 3:    sub    $0x10,%esp
 6:    movl   $0x3,-0xc(%ebp)
 d:    movl   $0x4,-0x8(%ebp)
14:    mov    -0x8(%ebp),%eax
...
```

Este programa se compila a 11 instrucciones.

Contando instrucciones

Una opción para contar cuánto tiempo tarda en ejecutarse una función, es contar la cantidad de instrucciones que se ejecutan cuando se llama a esa función.

```
 0:    push   %ebp
 1:    mov    %esp,%ebp
 3:    sub    $0x10,%esp
 6:    movl   $0x3,-0xc(%ebp)
 d:    movl   $0x4,-0x8(%ebp)
14:    mov    -0x8(%ebp),%eax
...
```

Este programa se compila a 11 instrucciones.
Al ejecutar esta función, se ejecutan exactamente 11 instrucciones.

Problemas de esta forma

Sin embargo, contar instrucciones de esta manera sigue teniendo desventajas:

- Depende del compilador.
- Depende de la arquitectura.

Problemas de esta forma

Sin embargo, contar instrucciones de esta manera sigue teniendo desventajas:

- Depende del compilador.
- Depende de la arquitectura.

Por estas razones vamos a dejar de lado la opción de contar instrucciones de procesador, y vamos a contar instrucciones de C.

Problemas de esta forma

Sin embargo, contar instrucciones de esta manera sigue teniendo desventajas:

- Depende del compilador.
- Depende de la arquitectura.

Por estas razones vamos a dejar de lado la opción de contar instrucciones de procesador, y vamos a contar instrucciones de C.

Supondremos que cada asignación, suma, resta, multiplicación, división, indexado de arreglo, indexado de puntero, compración, etc.... cuentan como una operación.

Ejercicio

Contar la cantidad de instrucciones que ejecuta al llamar a `f`:

```
void f() {
  int x, y, z;

  x = 3;
  y = 4;
  z = x + y;

  if (x == 1)
    z = 2;
}
```

Ejercicio

Contar la cantidad de instrucciones que ejecuta al llamar a `f`:

```
void f() {
  int x, y, z;

  x = 3;
  y = 4;
  z = x + y;

  if (x == 1)
    z = 2;
}
```

Respuesta: 5

Funciones con argumentos

Si la función no tiene entradas, entonces la cantidad de operaciones ejecutadas es un número fijo.

Funciones con argumentos

Si la función no tiene entradas, entonces la cantidad de operaciones ejecutadas es un número fijo.

Sin embargo, si la función tiene entradas, veremos que la complejidad deja de ser un número fijo:

```
int sum(int n) {
  int i, r = 0;

  for (i = 1; i <= n; i++)
    r = r + i;

  return r;
}
```

Funciones con argumentos

Si la función no tiene entradas, entonces la cantidad de operaciones ejecutadas es un número fijo.

Sin embargo, si la función tiene entradas, veremos que la complejidad deja de ser un número fijo:

```
int sum(int n) {
  int i, r = 0;

  for (i = 1; i <= n; i++)
    r = r + i;

  return r;
}
```

¿Cuántas instrucciones ejecuta esta función?

Complejidad dependiente

La complejidad de la función `sum` dependerá de `n`.

- Si `n` vale 0... ejecuta 4 instrucciones.
- Si `n` vale 1... ejecuta 8 instrucciones.
- Si `n` vale 2... ejecuta 12 instrucciones.
- Si `n` vale 3... ejecuta 16 instrucciones.
- Si `n` vale n... ejecuta

Complejidad dependiente

La complejidad de la función `sum` dependerá de `n`.

- Si `n` vale 0... ejecuta 4 instrucciones.
- Si `n` vale 1... ejecuta 8 instrucciones.
- Si `n` vale 2... ejecuta 12 instrucciones.
- Si `n` vale 3... ejecuta 16 instrucciones.
- Si `n` vale n... ejecuta $4 + 4n$ instrucciones.

Complejidad dependiente

La complejidad de la función `sum` dependerá de `n`.

- Si `n` vale 0... ejecuta 4 instrucciones.
- Si `n` vale 1... ejecuta 8 instrucciones.
- Si `n` vale 2... ejecuta 12 instrucciones.
- Si `n` vale 3... ejecuta 16 instrucciones.
- Si `n` vale n... ejecuta $4 + 4n$ instrucciones.

¡La complejidad depende del argumento de la función!

Formalmente, la complejidad de f, notada C_f, depende de n y vale: $C_f(n) = 4 + 4n$.

Funciones con más argumentos

La función puede depender de más de un argumento, o incluso de información "dinámica":

