UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA
GRADO EN CIENCIA E INGENIERÍA DE DATOS

Diseño y Análisis de Algoritmos
Práctica 1: Búsqueda de Pares con Suma Objetivo - Implementaciones con listas y conjuntos

16 de septiembre de 2024

Dada una lista de n números enteros $L = [s_1, s_2, s_3, \ldots, s_n]$ ordenados y no necesariamente consecutivos, y un número objetivo x, se deben encontrar todos los pares de números en dicha secuencia cuya suma sea igual a x. Si consideramos $L = [1, 2, 3, 4, 5, 6]$ y el objetivo $x = 7$, los pares que suman x en L serían:

$$(1, 6), (2, 5), (3, 4)$$

Se debe tener en cuenta que los pares duplicados, como $(6, 1)$, no deben considerarse. Puede ocurrir que no todos los números de L se utilicen para formar pares, pero aún así se encuentran soluciones que suman el objetivo. Por ejemplo, si consideramos $L = [1, 2, 3, \ldots, 10]$ y el número objetivo $x = 15$, los pares que suman x serían:

$$(5, 10), (6, 9), (7, 8)$$

Por otro lado, si consideramos $L = [1, 3, 5]$ y el número objetivo $x = 20$, no hay ningún par de números cuya suma sea igual a x.

En los siguientes pseudocódigos se presentan dos versiones del algoritmo que permiten resolver este problema. La primera utiliza estrictamente listas, mientras que en la segunda se hace uso de conjuntos.

Búsqueda de Pares con Suma Objetivo y Listas

```
function SearchPairsWithLists (L[1..n], target):
pairs := [ ]; #a list
for i := 1 to n:
    for j := i+1 to n:
        if L[i]+L[j] = target:
            if (L[i], L[j]) not in pairs:
                append (L[i], L[j]) to pairs;
return pairs;
```

Búsqueda de Pares con Suma Objetivo y Conjuntos

```
function SearchPairsWithSets (L[1..n], target):
pairs := {}; # create an empty set
seen := {}; # create an empty set to store the numbers that have been read
for i:=1 to n:
    c := target - L[i];
    if c in seen:
        add (min(L[i],c), max(L[i], c)) to the pairs set;
    add L[i] to seen;
return pairs;
```

Sobre listas y sets Las listas y los conjuntos son dos tipos de datos estándar y ampliamente utilizados en Python. Cada uno de ellos tiene características particulares en cuanto a eficiencia y estructura interna, lo que influye en el rendimiento de diferentes operaciones como la búsqueda, la inserción o la eliminación de elementos, que podemos explotar de forma distinta para distintos problemas. En la wiki oficial de Python[1] podemos conocer las complejidades de las principales operaciones para estos y otros tipos de datos complejos, como los diccionarios.

[1] https://wiki.python.org/moin/TimeComplexity

En esta práctica, exploraremos cómo varía el rendimiento de encontrar pares de números que suman un valor objetivo utilizando dos implementaciones distintas basadas en listas y conjuntos.

Se pide:

1. Implementar en Python los algoritmos propuestos en los pseudocódigos.
2. Validar que los algoritmos funcionan correctamente con los tres ejemplos ilustrados al principio del enunciado. Debéis implementar una función separada test y hacer uso del statement *assert* en Python para comprobar que la salida es correcta. Analizad si quedan sin probar casos límite y, si es el caso, incluid ejemplos adicionales para ellos también.
3. Determinar los tiempos de ejecución para los siguientes tamaños de entrada: $50, 100, 200, 400, 800, 1600, 3200$ para las dos versiones propuestas del algoritmo. Podéis tomar como base el código de la figura 1 para obtener la hora del sistema. Hay que tener en cuenta que la función `time.time_ns` está disponible desde la versión 3.7 y las funciones `time.process_time` y `time.perf_counter` desde la versión 3.3. Para la generación de las entradas a los algoritmos, podéis tomar como base el código indicado en la figura 2.
4. Analizar los resultados obtenidos realizando una comprobación empírica de la complejidad teórica (figura 3). Asimismo, se realizará una comprobación empírica utilizando una cota ligeramente subestimada y otra sobreestimada para cada algoritmo. Como pista, para la implementación del algoritmo *SearchPairsWithLists*, la cota ajustada es $\mathcal{O}(n^2)$, mientras que como cota inferior y superior se podrían considerar, por ejemplo, $\mathcal{O}(n \log n)$ y $\mathcal{O}(n^{2,2})$, respectivamente.

