Introducción a nociones de Complejidad Algorítmica

Introducción a la Programación

¿Cuánto tiempo demora mi programa?

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

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

Imaginen que tienen el problema de sumar desde 1 hasta n en dos versiones.

¿Cuánto demoran? ¿Qué versión es más lenta? ¿Por qué? ¿En cualquier compu? ¿De qué depende? ¿Cómo piensan que cambiaría el tiempo en `sumatoria_1` a medida que n aumenta? ¿y en `sumatoria_2`?

¿Cuánto tiempo demora mi programa?

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

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

¿Si tengo que pedir una compu prestada para correr mi programa, por cuánto tiempo lo pido? **¿Podremos saberlo de antemano (sin correr nada)?**

Objetivos de hoy

- Medir el **tiempo de ejecución** de programas en Python.
- Entender el **conteo** de operaciones elementales a nivel práctico e implementativo.
- Graficaremos los experimentos y analizaremos los resultados.
- Explorar qué tipo de **función se ajusta** bien a esos tiempos.

Continuará (con esteroides) en Algoritmos Y Estructuras de Datos (Algo 2 para LCD).

Importancia de poder medir tiempos

Medir el **tiempo de ejecución** de un programa es importante por varias razones:

- **Eficiencia**: Permite evaluar **qué tan rápido se ejecuta un programa**. En aplicaciones donde el tiempo es crítico, como en sistemas en tiempo real, videojuegos, transacciones financieras o análisis de grandes volúmenes de datos, el rendimiento puede ser un factor decisivo.
- **Optimización**: Identificar las partes del código que consumen más tiempo puede ayudar a optimizar el programa. Al saber dónde se están produciendo los cuellos de botella, los desarrolladores pueden concentrarse en mejorar esas áreas específicas.
- **Comparación de Algoritmos**: Para elegir el mejor algoritmo entre varias alternativas, es útil medir el tiempo de ejecución. Un algoritmo puede ser más eficiente que otro en términos de velocidad, lo que es crucial para aplicaciones donde el rendimiento es importante.
- **Eficiencia de Recursos**: Un programa más rápido generalmente consume menos recursos del sistema, como CPU y memoria, lo que puede ser importante en entornos con recursos limitados.

Medición del tiempo de ejecución

```
import time                                        # Biblioteca de python

def sumatoria_con_tiempos(n : int) -> int:
  t0 : float = time.time()                         # time.time = segundos desde 1970
  suma = sumatoria(n)                              # Función que nos interesa medir
  tf : float = time.time()

  tiempo_segundos : float = tf - t0                # Tiempo total (medido en segundos)
  return tiempo_segundos                           # Devolvemos solo el tiempo (porque sí)
```

```
In [49]: sumatoria_con_tiempos(10_000_000)
Out[49]: 1.3811
```

los guiones bajos en python no afectan al número. Solo más fácil de leer

Medición del tiempo de ejecución

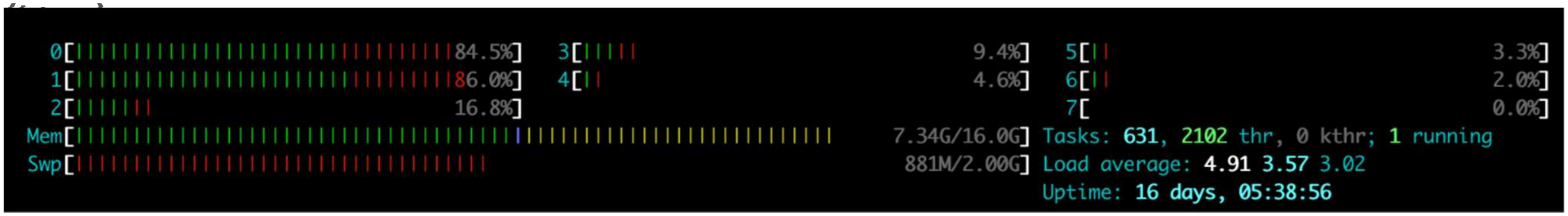
¿Qué ocurre si ejecutamos esta instrucción múltiples veces?

```
In [50]:  sumatoria_con_tiempos(10_000_000)
Out[50]: 1.2316

In [51]:  sumatoria_con_tiempos(10_000_000)
Out[51]: 1.2642

In [52]:  sumatoria_con_tiempos(10_000_000)
Out[52]: 1.4275
```

