Introducción a la Programación

Guía Práctica 11
Medidas de análisis de programas

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

1. Cantidad de tiempo de ejecución con distintos tipos de datos, mismos problemas

En Python para medir el tiempo de ejecución podemos usar la biblioteca *time*. Probar el siguiente código:

```
import time

print(time.time()) # devuelve el número de segundos,
# con precisión de microsegundos, que han pasado desde el 1 de Enero de 1970.
```

Para poder medir el tiempo que una función ejecuta, podemos restar el número final del número inicial. Por ejemplo, puedo medir cuánto tiempo tarda la operación de dormir un segundo, debería ser aproximadamente 1. Ejecutar varias veces y probar qué imprime cada vez:

```
import time

inicio = time.time()

# Código a medir
time.sleep(1)                          # Hago que se duerma 1 segundo
# -------------

fin = time.time()
print(fin-inicio)                      # Debería imprimir aproximadamente 1 segundo
```

Ejercicio 1. Vamos a medir el tiempo de distintas funciones que suman ($\sum_0^n i$) usando una función con el siguiente código para medir el tiempo de ejecución de cada una de las implementaciones:

```
import time                                  # Biblioteca de python

def sumatoria_con_tiempos(n : int) -> float:
    inicio : float = time.time()               # time.time = segundos desde 1970
    suma = sumatoria_1(n)                      # Función que nos interesa medir
    fin : float = time.time()

    tiempo_segundos : float = fin - inicio     # Tiempo total (medido en segundos)
    return tiempo_segundos                     # Devolvemos solo el tiempo (porque sí)
```

Notar que la función `sumatoria_con_tiempos` tiene de parámetro de entrada el n que nos sirve para probar la función sumatoria con distintos valores de entrada. Para las siguientes implementaciones de sumatoria:

1.
```
def sumatoria_1(n : int) -> int:
    total : int = 0
    i: int = 1
    while i < n:
      total = total + i
      i = i + 1
    return total
```

2.
```
def sumatoria_2(n : int) -> int:
    total : int = (n * (n-1)) // 2          # // es la división entera.
    return total
```

3.
```
def sumatoria_3(n: int) -> int:             # Versión recursiva (a la Haskell)
    if n == 0:
      return 0
    return n + sumatoria_3(n - 1)
```

- Probar ejecutar varias veces con el mismo valor de entrada. ¿Qué ocurre con los tiempos? ¿Era lo que esperabas?
- Probar ejecutar con distintos valores por ejemplo 10, 1000, 1000000. ¿Qué ocurre con los tiempos? ¿Era lo que esperabas?

Ejercicio 2. Vamos a implementar y comparar los tiempos de buscar el máximo elemento de una secuencia de tamaño n. Para esto debemos generar secuencias de datos aleatorios. Podemos usar random de esta forma:

```
import random

# usamos random.sample()
# para generar una lista con valores aleatorios
def generar_lista(desde, hasta, cantidad):
  res = random.sample(range(desde, hasta), cantidad)
  return res
```

Tener en cuenta que este programa genera una lista. Agregar los datos en una Pila y en una Cola para poder luego comparar con los **mismos valores** cuánto tiempo tarda en buscar el máximo con tres tipos de distintos de datos:

- En una Pila (from queue import LifoQueue as Pila)
- En una Cola (from queue import Queue as Cola)
- En una Lista

1. ¿Tardan lo mismo? ¿Era lo que esperabas?
2. ¿Cuál tarda más? ¿De qué depende que tarde más si todas las secuencias tienen los mismos elementos?
3. Probar los tiempos con las mismas 3 secuencias, pero ahora poner un elemento que sabemos que es el más grande siempre al principio. Por ejemplo, generamos los valores aleatorios entre 1 y 50 y ponemos el (50+1) al principio.

   ```
   import random

   def generar_lista(desde, hasta, cantidad):
     res = random.sample(range(desde, hasta), cantidad)
     res.insert(0, hasta+1)
     return res
   ```

4. Probar los tiempos con las mismas 3 secuencias, pero ahora poner un elemento que sabemos que es el más grande siempre al final. Por ejemplo, generamos los valores aleatorios entre 1 y 50 y ponemos el (50+1) al final.

   ```
   import random

   def generar_lista(desde, hasta, cantidad):
     res = random.sample(range(desde, hasta), cantidad)
     res.append(hasta+1)
     return res
   ```

2. Cantidad de operaciones

Dado que los tiempos dependen del tipo de dato usado, de los algoritmos y del tamaño de la entrada, queremos buscar una forma que nos calcule la cantidad de operaciones para poder comparar sin necesidad de ejecutar el código, qué algoritmo es mejor (tiene menos operaciones para llegar al resultado).