Nota: Se recomienda consultar el seminario sobre técnicas de verificación empírica de la complejidad, donde se explica cómo realizar mediciones de manera correcta para tiempos pequeños, utilizando en este caso $K = 1000$ (donde K es el número de repeticiones necesarias). Para esta práctica, se considera un tiempo pequeño, y por lo tanto potencialmente impreciso, cualquier valor por debajo de 500 microsegundos.

```python
# time
import time
from time import time_ns

#conversion to nanoseconds
def timetime_ns():
        return time.time() * (10**9)
def perfcounter_ns():
        return time.perf_counter() * (10**9)
def processtime_ns():
        return time.process_time() * (10**9)

#different time function options
time_functions = [time.time_ns,perfcounter_ns,timetime_ns,processtime_ns]

#select the preferred time function, it must be the same for both time calls
#in this example perfcounter_ns
t1 = time_functions[1]()
#run the selected algorithm here
t2 = time_functions[1]()
```

Figura 1: Obtención de la hora del sistema

```python
import random

def generate_input(n):
    """
    Generates a sorted list of `n` unique, non-consecutive integers and a target sum.

    Args:
        n (int): The number of elements to generate in the list.

    Returns:
        tuple: A tuple containing:
            - A list of `n` unique, non-consecutive integers in sorted order.
            - An integer representing the target sum of two randomly selected numbers from the list.

    Example:
        >>> generate_input(5)
        ([2, 4, 6, 9, 11], 13)
    """
    nums = []
    current_value = random.randint(1, 10)
    for _ in range(n):
        nums.append(current_value)
        current_value += random.randint(1, 5)
    target = random.choice(nums) + random.choice(nums)
    return nums, target
```

Figura 2: Función para generar la entrada que reciben las dos versiones de la búsqueda de pares

```
+------+----------+------------+-------------+--------+----------+
|  n   | Averaged |    Time    | O(n*log(n)) | O(n^2) | O(n^2.2) |
+------+----------+------------+-------------+--------+----------+
|  50  |   True   | 70596.514  |   360.921   | 28.239 |  12.914  |
| 100  |   True   | 291928.05  |   633.914   | 29.193 |  11.622  |
| 200  |  False   |  1168562   |   1102.767  | 29.214 |  10.125  |
| 400  |  False   |  4901940   |   2045.385  | 30.637 |  9.243   |
| 800  |  False   |  20144487  |   3766.952  | 31.476 |  8.267   |
| 1600 |  False   |  79525831  |   6736.957  | 31.065 |  7.103   |
| 3200 |  False   | 319455025  |  12369.082  | 31.197 |   6.21   |
+------+----------+------------+-------------+--------+----------+
```

Figura 3: Posible salida por pantalla del programa que mide los tiempos de ejecución del algoritmo *SearchPairsWithLists*

Evaluación y modo de entrega

- La fecha de entrega límite es el día **27 de septiembre de 2024** a las 23:59 horas.
- Será necesario depositar en la página de la asignatura en el CampusVirtual, el fichero `Python` que contiene el código fuente de la práctica y el informe con el estudio de complejidad.
- Revise detenidamente el documento PlantillaPracticas para saber qué se va a evaluar.
- Solo uno de los integrantes del equipo de prácticas debe depositar el contenido en el CampusVirtual. El nombre de **todos** los integrantes debe figurar en el encabezado de los documentos a entregar.
- Todos los aspectos relacionados con dispensa académica, dedicación al estudio, permanencia y fraude académico se regirán de acuerdo con la normativa académica vigente de la UDC. Si las pruebas o actividades de evaluación se llevan a cabo en grupos, todos los miembros del grupo

serán responsables solidariamente por el trabajo realizado y entregado, así como de sus posibles consecuencias.