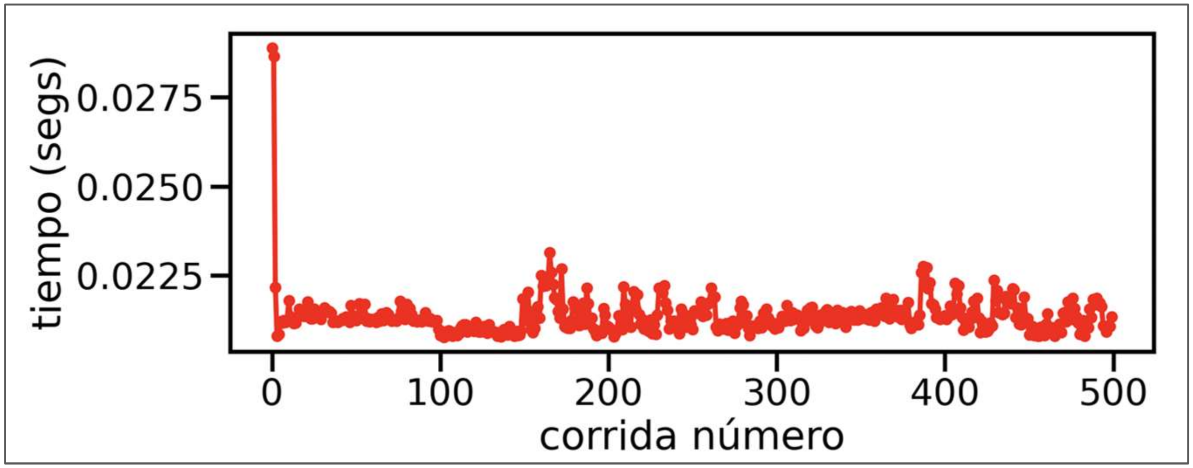
¿Por qué cambia? ⚠ Nuestro programa no es lo único que está corriendo en este momento



Medición del tiempo de ejecución

Gráfico que muestra **500** corridas de `sumatoria_con_tiempos(1_000_000)`

```
tiempos_de_ejecución = []
for i in range(500):
    tiempo_i = sumatoria_con_tiempos(1_000_000)
    tiempos_de_ejecución.append(tiempo_i)
graficar(tiempos_de_ejecución)
```



Las primeras corridas parecen llevar más tiempo, luego se estabiliza
(más en la materia sistemas operativos)

Cantidad de operaciones elementales

Una forma de medir la “complejidad” de una función, es a partir de **contar operaciones elementales** (sumas, restas, multiplicaciones, divisiones, asignaciones, condicionales, returns).
Método **agnóstico** a la computadora. Pero lamentablemente modifica su código:

```python
def sumatoria(n : int) -> int:

  total : int = 0             # Asignación de total a un valor
  i : int = 1                 # Asignación de i a un valor

  while (i < n):              # < comparación

    total = total + i         # Asignación de total a un valor y la suma de total con i
    i = i + 1                 # la suma de i con 1 y la asignación del nuevo valor a i
  return total                # el return
```

```
In [49]: sumatoria_con_cuentas(1_000)
Out[49]: 499500
```

Cantidad de operaciones elementales

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

Método **agnóstico** a la computadora. Pero lamentablemente modifica su código:

```python
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                 # la suma de i con 1 y la asignación del nuevo valor a i
    cant_operaciones += 2    # (op. agregada, no cuenta)
  cant_operaciones += 2       # op. Agregada no cuenta, es para contar la última guarda y el return
  return total, cant_operaciones
```

```
In [49]: sumatoria_con_cuentas(1_000)
Out[49]: (499500, 4999)
```

Cantidad de operaciones elementales

En general mirando el código, podemos hacer un análisis que nos permita conocer la cantidad total de operaciones **sin necesidad** de ejecutar el código.

```
def sumatoria(n : int) -> int:
  total : int = 0
  i : int = 1

  while (i< n):
    total = total + i
    i = i + 1
  return total
```

La cantidad total de operaciones de `sumatoria(1_000)` será:

$T_{sum}(\texttt{1000}) = 2 + 999 * 5 + 1 + \ 1 = \mathbf{4{,}999}$

¿Cuántas operaciones haremos si ahora lo corremos con `100_000_000`?

