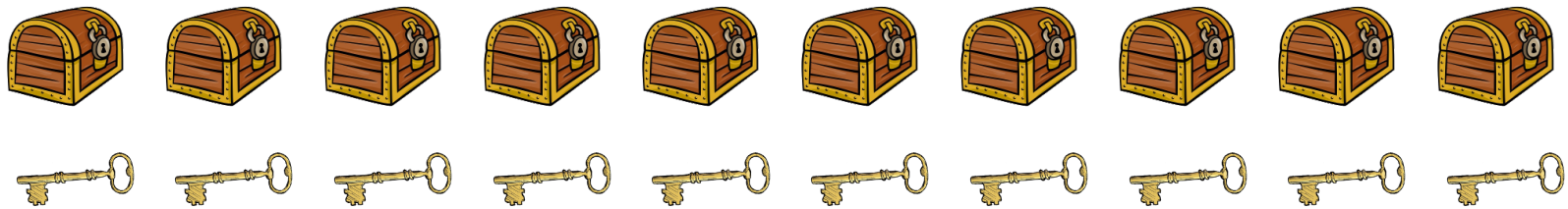
Introducción a la Programación

Prof. Agustín Gravano

Primer semestre de 2022

Clase teórica 17: Algoritmos de ordenamiento

Repaso



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

```
1  Abrir N cofres:
2      i = 0                                                    O(1)
3      mientras i < N:                  N iteraciones; condición: O(1)
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)
```

Este algoritmo tiene $O(N) \times O(N) = O(N^2)$.

Repaso

```
1  def lista_sumatorias_v1(n:int) -> List[int]:
2      ''' Devuelve la lista de sumatorias de 1,2,...,n. '''
3      vr:List[int] = []                    O(1)
4      i:int = 1                            O(1)
5      while i<=n:                          O(n) iteraciones; condición: O(1)
6          s:int = sumatoria(i)             ???
7          vr.append(s)                     O(1)
8          i = i + 1                        O(1)
9      return vr                            O(1)
```

La línea 6 tiene complejidad $O(\texttt{i})$: computar la sumatoria desde 1 hasta `i` lleva una cantidad de pasos proporcional a `i`.

De todas las iteraciones, la "peor" es la última, cuando hay que computar `sumatoria(n)`. Entonces, podemos afirmar que todas las iteraciones demandan como máximo `n` pasos, y por lo tanto tienen $O(\texttt{n})$. (Es decir, $O(n)$ es una cota superior válida de la complejidad de cada iteración.)

Luego, podemos ver que este algoritmo tiene $\texttt{O(n)} \times \texttt{O(n)} = \texttt{O(n}^2\texttt{)}$.

Asignaciones simultáneas

Python permite hacer asignaciones simultáneas de variables:

$$(\text{VAR}_1, \text{VAR}_2, \ldots, \text{VAR}_k) = (\text{EXPR}_1, \text{EXPR}_2, \ldots, \text{EXPR}_k)$$

Por ejemplo, si tenemos dos variables `a:int`, `b:str`:

```
1  (a, b) = (123, 'hola')
2  print(a, b)          # imprime: 123 hola
```

Entre otras cosas, esto nos permite intercambiar el contenido de dos variables (del mismo tipo) en una sola instrucción:

```
1  a:int = 10
2  b:int = 20
3  print(a, b)          # imprime: 10 20
4  (a, b) = (b, a)      # swap!
5  print(a, b)          # imprime: 20 10
```

Los paréntesis de la asignación son opcionales, pero aportan claridad.

Problema de ordenamiento

59	7	388	41	2	280	50	123

Selection sort

Para cada `i` entre `0` y `len(A)−1` (inclusive):
Buscar el menor elemento en `A[i:]`.
Intercambiarlo con `A[i]`.

