EI1022, 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, implementaremos esos algoritmos, con lo que repasaremos Python, 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 deberían seguir un esquema similar a este:

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

import sys

def read_data(f):
    # Leer del fichero f

def process(data):
    # Hacer cosas

def show_results(results):
    # Escribir los resultados

if __name__ == "__main__":
    data = read_data(sys.stdin)
    results = process(data)
    show_results(results)
```

Ejercicio 1: Primer programa

- Vamos a leer los datos de la entrada estándar
- Nuestro primer programa, simplemente lee los datos a una lista y los muestra por pantalla, uno por línea
- No tendremos `process`

Lectura de datos

```
def read_data(f):
    # En l tenemos una cadena por línea:
    lines = f.readlines()

    # Transformamos cada línea 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()
```

- Podemos leer todas 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)]
```

Escritura

```
def show_results(nums):
    # Recorremos las listas con el bucle for
    for num in nums:
        print(num)
```

Principal

```
if __name__ == "__main__":
    nums = read_data(sys.stdin)
    show_results(nums)
```

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 listar los números de `nums/nums10` hacemos:

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

Ejercicio 2: Mínimo de la lista

- Vamos a escribir un programa que encuentre el mínimo de los valores leídos
- Seguiremos el mismo esquema pero ahora añadiremos un procesamiento de los datos
- Recorreremos la lista elemento a elemento y compararemos con el mínimo
- La lectura no cambia
- ¿Cuál es el coste?

Procesamiento

```
def process(nums):
    m = nums[0]
    for num in nums:
        if num < m:
            m = num
    return m
```

Escritura y principal

- Escritura:

```
def show_results(m):
    print(m)
```

- Principal:

```
if __name__ == "__main__":
    nums = read_data(sys.stdin)
    m = process(nums)
    show_results(m)
```

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)
    nums = read_data(f)
    t0 = time.perf_counter()
    m = process(nums)
    t1 = time.perf_counter()
    print (f"{test+':':18} {t1-t0:f}")
```

Ejercicio 3: Varianza

- Podemos calcular la varianza de una lista de números $x_1, \ldots, x_n$ como

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

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

Primera aproximación

- Escribimos una función para la media:

```
def average(nums):
    return sum(nums)/len(nums)
```

- Y nuestro `process`:

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

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
- 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):
    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):
    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
- Añade un `break` después de `repeated = True`
- ¿Cómo afecta a los costes?
- Implementa la nueva versión y cronometra con los ficheros `nums*` y `dnums*` (hasta 10000)

Nueva mejora

- Podemos mejorar el mejor caso si hacemos `return` al encontrar el primer repetido
- ¿Cuáles son los nuevos costes?
- Implementa y cronometra

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):
    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
`s.add(a)`	—	añade un elemento
—	`a in s`	pertenencia
—	`a not in s`	no pertenencia
`s.union(t)`	`s | t`	unión
`s.intersection(t)`	`s & t`	intersección
`s.difference(t)`	`s - t`	diferencia

Implementación

- Una implementación:

```
def process(data):
    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):
    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 `ki` el valor `vi`
- Operaciones:
  - `a in d`: test de pertenencia
  - `d[k]`: devuelve el valor asociado a la clave `k`
  - `d[k] = v`: cambia el valor asociado a la clave `k`
- 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