$T_{sum}(\texttt{100\_000\_000}) = 4 + (100{,}000{,}000 - 1) * 5 = 499{,}999{,}999$

Cantidad de operaciones elementales

En general mirando el código, podemos hacer un análisis que nos permita conocer la cantidad total de operaciones **sin necesidad** de ejecutar el código.

```
def sumatoria(n : int) -> int:
  total : int = 0
  i : int = 1

  while (i< n):
    total = total + i
    i = i + 1
  return total
```

Podemos entonces deducir una fórmula general, que depende de los parámetros de la función:

La cantidad total de operaciones de `sumatoria(n)` será:

$$T_{sum}(\mathbf{n}) = 4 + (\mathbf{n}\text{-}1) * 5$$

Cantidad de operaciones elementales

En general mirando el código, podemos hacer un análisis que nos permita conocer la cantidad total de operaciones **sin necesidad** de ejecutar el código.

```
def f(n : int) -> int:
  i = 0
  j = 1000
  while( i < n ):
    i = i + 1
    j = j * 2
  return j
```

3 **Ejercicio, contar las operaciones elementales de esta función**

$$T_f(\mathbf{n}) = \ ??$$

Cantidad de operaciones elementales

En general mirando el código, podemos hacer un análisis que nos permita conocer la cantidad total de operaciones **sin necesidad** de ejecutar el código.

```
def g(n : int) -> int:
  i = 0
  j = 1000
  while( i < n ):
    j = j + sumatoria(n)
    i = i + 1
  return j
```

Lo que sea que lleve computar sumatoria(n)

5 = Una guarda + dos asignaciones + dos sumas

1 = Última evaluación de la guarda

1 = Return

$$T_g(\mathbf{n}) = 2 + n * (T_{sum}(\mathbf{n}) + 5) + 1 + 1$$

$$= 2 + n * (4 + (\mathbf{n}\text{-}1)*5 + 5) + 2$$

$$= 4 + 4n + 5\mathbf{n^2}$$

Notar que si sumatoria hubiera sido llamada sobre i o j, sería más complicado contar (pero se puede). Ejercicio → sumatoria(i)

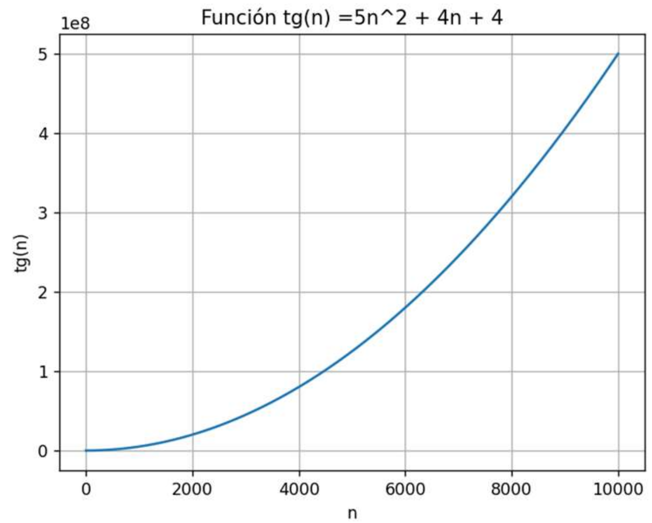
Cantidad de operaciones elementales

En general mirando el código, podemos hacer un análisis que nos permita conocer la cantidad total de operaciones **sin necesidad** de ejecutar el código.

```
def g(n : int) -> int:
  i = 0
  j = 1000
  while( i < n ):
    j = j + sumatoria(n)
    i = i + 1
  return j
```

¿Qué piensan qué ocurriría si medimos el tiempo de ejecución de esta función?