```
int min(int data[], int sz) {
  int i, min = data[0];

  for (i = 0; i < sz; i++)
    if (data[i] < min)
      min = data[i];

  return min;
}
```

Funciones con más argumentos

La función puede depender de más de un argumento, o incluso de información "dinámica":

```
int min(int data[], int sz) {
  int i, min = data[0];

  for (i = 0; i < sz; i++)
    if (data[i] < min)
      min = data[i];

  return min;
}
```

¿Cuántas instrucciones ejecuta esta función?

Complejidad dependiente con arreglos

La complejidad de la función `min` dependerá de `data` y `sz`.

- Si `sz` vale 0... ejecuta 5 instrucciones.
- Si `sz` vale 1... ejecuta 9 instrucciones.
- Si `sz` vale 2, y `data` apunta a $[2, 4]$... ejecuta 13 instrucciones.
- Si `sz` vale 2, y `data` apunta a $[4, 2]$... ejecuta 15 instrucciones.

Complejidad dependiente con arreglos

La complejidad de la función `min` dependerá de `data` y `sz`.

- Si `sz` vale 0... ejecuta 5 instrucciones.
- Si `sz` vale 1... ejecuta 9 instrucciones.
- Si `sz` vale 2, y `data` apunta a $[2, 4]$... ejecuta 13 instrucciones.
- Si `sz` vale 2, y `data` apunta a $[4, 2]$... ejecuta 15 instrucciones.

La cantidad de instrucciones ejecutadas no solo depende de `sz`, si no también de la forma del arreglo al que apunta `data`.

Ejercicio

Asumiendo que `data` apunta a un arreglo ordenado de tamaño n, y que `sz` vale n.

```
int min(int data[], int sz) {
  int i, min = data[0];

  for (i = 0; i < sz; i++)
    if (data[i] < min)
      min = data[i];

  return min;
}
```

¿Cuántas instrucciones ejecuta esta función?

Ejercicio

Asumiendo que `data` apunta a un arreglo ordenado de tamaño n, y que `sz` vale n.

```
int min(int data[], int sz) {
  int i, min = data[0];

  for (i = 0; i < sz; i++)
    if (data[i] < min)
      min = data[i];

