Complejidad Algorítmica: Insertion Sort

Repaso de las reglas de cálculo de órdenes de complejidad:

- **Operaciones simples:** Tienen complejidad constante, $O(1)$. Ejemplos: lecturas y escrituras de variables; operaciones simples de `bool`, `int` y `float`; `len(·)` y `·[·]` para `str` y `List[T]`; `·.append(·)`, `·.pop()` y otras más para listas.[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 $O(h)$, respectivamente, entonces `if CONDICIÓN: CÓDIGO1 else: CÓDIGO2` tiene $O(\text{máx}(f, g, h))$.
- **Ciclo:** Si `CÓDIGO` tiene $O(g)$ y se lo ejecuta $O(f)$ veces, entonces `for ...: CÓDIGO` tiene $O(f) * O(g) = O(f * g)$.

Observación: Un caso particular de la secuencialización es la evaluación de **expresiones compuestas**. Cuando una expresión tiene varias operaciones, debemos identificar los órdenes de complejidad de ellas y sumarlos. Por ejemplo, la expresión `xs[i] == len(xs) + 1` se evalúa con la siguiente secuencia de operaciones: longitud de la lista `xs` ($O(1)$), suma de enteros ($O(1)$), lectura de la variable `i` ($O(1)$), lectura de la posición `xs[i]` ($O(1)$), y comparación de enteros ($O(1)$). Luego, la complejidad de la evaluación completa se calcula de esta forma:

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

Complejidad de insertion sort

Consideremos primero la siguiente función auxiliar `pos_primer_mayor`:

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

La **línea 6** tiene la lectura de las variables `j` y `x`, la lectura de la posición `L[j]`, y una comparación entre enteros; todas $O(1)$. Entonces, esta línea es $O(1) + O(1) + O(1) + O(1) = O(\text{máx}(1, 1, 1, 1)) = O(1)$.

La **línea 7** tiene la lectura de una variable y el retorno de un valor, ambas operaciones de tiempo constante.

La **línea 8** tiene la consulta de la longitud de una lista y el retorno de un valor, ambas $O(1)$. Entonces, esta línea es $O(1) + O(1) = O(\text{máx}(1, 1)) = O(1)$.

El cuerpo del ciclo se ejecuta a lo sumo `len(L)` veces; es decir, tiene $O(\texttt{len(L)})$ iteraciones.

[1] Consultar la complejidad de las operaciones de listas en Python en `https://wiki.python.org/moin/TimeComplexity`.

El cálculo de órdenes de la función `pos_primer_mayor` se desarrolla de esta manera (en verde se indica el cuerpo del ciclo):

$$
\begin{aligned}
&O(\texttt{len(L)}) * O(\text{máx}(1,1)) + O(1) = \\
&O(\texttt{len(L)}) * O(1) \quad\quad\quad + O(1) = \\
&O(\texttt{len(L)} * 1) \quad\quad\quad\quad + O(1) = \\
&O(\texttt{len(L)}) \quad\quad\quad\quad\quad + O(1) = \\
&O(\text{máx}(\texttt{len(L)}, 1) = O(\texttt{len(L)})
\end{aligned}
$$

Entonces, `pos_primer_mayor(x, L)` tiene **complejidad lineal** respecto de `len(L)`.

Consideremos ahora la función `insertion_sort`:

```
1  def insertion_sort(A:List[int]):
2      ''' Requiere: Nada.
3          Devuelve: Nada.
4          Modifica: la lista A tiene los mismos elementos que al principio,
5          pero ahora ordenados de menor a mayor.
6      '''
7      for i in range(0, len(A)):                          O(len(A)) iteraciones
8          x:int = A.pop(i)                                    O(len(A))
9          sublista:List[int] = A[0:i]                         O(len(A))
10         j:int = pos_primer_mayor(x, sublista)               O(len(A))
11         A.insert(j, x)                                      O(len(A))
```

La **línea 8** tiene varias operaciones: la lectura de la variable `i`, que es $O(1)$; la ejecución del método `pop(i)` de listas[2], que es $O(\texttt{len(A)})$; y la escritura en la variable `x`, que es $O(1)$. Entonces, esta línea es $O(1) + O(\texttt{len(A)}) + O(1) = O(\text{máx}(1, \texttt{len(A)}, 1)) = O(\texttt{len(A)})$.

La **línea 9** tiene la lectura de la variable `i`, que es $O(1)$; el cómputo de la sublista, que es $O(\texttt{i})$ (porque la sublista obtenida tiene longitud `i`); y la escritura en la variable `sublista`, que tiene $O(1)$ (esta escritura consiste sólo en que `sublista` apunte a la nueva lista; no se hace una copia). Entonces, esta línea es $O(1) + O(\texttt{i}) + O(1) = O(\texttt{i})$. En el peor caso, `i` vale `len(A)-1`, por lo que podemos acotar superiormente esta complejidad y decir que es $O(\texttt{len(A)})$.

La **línea 10** tiene una invocación a la función `pos_primer_mayor(x, sublista)`, que como vimos tiene $O(\texttt{len(sublista)})$. La lista que pasamos como argumento (`A[0:i]`) tiene longitud `i`. Entonces, esta invocación a `pos_primer_mayor` tiene complejidad $O(\texttt{i})$. La línea 10 también tiene una escritura de la variable `j`, que es $O(1)$. Entonces, esta línea es $O(\texttt{i}) + O(1) = O(\texttt{i})$. Al igual que en la línea anterior, podemos decir que es $O(\texttt{len(A)})$.

La **línea 11** tiene la lectura de las variables `j` y `x`, $O(1)$; y la ejecución del método `insert` de listas, que es $O(\texttt{len(A)})$. Entonces, esta línea es $O(1) + O(1) + O(\texttt{len(A)}) = O(\texttt{len(A)})$.

Es fácil ver que el cuerpo del ciclo se ejecuta $O(\texttt{len(A)})$ veces. El cálculo de órdenes de la función `insertion_sort` puede desarrollarse de la siguiente manera (en verde se indica el cuerpo del ciclo):

$$
\begin{aligned}
&O(\texttt{len(A)}) * \big(O(\texttt{len(A)}) + O(\texttt{len(A)}) + O(\texttt{len(A)}) + O(\texttt{len(A)})\big) = \\
&O(\texttt{len(A)}) * O\big(\text{máx}(\texttt{len(A)}, \texttt{len(A)}, \texttt{len(A)}, \texttt{len(A)})\big) = \\
&O(\texttt{len(A)}) * O(\texttt{len(A)}) = \\
&O(\texttt{len(A)} * \texttt{len(A)}) = O(\texttt{len(A)}^2)
\end{aligned}
$$

Así, concluimos que `insertion_sort(A)` tiene **complejidad cuadrática** respecto de `len(A)`.

[2] No confundir con el método `pop()` de listas, sin argumentos, que tiene complejidad constante.