Una forma de medir la "complejidad" de una función, es a partir de contar operaciones elementales (sumas, restas, multiplicaciones, divisiones, asignaciones, condicionales, returns).

Este método no depende de la computadora en donde estamos ejecutando el código. Es un método agnóstico a la computadora.

Ejercicio 3. ★ Vamos a modificar la función `sumatoria_1`, le agregamos que cuente cuántas operaciones hace y nos devuelva el resultado:

```
def sumatoria_con_cuentas(n : int) -> int:
  cant_operaciones : int = 0  # (op. agregada, no cuenta)
  total : int = 0             # Asignación de total a un valor
  cant_operaciones += 1       # (op. agregada, no cuenta)

  i: int = 1                  # Asignación de i a un valor
  cant_operaciones += 1       # (op. agregada, no cuenta)
  while i < n:                # Comparación
    cant_operaciones += 1     # (op. agregada, no cuenta)
    total = total + i         # Asignación de total a un valor y la suma de total con i
    cant_operaciones += 2     # (op. agregada, no cuenta)
    i = i + 1                 # Asignación de i a un valor y la suma de i con 1
    cant_operaciones += 2     # (op. agregada, no cuenta)
  return total, cant_operaciones
```

1. Modificar el código de `sumatoria_2` para que devuelva la cantidad de operaciones.
2. Modificar el código de `sumatoria_3` para que devuelva la cantidad de operaciones.
3. Ejecutar los 3 programas y comparar para mismo parámetro de entrada qué resultado devuelve. Probar con distintos parámetros, por ejemplo: 100, 1000, 10000, 100000.

3. Graficamos los resultados

Con el siguiente código podemos graficar la función de sumatoria con tiempos para poder comparar visualmente qué pasa con la función a medida que el parámetro crece:

```
import matplotlib.pyplot as plt

tiempos = []
for t in range(1000,1000000):
  tiempo = sumatoria_con_tiempos(t)
  tiempos.append(tiempo)

plt.plot(tiempos)
plt.xlabel("tamaño de la entrada")
plt.ylabel("tiempo (segundos)")
plt.title("Experimento tiempos")
plt.show()
```

4. Más ejercicios: cantidad de operaciones

Ejercicio 4. ★

Calcular la cantidad de operaciones T(n) que realiza cada una de las siguientes funciones, siendo n el parámetro de entrada. ¿Cómo es el crecimiento de cada uno de los T(n) con respecto a n (lineal, cuadrático, exponencial, etc.)?

1. ```
   def producto_0(n: int) -> int:
       res: int = 1 * 2
       return res
   ```
2. ```
   def producto_1(n: int) -> int:
       res: int = 1
       i: int = 1
       while i <= n:
           res = res * i
           i = i + 1
       return res
   ```
3. ```
   def producto_2(n: int) -> int:
       res: int = 1
       i: int = 1
       while i <= n:
           res = res * i
   ```
   ```

```
        i = i + 2
    return res
```

4.
```
def producto_3(n: int) -> int:
    res: int = 1
    i: int = 1
    while i <= n:
        j: int = 1
        while j <= n:
            res = res * i * j
            j = j + 1
        i = i + 1
    return res
```

5.
```
def producto_4(n: int) -> int:
    res: int = 1
    i: int = 1
    while i <= n:
        j: int = 1
        while j <= n:
            k: int = 1
            while k <= n:
                res = res * i * j * k
                k = k + 1
            j = j + 1
        i = i + 1
    return res
```

6.
```
def producto_5(n: int) -> int:
    res: int = 1
    i: int = 1
    while i < n:
        res = res * i
        i = i * 2
    return res
```

7.
```
def producto_6(n: int) -> int:
    res: int = 1
    i: int = 1
    while i <= 2**n:
        producto: int = 1
        j: int = 1
        while j <= n:
            if (i // (2 ** (j-1))) % 2 == 1:
                producto = producto * j
            else:
                producto = producto * 1
            j = j + 1
        i = i + 1
        res = res * producto
    return res
```

Ejercicio 5. Para cada uno de los siguientes problemas, calcular T(n) para cada una de las implementaciones planteadas, siendo n el tamaño de la lista s o la cantidad de filas de la matriz m, según corresponda. Comparar entre sí, para cada problema, los valores de T(n) obtenidos para cada implementación.

