



EL1022, Algoritmia

# Práctica 1: Python y costes

# Introducción

- ▶ Un elemento fundamental para evaluar un algoritmo es su coste, tanto espacial como temporal
- ▶ En esta práctica veremos distintos algoritmos para algunos problemas sencillos y calcularemos sus costes
- ▶ Para comprobar experimentalmente los costes y también para repasar Python implementaremos esos algoritmos y mediremos los tiempos con una serie de ficheros de prueba
- ▶ Empezaremos con el esquema general que seguirán nuestros programas para facilitar las pruebas

# Esquema general

Nuestros programas seguirán un esquema similar a este:

```
#!/usr/bin/env python3
import sys
from typing import TextIO

def read_data(f: TextIO) -> Data:
    # Leer del fichero f

def process(data: Data) -> Result:
    # Hacer cosas

def show_result(result: Result):
    # Escribir el resultado

if __name__ == "__main__":
    data0 = read_data(sys.stdin)
    result0 = process(data0)
    show_result(result0)
```

# Tipos en Python

- ▶ No es necesario pero sí conveniente especificar tipos en Python
- ▶ Sirven de documentación
- ▶ Permiten al IDE encontrar *bugs* y le ayudan a completar código y a refactorizar
- ▶ No afectan al rendimiento del programa

# Tipos en Python (2)

- ▶ Son objetos Python (se pueden asignar a variables):

```
URL = str
Page = str
def download(url: URL) -> Page:
    # ...
```

- ▶ Se puede usar type para crear alias de los tipos:

```
type URL = str
type Page = str
def download(url: URL) -> Page:
    # ...
```

- ▶ type tiene también algunas ventajas adicionales

# Tipos básicos

- ▶ Se corresponden con las clases (predefinidas o definidas por el usuario)
- ▶ Clases predefinidas: `bool`, `int`, `float`, `str`, `None`

# Tipos estructurados

- ▶ Disponibles directamente:
  - Tuplas: tuple[int, int, bool]
  - Listas: list[float]
  - Diccionarios: dict[str, int]
  - Conjuntos: set[str]
  - Tipos union: int | float
- ▶ Importados de typing:
  - Funciones: Callable[[str, int], char]
  - Opcional: Optional[char] (equivalente a char | None)

# Ejercicio 1: Primer programa

- ▶ Nuestro primer programa, `suma.py`, lee los datos a una lista y los muestra su suma por pantalla
- ▶ Vamos a leer los datos de la entrada estándar
- ▶ Tipos:

```
type Data = list[int]
```

```
type Result = int
```

# Lectura de datos

```
def read_data(f: TextIO) -> Data:  
    # En l tenemos una cadena por linea:  
    lines = f.readlines()  
  
    # Transformamos cada linea en un entero:  
    return [int(line) for line in lines]
```

# Lectura de ficheros

- ▶ Hay varias maneras de leer un fichero en Python, nosotros veremos la lectura de ficheros de texto línea a línea

- ▶ Abrimos un fichero con open:

```
f = open("nombreFichero")
```

- ▶ Leemos una línea con readline():

```
l = f.readline()
```

- ▶ Leemos las líneas que quedan con readlines():

```
ls = f.readlines()
```

- ▶ La entrada estándar es sys.stdin

# Expresiones generatrices

- ▶ La lista que transforma las líneas en enteros utiliza una *expresión generatriz*:

```
return [int(line) for line in lines]
```

- ▶ Es equivalente a un bucle:

```
ints = []
for line in lines:
    ints.append(int(line))
return ints
```

- ▶ Podemos usar cualquier tipo de expresión:

```
squares = [x*x for x in range(1, n)]
```

# Proceso

```
def process(data: Data) -> Result:  
    # La función sum suma los elementos de un iterable  
    return sum(data)
```

# Escritura

```
def show_result(result: Result):  
    print(result)
```

# Principal

```
if __name__ == "__main__":
    data0 = read_data(sys.stdin)
    result0 = process(data0)
    show_result(result0)
```

# Ficheros de prueba

- ▶ En el aula virtual hay un fichero .tgz con varios ficheros de prueba:
  - El fichero `nums<n>` tiene  $n$  números aleatorios del 0 al 1000
  - El fichero `dnums<n>` tiene  $n$  números distintos y aleatorios del 0 al 1000000
- ▶ Por ejemplo, para sumar los números de `nums/nums10` hacemos:

```
python3 suma.py < nums/nums10
```

## Ejercicio 2: Mínimo de la lista

- ▶ Vamos a escribir un programa, `minimo.py`, que encuentre el mínimo de los valores leídos
- ▶ Seguiremos el esquema
- ▶ Recorremos la lista comparando cada elemento con el mínimo provisional
- ▶ ¿Cuál es el coste?

