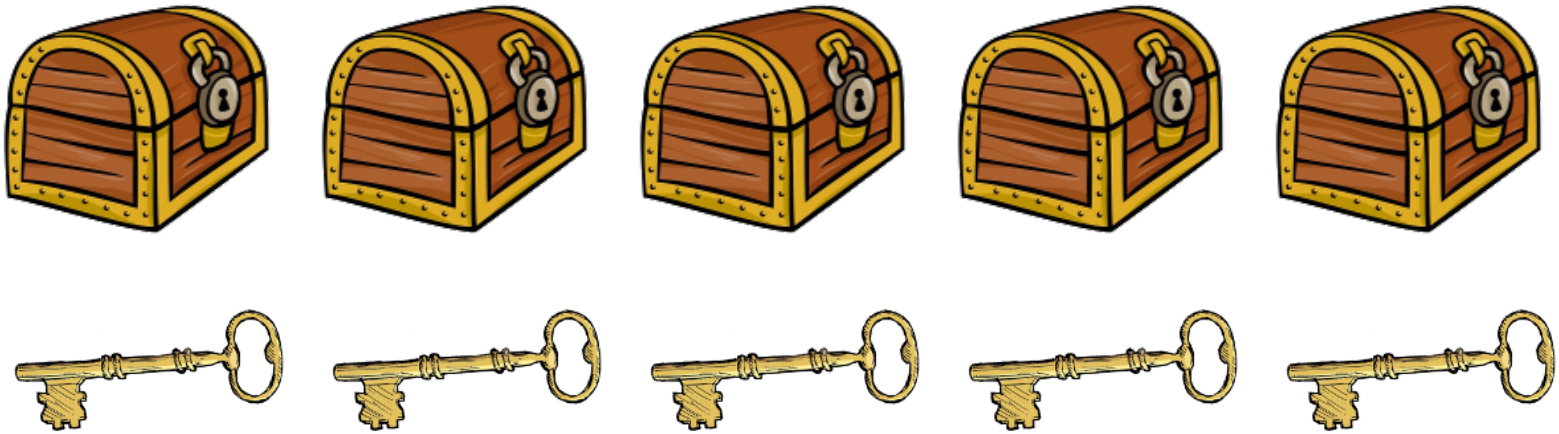
Introducción a la Programación

Prof. Agustín Gravano

Primer semestre de 2022

Clase teórica 16: Complejidad algorítmica

Problema: Cofres y llaves



Tenemos N cofres cerrados con llave y N llaves necesarias para abrirlos, pero no sabemos cuál llave abre cuál cofre.

Podría ocurrir, por ejemplo, que algunas llaves sean inútiles y otras llaves abran más de un cofre. Pero no sabemos nada al respecto.

Lo único que sabemos con certeza es que es posible abrir todos los cofres.

Queremos escribir un algoritmo para abrir todos los cofres.
¿Cuántos pasos llevará? ¿Cuál será la complejidad de este algoritmo?

Problema: Cofres y llaves - Posible algoritmo

```
1  Abrir los N cofres:
2      i = 0                                              O(1)
3      mientras i < N:                                    N iteraciones
4          j = buscar una llave para el cofre i           O(?)
5          abrir el cofre i con la llave j                O(1)
6          i = i + 1                                      O(1)
```

Las líneas 2, 5 y 6 se ejecutan en tiempo constante.

El ciclo se repite N veces (la cantidad de cofres).

¿Pero qué pasa con la línea 4?

Buscar una llave correcta para el cofre i depende de la cantidad de llaves (también N). Por eso, la línea 4 no tiene O(1).

Para analizar la complejidad de la línea 4, veamos un posible algoritmo que la implemente.

Problema: Cofres y llaves - Posible algoritmo

```
1  Buscar una llave que abra el cofre c:
2      j = 0                                          O(1)
3      mientras j < N:                                N iteraciones
4          si la llave j abre el cofre c:             O(1)
5              devolver j                             O(1)
6          j = j + 1                                  O(1)
```

Las líneas 2, 4, 5 y 6 se ejecutan en tiempo constante.

El ciclo se repite N veces (la cantidad de llaves) en el peor caso.

Entonces, buscar una llave correcta para un cofre tiene O(N).

(Notar que esto es similar a buscar el A♡ en un mazo de naipes.)

Problema: Cofres y llaves - Posible algoritmo

```
1  Abrir los N cofres:
2      i = 0                                              O(1)
3      mientras i < N:                                    N iteraciones
4          j = buscar una llave para el cofre i           O(N)
5          abrir el cofre i con la llave j                O(1)
6          i = i + 1                                      O(1)
```

Volviendo al algoritmo principal, tenemos que el ciclo ejecuta N veces una operación que tiene O(N).

Razonamiento similar: ¿cuánto tiempo demanda repetir 10 veces una tarea de 5 minutos? Claramente, $10 \times 5 = 50$ minutos.

Por lo tanto, el algoritmo principal tiene $O(N \times N) = O(N^2)$.

Para abrir todos los cofres, la cantidad de operaciones crece cuadráticamente con la cantidad de llaves y cofres.

Cálculo de órdenes de complejidad

Operaciones con complejidad constante, O(1):

- lectura/escritura de variables
- operaciones simples de `bool`: `not and or ==` ...
- operaciones simples de `int` y `float`: `+ - * / // % == > <` ...
- operaciones del módulo `math`: `pi e sin cos tan log sqrt` ...
- para `s:str`: `len(s) s[i] ord(s) chr(s)`
- para `xs:List[T]`: `xs[i] len(xs) xs.append(t) xs.pop()`

Esto no significa que todas estas operaciones tarden lo mismo en ejecutarse. Significa que todas tardan cierto tiempo constante.