1. ★

problema maxima_diferencia (in s:$seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ {

requiere: { $True$ }

asegura: { res = a la máxima diferencia entre dos números de s }

}

```
# Función auxiliar
def mayor(x: int, y: int) -> int:
    if x > y:
        res: int = x
    else:
        res: int = y
    return res

# Función auxiliar
def menor(x: int, y: int) -> int:
    if x < y:
        res: int = x
    else:
        res: int = y
    return res

# Función auxiliar
def valor_absoluto(n: int) -> int:
    if n > 0:
        res: int = n
    else:
        res: int = -n
    return res

def maxima_diferencia_v1(s: list[int]) -> int:
    res: int = 0
    i: int = 0
    while i < len(s):
        j: int = 0
        while j < len(s):
            res = mayor(res, valor_absoluto(s[i] - s[j]))
            j = j + 1
        i = i + 1
    return res

def maxima_diferencia_v2(s: list[int]) -> int:
    maximo: int = s[0]
    i: int = 1
    while i < len(s):
        maximo = mayor(maximo, s[i])
        i = i + 1
    minimo: int = s[0]
    i = 1
    while i < len(s):
        minimo = menor(minimo, s[i])
        i = i + 1
    return maximo - minimo

def maxima_diferencia_v3(s: list[int]) -> int:
    maximo: int = s[0]
    minimo: int = s[0]
    i: int = 1
    while i < len(s):
        maximo = mayor(maximo, s[i])
        minimo = menor(minimo, s[i])
        i = i + 1
    return maximo - minimo
```

2. problema suma_de_productos (in s:$seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ {

   requiere: { $True$ }

   asegura: { $res =$ a la suma de los productos de todos los números $s[i]$ y $s[j]$ para todos los índices tales que i $\neq$ j }

}

```
def suma_de_productos_v1(s: list[int]) -> int:
    res: int = 0
    i: int = 0
    while i < len(s):
        j: int = 0
        while j < len(s):
            if i != j:
                res = res + s[i] * s[j]
            j = j + 1
        i = i + 1
    return res // 2 # Cada producto se sumó dos veces

def suma_de_productos_v2(s: list[int]) -> int:
    res: int = 0
    i: int = 0
    while i < len(s):
        j: int = i + 1
        while j < len(s):
            res = res + s[i] * s[j]
            j = j + 1
        i = i + 1
    return res
```

3. ★

problema elementos_en_pos_pares (in s:$seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {

requiere: { $True$ }

asegura: { res = a una secuencia con los números de las posiciones pares de s, en el mismo orden relativo }

}

```
def elementos_en_pos_pares_v1(s: list[int]) -> list[int]:
    res: list[int] = []
    i: int = 0
    while i < len(s):
        if i % 2 == 0:
            res.append(s[i])
        i = i + 1
    return res

def elementos_en_pos_pares_v2(s: list[int]) -> list[int]:
    res: list[int] = []
    i: int = 0
    while i < len(s):
        res.append(s[i])
        i = i + 2
    return res
```

4. problema sumar_cantidad_veces_indice (inout s:$seq\langle\mathbb{Z}\rangle$) {

requiere: { $True$ }

modifica: { s }

asegura: { ($|s| = |s@pre|$) y (para todo i entero, con $0 \leq i < |s|$, $s[i] = \sum_{j=1}^{i} s@pre[i]$) }

}

```
def sumar_cantidad_veces_indice_v1(s: list[int]) -> None:
    i: int = 0
    while i < len(s):
        suma: int = 0
```

```
        j: int = 0
        while j < i:
            suma = suma + s[i]
            j = j + 1
        s[i] = suma
        i = i + 1

def sumar_cantidad_veces_indice_v2(s: list[int]) -> None:
    i: int = 0
    while i < len(s):
        s[i] = s[i] * i
        i = i + 1
```

5. `problema diagonal_principal` (in m:$seq\langle seq\langle \mathbb{Z} \rangle\rangle$) : $seq\langle \mathbb{Z} \rangle$ {
   `requiere:` { esMatriz(m) }
   `requiere:` { $|m| = |m[0]|$ }
   `asegura:` { res = a los elementos de la diagonal principal de m }
   }

```
def diagonal_principal_v1(m: list[list[int]]) -> list[int]:
    res: list[int] = []
    n: int = len(m)
    i: int = 0
    while i < n:
        j: int = 0
        while j < n:
            if i == j:
                res.append(m[i][j])
            j = j + 1
        i = i + 1
    return res

def diagonal_principal_v2(m: list[list[int]]) -> list[int]:
    res: list[int] = []
    n: int = len(m)
    i: int = 0
    while i < n:
        res.append(m[i][i])
        i = i + 1
    return res
```

6. `problema convertir_en_matriz_identidad` (inout m:$seq\langle seq\langle \mathbb{Z} \rangle\rangle$) {
   `requiere:` { esMatriz(m) }
   `requiere:` { $|m| = |m[0]|$ }
   `modifica:` { m }
   `asegura:` { esMatriz(m) }
   `asegura:` { $|m| = |m[0]|$ }
   `asegura:` { m es la matriz identidad }
   }