# Implementación

- ▶ Los tipos son

```
type Data = list[int]
type Result = int
```

- ▶ La función `read_data` no cambia
- ▶ La función `process`:

```
def process(data: Data) -> Result:
    m = data[0]
    for num in data:
        if num < m:
            m = num
    return m
```

# Implementación (2)

- ▶ La función show\_result:

```
def show_result(result: Result):
    print(result)
```

- ▶ Principal:

```
if __name__ == "__main__":
    data0 = read_data(sys.stdin)
    result0 = process(data0)
    show_result(result0)
```

# Medida de tiempos

- ▶ Podemos cronometrar el tiempo de una ejecución con `time`:  
`time python minimo.py < nums/nums1000`
- ▶ Prueba a ver cuánto tarda en ejecutarse `minimo.py` con  
`nums/nums10`, `nums/nums1000` y `nums/nums1000000`
- ▶ ¿Son consistentes los tiempos con el coste teórico?

# Medida de tiempos con Python

- ▶ Necesitamos eliminar tiempos ajenos a la función process
- ▶ Podemos usar la biblioteca time de Python
- ▶ Aprovecharemos la estructura de nuestro programa

# Medida de tiempos con Python (2)

```
#!/usr/bin/env python3

import time
from minimo import *

for n in [10, 100, 1000, 10000, 100000, 1000000]:
    test = f"nums/nums{n}"
    f = open(test)
    data = read_data(f)
    t0 = time.process_time()
    result = process(data)
    t1 = time.process_time()
    print (f"{test+':':18} {t1-t0:f}")
```

# Ejercicio 3: Varianza

- ▶ Podemos calcular la varianza de una lista de números  $\{x_1, \dots, x_n\}$  con la fórmula

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2,$$

donde  $\bar{x}$  es la media de los  $x_i$ :

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i,$$

- ▶ Los tipos serán:

```
type Data = list[int]
type Result = float
```

# Primera aproximación

- ▶ Escribimos una función auxiliar para calcular la media:

```
def average(nums: list[int]) -> float:  
    return sum(nums)/len(nums)
```

- ▶ Y nuestro process:

```
def process(data: Data) -> Result:  
    s = 0  
    for num in data:  
        s += (num - average(data)) ** 2  
    return s/len(data)
```

# Costes

- ▶ ¿Hay mejor y peor caso?
- ▶ ¿Cuáles son los costes?
- ▶ Cronometra los tiempos con los ficheros `nums*` hasta `nums10000`

# Costes (2)

- ▶ Podemos reducir fácilmente el coste con una variable auxiliar
- ▶ ¿Cuál es el nuevo coste?
- ▶ Cronometra con todos los ficheros `nums*`

# Ejercicio 4: Repetidos

- ▶ Queremos hacer una función que devuelva True si en la lista hay elementos repetidos y False en caso contrario

- ▶ Los tipos son:

```
type Data = list[int]
```

```
type Result = bool
```

- ▶ Primera aproximación:

- Provisionalmente, `repeated` es False
- Recorrer la lista y comparar cada elemento con los siguientes
- Si alguna comparación es cierta, cambiar `repeated`

# Implementación

```
def process(data: Data) -> Result:  
    repeated = False  
    for i in range(len(data)):  
        for j in range(i+1, len(data)):  
            if data[i] == data[j]:  
                repeated = True  
    return repeated  
  
def show_result(result: Result):  
    print("No hay repetidos" if not result  
          else "Hay repetidos")
```

# Costes

- ▶ ¿Hay mejor y peor caso?
- ▶ ¿Cuál es el coste temporal?
- ▶ Cronometra los tiempos con los ficheros `nums*` hasta `nums10000`

## Ejercicio 5: Mejoramos repetidos

- ▶ Podemos interrumpir el bucle en cuanto encontremos algún repetido
- ▶ Al encontrar el primer repetido, podemos hacer `return True`
- ▶ ¿Cómo afecta a los costes?
- ▶ Implementa la nueva versión y cronometra con los ficheros `nums*` y `dnums*` (hasta 10000)