La **"pinta"** se mantendría. Contar operaciones es un buen aproximador a cómo aumentaría el tiempo de ejecución.



$$\mathrm{TiempoReal}_g(\mathbf{n}) \approx c * T_g(\mathbf{n})$$

donde **c** es una constante que depende de la computadora, de otros procesos corriendo al mismo tiempo, etc.

Análisis Complejidad Algorítmica

El análisis del tiempo de ejecución **sin necesidad de correr el código**, conocido como **análisis de complejidad algorítmica**, también es fundamental:

- **Independencia del Hardware:** El análisis teórico de la complejidad (gran parte de Algo 2) permite estimar el rendimiento del programa sin depender del hardware específico. Esto proporciona una evaluación más universal del rendimiento del algoritmo.
- **Predicción del Comportamiento Escalable:** Entender cómo se comporta un algoritmo a medida que crecen los datos de entrada.
- **Detección de Ineficiencias:** Sin necesidad de implementar y ejecutar el código, se pueden detectar ineficiencias potenciales en la fase de diseño (Algo 2).
- **Guía para la Implementación:** Un buen análisis algorítmico puede guiar la implementación, ayudando a los desarrolladores a elegir estructuras de datos y métodos que sean más eficientes para el problema en cuestión (Algo 2).

Para poder entender bien estos conceptos, hay que saber contar bien operaciones.

Contando operaciones con listas

```python
def reverso(s : list[int]) -> list[int]:
  i = 0
  res = []
  while( i < len(s) ):
    elem = s[len(s) - i - 1]
    res.append(elem)
    i = i + 1
  return res
```

$T_{\text{reverso}}(\mathbf{n}) = ??$ **¿Quién es n?**

Generalmente, $\mathbf{n}$ hace referencia al tamaño de un contenedor (en este caso listas).

Si fuera diccionario podría ser con respecto a la cantidad de claves o con respecto a la cantidad de valores.

Contando operaciones con listas

```python
def reverso(s : list[int]) -> list[int]:
  i = 0
  res = []
  while( i < len(s) ):
    elem = s[len(s) - i - 1]
    res.append(elem)
    i = i + 1
  return res
```

Ejercicio: Identificar de dónde viene cada término en esta suma.

$T_{reverso}(\mathbf{n})$ en donde $\mathbf{n == |s|}$:

$$\overset{a}{1} + \overset{b}{1} + \sum_{i=0}^{n-1} \left(\overset{d}{T_{len}(\mathbf{n})} + \overset{e}{1} + \overset{f}{T_{[\cdot]}(n)} + \overset{g}{T_{len}(\mathbf{n})} + \overset{h}{1} + \overset{i}{1} + \overset{j}{1} + \overset{k}{T_{append}(\mathbf{i})} + \overset{l}{2}\right) + \overset{m}{T_{len}(n)} + \overset{n}{1} + \overset{ñ}{1}$$

Spoiler: en Algo2 verán cómo obtener los T faltantes y simplificar la fórmula. **En esta materia alcanza con plantear el término** (y simplificar un pelín): $T_{reverso}(\mathbf{n}) = 4 + T_{len}(n) + \sum_{i=0}^{n} (6 + 2*T_{len}(\mathbf{n}) + T_{[\cdot]}(n) + T_{append}(\mathbf{i}))$

Contando operaciones con listas

```
def concatenar(s1 : list[int], s2 : list[int]) -> list[int]:
  i = 0
  j = 0
  res = []
  l1 = len(s1)
  l2 = len(s2)
  while ( i < l1)
    res.append(s1[i])
    i = i + 1
  while ( j < l2)
    res.append(s2[j])
    j = j + 1
  return res
```

¿Por qué este +n?