Atención: Las restantes operaciones de strings y listas en general **no** tienen complejidad constante. Por ejemplo:

- Para `s1,s2:str`, `s1+s2`, `s1 in s2`, `s1==s2`, `s1[i:j:s]` son lineales.
- Para `xs,ys:List[T]`, `t in xs`, `xs.insert(i,t)`, `xs.pop(i)`, `xs==ys`, `xs+ys`, `xs[i:j:s]` son lineales (https://wiki.python.org/moin/TimeComplexity).

Cálculo de órdenes de complejidad

¿Qué complejidad tienen estas instrucciones?

```
1  n:int = 10
2  m:int = n * 2 + 5
```

Línea 1: solo tiene una escritura en la variable `n`, lo cual es $O(1)$.

Línea 2: involucra varias operaciones, que se ejecutan secuencialmente:

a) lectura de la variable `n` $O(1)$
b) producto de enteros $O(1)$
c) suma de enteros $O(1)$
d) escritura en la variable `m` $O(1)$

Ejecutar un número fijo (o sea, constante) de cosas que tardan un tiempo constante, también tarda un tiempo constante. (Mayor, pero constante!)

Formalmente, decimos: $O(1) + O(1) + O(1) + O(1) = O(1)$

Cálculo de órdenes de complejidad

Secuencialización:

Si CÓDIGO1 tiene $O(f)$ y CÓDIGO2 tiene $O(g)$, entonces:

CÓDIGO1
CÓDIGO2

tiene $O(f) + O(g) = O(\text{máx}(f, g))$.

Ejemplo: (dadas `n, m: int`)

```
1  m = n * 2 + 5              # O(1)
2  m = sumatoria(n)           # O(n)
```

El código de la línea 1 tiene $O(1)$, y el código de la línea 2 tiene $O(n)$.

La complejidad del programa completo es $O(\text{máx}(1, n)) = O(n)$.

Cálculo de órdenes de complejidad

Ciclo:

Si CÓDIGO tiene $O(g)$ y se lo ejecuta $O(f)$ veces, entonces estos dos programas:

```
for ...:
    CÓDIGO
```

```
while ...:
    CÓDIGO
```

tienen $O(f) * O(g) = O(f * g)$.

Ejemplo: (dadas `i, n, m: int`)

```
1  for i in range(n):      # O(n) iteraciones
2      m = m + i           # O(1)
```

La complejidad del programa completo es $O(n * 1) = O(n)$.

Cálculo de órdenes de complejidad

Condicional:

Si CONDICIÓN tiene $O(f)$, CÓDIGO1 tiene $O(g)$ y CÓDIGO2 tiene $O(h)$, entonces:

```
if CONDICIÓN:
    CÓDIGO1
else:
    CÓDIGO2
```

tiene $O(f) + O(\text{máx}(g, h)) = O(\text{máx}(f, g, h))$.

Ejemplo: (dadas `n, m: int`)

```
1  if es_primo(n):          # O(√n)
2      m = n                # O(1)
3  else:
4      m = sumatoria(n)     # O(n)
```

La complejidad del programa completo es $O(\text{máx}(f, g, h)) = O(n)$.
(Recordar que nos interesa siempre el peor caso.)

Cálculo de órdenes de complejidad · Resumen

- Operaciones simples: tienen complejidad constante, $O(1)$.
- Secuencialización: Si `CÓDIGO1` y `CÓDIGO2` tienen $O(f)$ y $O(g)$, entonces `CÓDIGO1;CÓDIGO2` tiene $O(f) + O(g) = O(\text{máx}(f, g))$.
- Condicional: Si `CONDICIÓN`, `CÓDIGO1` y `CÓDIGO2` tienen $O(f)$, $O(g)$ y $O(h)$, entonces `if CONDICIÓN: CÓDIGO1 else: CÓDIGO2` tiene $O(f) + O(\text{máx}(g, h)) = O(\text{máx}(f, g, h))$.
- Ciclo: Si `CÓDIGO` tiene $O(g)$ y se lo ejecuta $O(f)$ veces, entonces `for ...: CÓDIGO` tiene $O(f) * O(g) = O(f * g)$.

Ejercicio: Determinar la complejidad de cada función

```python
def sumatoria(n:int) -> int:
    ''' Devuelve la suma de los primeros n naturales. '''
    vr:int = 0
    i:int = 1
    while i <= n:
        vr = vr + i
        i = i + 1
    return vr

def lista_sumatorias_v1(n:int) -> List[int]:
    ''' Devuelve la lista de sumatorias de 1, 2, ..., n. '''
    vr:List[int] = []
    i:int = 1
    while i<=n:
        vr.append(sumatoria(i))
        i = i + 1
    return vr

def lista_sumatorias_v2(n:int) -> List[int]:
    ''' Devuelve la lista de sumatorias de 1, 2, ..., n. '''
    vr:List[int] = []
    i:int = 1
    s:int = 1
    while i<=n:
        vr.append(s)
        i = i + 1
        s = s + i
    return vr
```

Repaso de la clase de hoy

- Algoritmos con complejidad cuadrática: $O(N^2)$
- Cálculo de órdenes de complejidad.

Con lo visto, ya pueden resolver hasta el Ejercicio 5 (inclusive) de la Guía de Ejercicios 7.

Bibliografía: capítulo 2 de "The Algorithm Design Manual", S. Skiena, 2da edición, Springer (disponible en el campus).