0	1	2	3	4	5	6	7
59	7	388	41	2	280	50	123
2	7	388	41	59	280	50	123
2	7	388	41	59	280	50	123
2	7	41	388	59	280	50	123
2	7	41	50	59	280	388	123
2	7	41	50	59	280	388	123
2	7	41	50	59	123	388	280
2	7	41	50	59	123	280	388
2	7	41	50	59	123	280	388

Predicado invariante: $0 \leq$ `i` $\leq$ `len(A)` y `A[0:i]` está ordenada.

```python
def pos_minimo(L:List[int]) -> int:
    ''' Devuelve la posición del elemento mínimo de L. '''
    pos:int = 0
    for j in range(0, len(L)):
        if L[j] < L[pos]:
            pos = j
    return pos
```

```python
def selection_sort(A:List[int]):
    ''' Ordena (modifica) la lista A de menor a mayor. '''
    for i in range(0, len(A)):
        sublista:List[int] = A[i:]
        pm:int = pos_minimo(sublista) + i
        (A[i],A[pm]) = (A[pm],A[i])  #swap
```

Cálculo de órdenes de complejidad · Repaso

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

```
1  def pos_minimo(L:List[int]) -> int:
2      ''' Devuelve la posición del elemento mínimo de L. '''
3      pos:int = 0                                  O(1)
4      for j in range(0, len(L)):                   O(len(L)) iteraciones
5          if L[j] < L[pos]:                            O(1)
6              pos = j                                  O(1)
7      return pos                                   O(1)
```

$O(1) + O(len(L)) * \text{máx}(O(1), O(1)) + O(1) = O(len(L))$

Entonces, `pos_minimo(L)` es lineal respecto de `len(L)`.

```
1  def selection_sort(A:List[int]):
2      ''' Ordena (modifica) la lista A de menor a mayor. '''
3      for i in range(0, len(A)):                   O(len(A)) iteraciones
4          sublista:List[int] = A[i:]                   O(len(A))
5          pm:int = pos_minimo(sublista) + i            O(len(A))
6          (A[i],A[pm]) = (A[pm],A[i])  #swap           O(1)
```

$O(len(A)) * \Big(O(len(A)) + O(len(A)) + O(1)\Big) = O(len(A)^2)$

Entonces, `selection_sort(A)` es cuadrático respecto de `len(A)`.

Insertion sort

Para cada `i` entre `0` y `len(A)`-1 (inclusive):
Sacar el elemento `A[i]`.
Insertarlo en la posición correcta en `A[0:i]`.

0	1	2	3	4	5	6	7
59	7	388	41	2	280	50	123
59	7	388	41	2	280	50	123
7	59	388	41	2	280	50	123
7	59	388	41	2	280	50	123
7	41	59	388	2	280	50	123
2	7	41	59	388	280	50	123
2	7	41	59	280	388	50	123
2	7	41	50	59	280	388	123
2	7	41	50	59	123	280	388

Predicado invariante: $0 \leq$ `i` $\leq$ `len(A)` y `A[0:i]` está ordenada.

```
1  def pos_primer_mayor(x:int, L:List[int]) -> int:
2      ''' Devuelve la posición del primer número mayor que x en L. '''
3      for j in range(0, len(L)):          O(len(L)) iteraciones
4          if L[j] > x:                        O(1)
5              return j                        O(1)
6      return len(L)                       O(1)
```

$O(len(L)) * \text{máx}(O(1), O(1)) + O(1) = O(len(L))$
Entonces, `pos_primer_mayor(x, L)` es lineal respecto de `len(L)`.

```
1  def insertion_sort(A:List[int]):
2      ''' Ordena (modifica) la lista A de menor a mayor. '''
3      for i in range(0, len(A)):                  O(len(A)) iteraciones
4          x:int = A.pop(i)                            O(len(A))
5          sublista:List[int] = A[0:i]                 O(len(A))
6          j:int = pos_primer_mayor(x, sublista)       O(len(A))
7          A.insert(j, x)                              O(len(A))
```

$O(len(A)) * \Big(O(len(A)) + O(len(A)) + O(len(A)) + O(len(A))\Big) =$
$O(len(A)^2)$. Así, `insertion_sort(A)` es cuadrático respecto de `len(A)`.

Complejidad y Ordenamiento

Algoritmos:

- $O(n^2)$: selection, insertion, bubble (ver la guía).
- $O(n \log n)$: merge sort, heapsort.
- Límite teórico para algoritmos basados en comparaciones: $O(n \log n)$.

Demos y otras yerbas:

- http://www.sorting-algorithms.com/
- http://www.youtube.com/watch?v=MtcrEhrt_K0
- http://www.youtube.com/watch?v=INHF_5RIxTE

Repaso de la clase de hoy:

- Algoritmos de ordenamiento: selection sort, insertion sort.
- Con lo visto, ya pueden resolver hasta el Ejercicio 6 (inclusive) de la Guía de Ejercicios 7.

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