# Ejercicio 6: Preproceso

- ▶ Otra posibilidad es ordenar primero la lista
- ▶ ¿Cuáles son los costes ahora?
- ▶ Implementa y cronometra

# Ejercicio 7: Funciones predefinidas

- ▶ Normalmente, las funciones predefinidas son más eficientes
- ▶ Pero los costes asintóticos siguen importando
- ▶ Nuestro bucle interno se puede sustituir por un `in`

```
def process(data: Data) -> Result:  
    for i in range(len(data)):  
        if data[i] in data[i+1:]:  
            return True  
    return False
```

# Costes

- ▶ Calcula los nuevos costes
- ▶ Cronometra
- ▶ Prueba a ejecutarlo con `dnums100000`

# Ejercicio 8: Uso de conjuntos

- ▶ Podemos cambiar la estrategia y usar conjuntos:
  - Creamos un conjunto vacío, `seen`
  - Si el elemento actual está en `seen`, hay repetidos
  - Si no, lo guardamos en `seen` y pasamos al siguiente
- ▶ ¿Cuáles son los costes ahora?
- ▶ ¿Cuál es el coste espacial?

# Conjuntos

## ▶ Creación:

- `set()` crea el conjunto vacío
- `set(l)` crea un conjunto con los elementos de `l`
- `{a,b,c}` crea un conjunto con los elementos `a`, `b` y `c`.

## ▶ Operaciones:

Método	Operador	Significado
<code>s.add(a)</code>	—	añade un elemento
—	<code>a in s</code>	pertenencia
—	<code>a not in s</code>	no pertenencia
<code>s.union(t)</code>	<code>s   t</code>	unión
<code>s.intersection(t)</code>	<code>s &amp; t</code>	intersección
<code>s.difference(t)</code>	<code>s - t</code>	diferencia

# Implementación

- ▶ Una implementación:

```
def process(data: Data) -> Result:  
    seen = set()  
    for n in data:  
        if n in seen:  
            return True  
        seen.add(n)  
    return False
```

- ▶ Cronometra usando todos los conjuntos de prueba

## Ejercicio 9: Sumas diferentes

- ▶ Queremos saber cuántos valores distintos se pueden conseguir sumando subconjuntos de los números leídos
- ▶ Por ejemplo con los números  $(2, 2, 3, 5)$  podemos conseguir nueve sumas:  $(2, 3, 4, 5, 7, 8, 9, 10, 12)$

# Algoritmo

- ▶ Podemos utilizar un conjunto para almacenar las sumas que llevamos hasta el momento
- ▶ Cuando analizamos un nuevo número tenemos que añadir la suma del número con todos los anteriores así como el propio número:

```
def process(nums: Data) -> Result:  
    sums = set()  
    for num in nums:  
        for s in list(sums):  
            sums.add(s+num)  
        sums.add(num)  
    return len(sums)
```

# Costes

- ▶ ¿Cuál es el mejor y el peor caso?
- ▶ ¿Cuáles son los costes temporales y espaciales en cada caso?
- ▶ Ejecuta el programa con `time` para `nums10`, `nums100` y `dnums10`
- ▶ ¿Sería posible ejecutarlo con `dnums100`?

# Ejercicio 10: Moda

- ▶ La *moda* de una lista es el elemento que más veces aparece
- ▶ Escribe un programa que calcule la moda de los números leídos
- ▶ Calcula los costes y comprueba tu estimación con los ficheros num\* y dnum\*
- ▶ ¿Puedes conseguir un coste lineal? Pista: usa diccionarios

# Diccionarios

## ► Creación:

- `{}` → diccionario vacío
- `{k1: v1, k2: v2, ..., kn: vn}` → diccionario que asocia a  $k_i$  el valor  $v_i$

## ► Operaciones:

- `d[k]` → devuelve el valor asociado a la clave  $k$
- `d[k] = v` → asocia el valor  $v$  a la clave  $k$
- `k in d` → comprueba si la clave  $k$  está en el diccionario  $d$

## ► Recorridos:

- `for k in d:` → recorre las claves del diccionario
- `for v in d.values():` → recorre los valores del diccionario
- `for k, v in d.items():` → recorre simultáneamente claves y valores