   Nota: la matriz identidad tiene 1s en la diagonal principal y 0s en el resto de las posiciones.

```
def convertir_en_matriz_identidad_v1(m: list[list[int]]) -> None:
    n: int = len(m)
    i: int = 0
    while i < n:
        j: int = 0
        while j < n:
```

```
            if i == j:
                m[i][j] = 1
            else:
                m[i][j] = 0
            j = j + 1
        i = i + 1

def convertir_en_matriz_identidad_v2(m: list[list[int]]) -> None:
    n: int = len(m)
    i: int = 0
    while i < n:
        j: int = 0
        while j < n:
            m[i][j] = 0
            j = j + 1
        m[i][i] = 1
        i = i + 1
```

7. problema sumar_elementos_matriz (in m:$seq\langle seq\langle \mathbb{Z} \rangle\rangle$) : $\mathbb{Z}$ {
   requiere: { esMatriz(m) }
   asegura: { res = a la suma de los elementos de m }
   }

```
#Función auxiliar
def sumar_elementos_lista(s: list[int]) -> int:
    res: int = 0
    i: int = 0
    while i < len(s):
        res = res + s[i]
        i = i + 1
    return res

def sumar_elementos_matriz_v1(m: list[list[int]]) -> int:
    suma_por_fila: list[int]
    i: int = 0
    while i < len(m):
        suma_por_fila.append(sumar_elementos_lista(m[i]))
        i = i + 1
    return sumar_elementos_lista(suma_por_fila)

def sumar_elementos_matriz_v2(m: list[list[int]]) -> int:
    res: int = 0
    i: int = 0
    while i < len(m):
        res = res + sumar_elementos_lista(m[i])
        i = i + 1
    return res

def sumar_elementos_matriz_v3(m: list[list[int]]) -> int:
    res: int = 0
    i: int = 0
    while i < len(m):
        j: int = 0
        while j < len(m[i]):
            res += m[i][j]
            j = j + 1
        i = i + 1
    return res
```

Ejercicio 6. Para cada uno de los siguientes problemas, buscar el mejor y el peor caso del parámetro de entrada. Calcular T_{mejor}(n) y T_{peor}(n) y comparar ambos valores entre sí, siendo n el tamaño de la lista s, la cantidad de caracteres de texto o la cantidad de filas de m, según corresponda.

1. ★

   problema contar_pares (in s:$seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ {
   requiere: { $True$ }
   asegura: { res = a la suma de los elementos pares de s }
   }

```
def contar_pares(s: list[int]) -> int:
    res: int = 0
    i: int = 0
    while i < len(s):
        if s[i] % 2 == 0:
            res = res + s[i]
        i = i + 1
    return res
```

2. problema suma_hasta_umbral (in s:$seq\langle\mathbb{Z}\rangle$, in umbral: $\mathbb{Z}$) : $\mathbb{Z}$ {
   requiere: { $True$ }
   asegura: { res = a la suma de los elementos de s menores a umbral que aparecen en forma consecutiva al inicio de s }
   }

```
def suma_hasta_umbral(s: list[int], umbral: int) -> int:
    res: int = 0
    i: int = 0
    while i < len(s) and s[i] < umbral:
        res = res + s[i]
        i = i + 1
    return res
```

3. problema contar_minimos (in s:$seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ {
   requiere: { $|s| > 0$ }
   asegura: { res = a la cantidad de apariciones en s del mínimo de s }
   }

```
def contar_minimos(s: list[int]) -> int:
    minimo: int = s[0]
    i: int = 1
    while i < len(s):
        if s[i] < minimo:
            minimo = s[i]
        i = i + 1
    res: int = 0
    i = 0
    while i < len(s):
        if s[i] == minimo:
            res = res + 1
        i = i + 1
    return res
```

4. problema colapsar_repetidos (in texto:$seq\langle\mathsf{Char}\rangle$) : $seq\langle\mathsf{Char}\rangle$ {
   requiere: { $|texto| > 0$ }
   asegura: { res = a texto, eliminando las apariciones consecutivas de un mismo caracter }
   }

```
def colapsar_repetidos(texto: str) -> str:
    res: str = texto[0]
    i: int = 1
    while i < len(texto):
        if texto[i] != texto[i - 1]:
            res = res + texto[i]
        i = i + 1
    return res
```

5. ★

problema sumar_filas_negativas (in m:$seq\langle seq\langle \mathbb{Z} \rangle\rangle$) : $\mathbb{Z}$ {

requiere: { esMatriz(m) }

requiere: { Cada fila de m está compuesta, o bien únicamente de números positivos, o bien únicamente de números negativos }

asegura: { res = a la suma de los elementos negativos de m }

}

```
def sumar_filas_negativas(m: list[list[int]]) -> int:
    res: int = 0
    i: int = 0
    while i < len(m):
        if m[i][0] < 0:
            j: int = 0
            while j < len(m[i]):
                res = res + m[i][j]
                j = j + 1
        i = i + 1
    return res
```