$$T_{concat}(\mathbf{n}, \mathbf{m}) = 5 + T_{len}(n) + T_{len}(m) + \sum_{i=0}^{n-1} (1 + T_{append}(i) + T_{[\cdot]}(\mathbf{n}) + 2) + 1 + \sum_{j=0}^{m-1} (1 + T_{append}(j + \mathbf{n}) + T_{[\cdot]}(\mathbf{m}) + 2) + 1 + 1$$

en donde $n == |s1|$ **y** $m == |s2|$

“buenas” vs “malas” entradas

```
def buscar(s : list[int]) -> bool:
  i = 0
  while( i < len(s) ):
    if (s[i] == 100):
      return True
    i = i + 1
  return False
```

$T_{buscar}(\mathbf{n})$ en donde **n == |s|:** **???**

¿La cantidad de operaciones de `buscar([1,100,5,10])` **es igual a** `buscar([1,2,5,10])`**?**

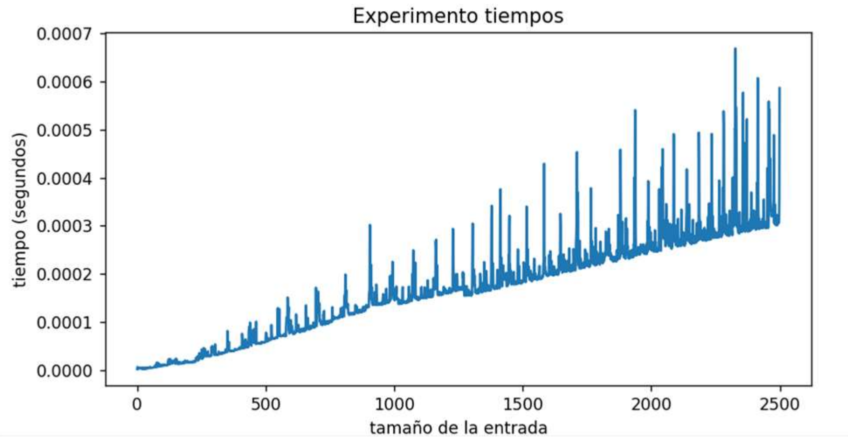
Muchas veces la cantidad de operaciones **depende, no sólo del tamaño**, sino también de los **valores propiamente dichos**. Ya que estamos estimando costos, seremos cautelosos y siempre pensaremos **“cuántas operaciones en el peor caso”** (es decir, ser pesimistas). Por ello:

$T_{buscar}(\mathbf{n}) = T_{buscar\_peor\_caso}(\mathbf{n}) = 1 + \mathbf{n} * (T_{len}(\mathbf{n}) + 1 + T_{[.]}(\mathbf{n}) + 1 + 1 + 1) + 1 + T_{len}(\mathbf{n}) + 1$

Notar que en esta cuenta no consideramos entrar por la rama True ya que no sería el peor caso.

Gráficos

Como vimos en casos anteriores, devolver un número de operaciones concreta puede ser muy complicado. Sin embargo nos interesaría entender **cómo crece** la cantidad de operaciones a **medida que crece el input**. Para ello, una opción interesante es **graficar** los tiempos de ejecución.



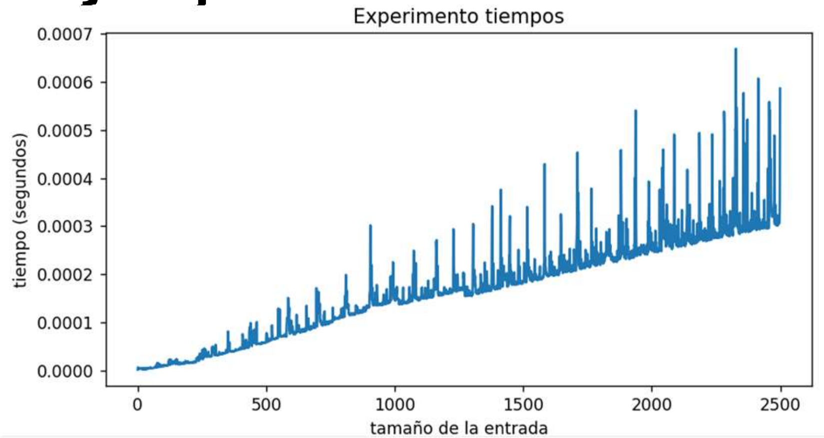

```
import time

def tiempos_de_reverso(max):
  tiempos = []
  i = 0
  while (i < max):
    t0_i : float = time.time()
    reverso(list(range(i)))
    tf_i : float = time.time()
    tiempo_i : float = tf_i - t0_i
    tiempos.append(tiempo_i)
    print(i)
    i += 1
  return tiempos
```

```
import matplotlib.pyplot as plt

tiempos = tiempos_de_reverso(2500)
plt.plot(tiempos)
plt.xlabel("tamaño de la entrada")
plt.ylabel("tiempo (segundos)")
plt.title("Experimento tiempos")
plt.show()
```