  return min;
}
```

¿Cuántas instrucciones ejecuta esta función? $5 + 4n$.
Formalmente, $C_f(n) = 5 + 4n$.

Tirando las constantes

Esta medida ya no depende del hardware, el software, o la activdad que esté desarrollando el usuario.

Sin embargo, depende del lenguaje de programación que estamos utilizando: C.

Tirando las constantes

Esta medida ya no depende del hardware, el software, o la activdad que esté desarrollando el usuario.

Sin embargo, depende del lenguaje de programación que estamos utilizando: C.

En otros lenguajes de programación, tenemos diferentes operaciones constantes, y quisieramos que la medida de tiempo de nuestros algoritmos se pueda comparar sin importar dónde lo implementamos.

Tirando las constantes

Esta medida ya no depende del hardware, el software, o la activdad que esté desarrollando el usuario.

Sin embargo, depende del lenguaje de programación que estamos utilizando: C.

En otros lenguajes de programación, tenemos diferentes operaciones constantes, y quisieramos que la medida de tiempo de nuestros algoritmos se pueda comparar sin importar dónde lo implementamos.

Es por esto que “tiraremos las constantes”.

Comportamiento asintótico

La idea formal que estará detrás de "tirar constantes" será la idea de *comportamiento asintótico*.

Comportamiento asintótico

La idea formal que estará detrás de "tirar constantes" será la idea de *comportamiento asintótico*.

Nos quedaremos con el coeficiente de mayor crecimiento de la función que determina la cantidad de instrucciones.

- Si $C_f(n) = 4 + 3n$, entonces nos quedamos con n.
- Si $C_g(n) = 11$, entonces nos quedamos con 1.
- Si $C_h(n) = 3n^2 + 5n + 2$, entonces nos quedamos con n^2.

Comportamiento asintótico

La idea formal que estará detrás de "tirar constantes" será la idea de *comportamiento asintótico*.

Nos quedaremos con el coeficiente de mayor crecimiento de la función que determina la cantidad de instrucciones.

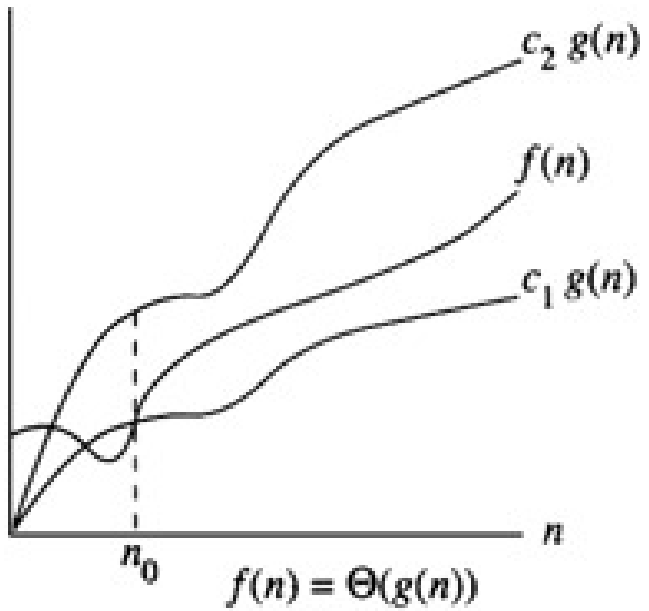
- Si $C_f(n) = 4 + 3n$, entonces nos quedamos con n.
- Si $C_g(n) = 11$, entonces nos quedamos con 1.
- Si $C_h(n) = 3n^2 + 5n + 2$, entonces nos quedamos con n^2.

Esto lo notaremos utilizando la notación Θ ("zeta"):

- $C_f \in \Theta(n)$
- $C_g \in \Theta(1)$
- $C_h \in \Theta(n^2)$

Un poco de análisis

Matemáticamente, decimos que $f \in \Theta(g)$ si
$\exists c_1 > 0 \cdot \exists c_2 > 0 \cdot \exists n_0 \cdot \forall n > n_0 \cdot c_1 g(n) \leq f(n) \leq c_2 g(n)$



Informalmente, esto dice que f en el límite está acotada inferiormente por un múltiplo de g, y superiormente por otro múltiplo de g.

Volviendo al problema

Para calcular la cantidad de instrucciones que ejecutaba `min`, supusimos que el arreglo apuntado por `data` estaba ordenado.

Si quitamos esta restricción, nos vemos imposibilitados de calcular la complejidad.

¿Qué es lo que hacemos cuando nos restringimos a arreglos ordenados? Nos restringimos al "peor caso".
Cuando consideramos el *peor caso*, estamos considerando una cota superior de qué tan malo va a ser el algoritmo.

Empeorando una función

Imaginemos que tenemos una función `f`:

```
int f(int n) {
  int i, j, r = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < i; j++)
      r = r + j;

  return r;
}
```

La complejidad dependerá de n. El bucle interior depende de una variable que cambia en cada iteración del bucle exterior.

Nuestro enfoque cuando tengamos esta clase de conflicto será: empeorar la función.

Empeorando una función

Modificando `f` a `g`:

```
int g(int n) {
  int i, j, r = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      r = r + j;

  return r;
}
```

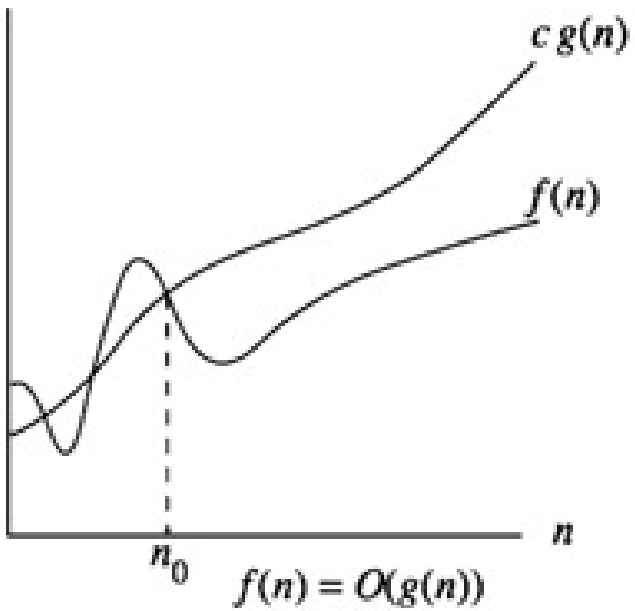
Cambiamos a `i` por `n` en el bucle interior.

La complejidad ahora es fácil de calcular: $C_g \in \Theta(n^2)$.

Notación Big O

En general, tendremos una notación para decir que una función f se comporta a lo sumo tan mal como otra g: $f \in O(g)$.

Formalmente, esto corresponde también a un límite:
$\exists c > 0 \cdot \exists n_0 \cdot \forall n > n_0 \cdot f(n) \leq cg(n)$



Recursión

El siguiente algoritmo calcula factorial recursivamente:

```
int f(int n) {
  if (n == 0)
    return 1;
  return n*fact(n - 1);
}
```

Recursión

El siguiente algoritmo calcula factorial recursivamente:

```
int f(int n) {
  if (n == 0)
    return 1;
  return n*fact(n - 1);
}
```

¿Cómo calculamos la complejidad de `f`?

Recursión

El siguiente algoritmo calcula factorial recursivamente:

```
int f(int n) {
  if (n == 0)
    return 1;
  return n*fact(n - 1);
}
```

¿Cómo calculamos la complejidad de `f`?

Si intentamos escribir la ecuación, obtenemos algo como:
$C_f(n) = 3 + C_f(n-1)$ si $n > 0$.

Solución recursiva

La función `f` define un sistema de dos ecuaciones:

$$\begin{aligned} C_f(0) &= 2 \\ C_f(n) &= 4 + C_f(n-1) \text{ si } n > 0 \end{aligned}$$

es una ecuación funcional: la incógnita, C_f es una función.

Solución recursiva

La función `f` define un sistema de dos ecuaciones:

$$\begin{aligned} C_f(0) &= 2 \\ C_f(n) &= 4 + C_f(n-1) \text{ si } n > 0 \end{aligned}$$

es una ecuación funcional: la incógnita, C_f es una función.

En este caso la solución es sencilla de encontrar:

$$C_f(n) = 2 + 4n$$

Recursivas no lineales

En algunos casos, la recursión utiliza casos no inmediatamente anteriores, si no casos que utilizan una complejidad de la mitad del tamaño:

```
void sum(int data[], int init, int last) {
  int mid = (init + last)/2;

  if (last < init)
    return 0;
  else if (init == last)
    return data[init];
  else
    return sum(data, init, mid) +
           sum(data, mid + 1, last);
}
```

Recursivas no lineales

En este caso, la ecuación de complejidad de `sum` queda como

$$C_{sum}(n) = C_{sum}(n/2) + C_{sum}(n/2) + 5$$

Recursivas no lineales

En este caso, la ecuación de complejidad de `sum` queda como

$$C_{sum}(n) = C_{sum}(n/2) + C_{sum}(n/2) + 5$$

La solución a este tipo de ecuaciones se puede realizar utilizando el *teorema maestro*. Este método nos brinda directamente la complejidad del algoritmo en notación Θ.

Recursivas no lineales

En este caso, la ecuación de complejidad de `sum` queda como

$$C_{sum}(n) = C_{sum}(n/2) + C_{sum}(n/2) + 5$$

La solución a este tipo de ecuaciones se puede realizar utilizando el *teorema maestro*. Este método nos brinda directamente la complejidad del algoritmo en notación Θ.

Es sencillo verificar, a partir del teorema, que $C_{sum} \in \Theta(n)$.

Resumen

Hoy estudiamos el problema de complejidad temporal de un algoritmo:

- Delimitamos mejor el problema.
- Introdujimos notación para determinar a qué "velocidad" se ejecuta un algoritmo.
- Pero siempre teniendo en cuenta que esta medida es teórica: son límites.

Además, vimos una noción de empeoramiento de programa para poder calcular cotas superiores, y comentamos la existencia de un teorema para resolver funciones recursivas.