Ejemplo Lineal



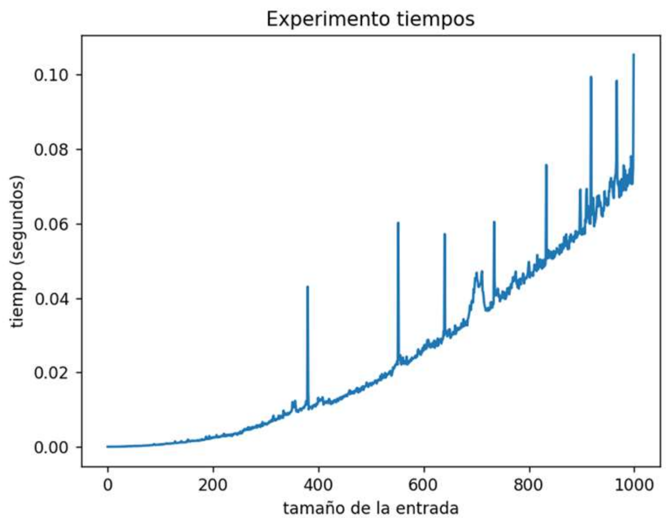

```python
def reverso(s : list[int]) -> list[int]:
  i = 0
  res = []
  while( i < len(s) ):
    elem = s[len(s) - i - 1]
    res.append(elem)
    i = i + 1
  return res
```

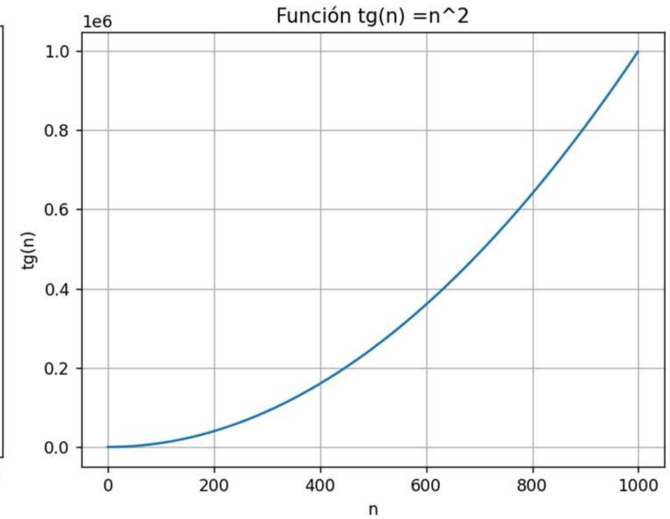
$$T_{reverso}(\mathbf{n}) = 4 + T_{len}(n) + \sum_{\mathbf{i=0}}^{\mathbf{n-1}} (6 + 2*T_{len}(\mathbf{n}) + T_{[\cdot]}(n) + T_{append}(\mathbf{i}))$$

En este gráfico se ven cosas interesantes. Algunas serán fáciles de entender mirando el código, para otras, necesitaremos conocimiento de Algo 2 y materias del área de sistemas (por manejo de memoria, recolector de basura del sistema, etc)

Ejemplo Cuadrático

```
def suma_matriz(matriz: list[list[int]]) -> int:
    total = 0
    i = 0
    while i < len(matriz):
        j = 0
        while j < len(matriz[i]):
            total += matriz[i][j]
            j += 1
        i += 1
    return total
```





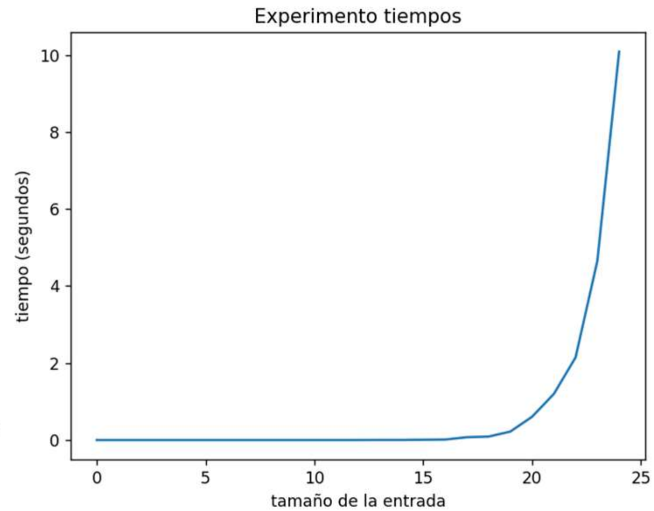
$T_{suma\_matriz}(\mathbf{n}) = 2 + 1 + T_{len}(n) + 1 + \sum_{i=0}^{n-1} (1 + T_{len}(\mathbf{n}) + 1 + 2 + T_{[\cdot]}(n) + T_{len}(\mathbf{m}) + \sum_{j=0}^{m-1} (T_{[\cdot]}(n) + T_{len}(\mathbf{m}) + 1 + T_{[\cdot]}(n) + T_{[\cdot]}(m) + 1 + 1 + 2) \approx C + K*n*m \approx C + Kn^2$ (si es una matriz cuadrada)

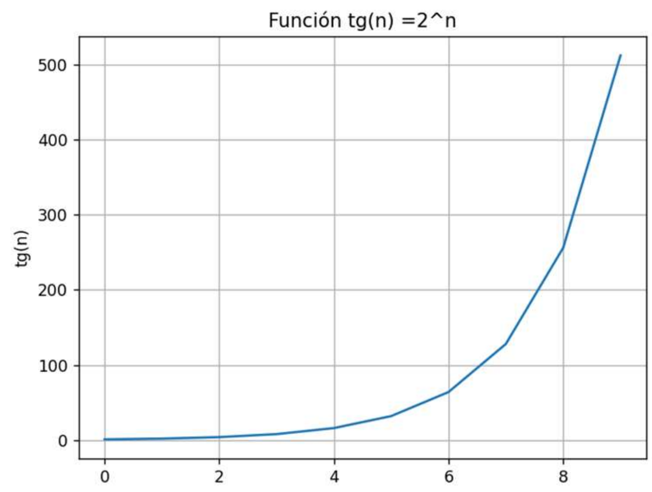
Ejemplo Exponencial

```
def generar_combinaciones(lista):
    combinaciones = [[]]  # Empezamos con el subconjunto vacío
    i:int = 0
    # Recorremos cada elemento en la lista
    while (i < len(lista)):
        # Para cada combinación existente, creamos una nueva agregando el e
        nuevas_combinaciones = []
        j:int = 0
        while (j < len(combinaciones)):
            nuevas_combinaciones.append(combinaciones[j] + [lista[i]])
            j += 1

        # Unimos las combinaciones previas con las nuevas
        combinaciones.extend(nuevas_combinaciones)
        i += 1
    return combinaciones
```

$$T_{generar\_combinaciones}(n) \approx C + \sum_{i=0}^{n-1} K*2^{i}$$





Resumen

- Vimos cómo medir el **tiempo de ejecución** de programas en Python: $\mathrm{TiempoReal(n)}$.
- Vimos cómo se define el **conteo de operaciones elementales** a nivel implementativo y también a nivel teórico: $\mathrm{T(n)}$.
- Graficamos experimentos y analizar los resultados.

Contar bien será clave para el análisis de programas en materias posteriores.
Aprovechen esta materia para aprender a hacerlo lo más rigurosamente posible.

FIN