II EXAMEN PARCIAL (Segunda parte)

EDyA 361075 Carlos Eduardo Sánchez Torres

3 de mayo de 2021

1. Orden de complejidad

Los bucles son la estructura de control de flujo que suele determinar la complejidad computacional del algoritmo, ya que en ellos se realiza un mayor número de operaciones. Determine el orden de complejidad de los siguientes bucles:

1.1.

```
for (int i= 0; i < K; i++) {
   O (1)
}
```

Donde K es una constante multiplicativa, independiente de n (n es el número total de datos).

La suma de K costes del algoritmo para n muy grandes (el peor caso):

$$T(n) = \sum_{i=0}^{K} O(1) = KO(1) = O(1) \text{ (Orden de complejidad)} \tag{1}$$

1.2.

```
for (int i=0; i < n; i++) {
 for (int j=0; j < n; j++) {
    O (1) //representa alguna sentencia sencilla
 }
}
```

Suma de costes y orden de complejidad para n muy grandes (el peor caso):

$$T(n) = \sum_{i=0}^{n}\sum_{j=0}^{n} O(1) = \sum_{i=0}^{n} O(n)O(1) = O(n^2)O(1) = O(n^2) \tag{2}$$

1.3.

```
c=1;
while (c < n) {
    O (1) //representa alguna sentencia sencilla
    c=2*c;
}
```

Sabemos que $c = 1$ para la primera iteración, así

$$c = 2^{c-1} < n \tag{3}$$

$$c < log_2(n), c \in \mathbb{N} \tag{4}$$

$$T(n) = \underbrace{O(1) + O(1) + \cdots + O(1)}_{\lfloor log_2(n) \rfloor \text{ veces}} = \sum_{c=1}^{\lfloor log_2(n) \rfloor} O(1) = O(log_2(n))O(1) = O(log_2(n)) \tag{5}$$

1.4.

```
for (int i=0; i < n; i++) {
 c=i;
 while (c > 0) {
    O (1) //representa alguna sentencia sencilla
    c=c/2;
 }
}
```

Para el ciclo exterior, donde $f(n)$ es el ciclo interior:

$$T(n) = \sum_{i=0}^{n} f(i) \tag{6}$$

Si c es entero y definimos c con respecto a la iteración:

$$\underbrace{i, \lfloor \frac{i}{2} \rfloor, \lfloor \frac{i}{4} \rfloor, \lfloor \frac{i}{8} \rfloor, \lfloor \frac{i}{16} \rfloor, \cdots, \lfloor \frac{i}{2^j} \rfloor, \cdots, 0}_{\lfloor log_2(i) \rfloor + 1 \text{ veces}} \tag{7}$$

$$f(i) = \sum_{c=0}^{\lfloor log_2(i) \rfloor + 1} O(1) = O(log_2(i))O(1) = O(log_2(i)) \tag{8}$$

Como n es muy grande, solo interesa $i \to n$. Por tanto, $T(n) = O(n)O(log_2(n)) = O(nlog_2(n))$.

Si c es flotante, definimos c con respecto a la iteración: $i, \frac{i}{2}, \frac{i}{4}, \frac{i}{8}, \frac{i}{16}, \cdots, \frac{i}{2^j} > 0$. Vemos que el predicado siempre es verdadero. Y por tanto, f no termina, el algoritmo no acaba. Aunque, en la práctica, la memoria es finita, el algoritmo acaba con errores inesperados.

2. Algoritmos de búsqueda

Implemente los algoritmos de búsqueda, determine cual es el mejor y argumente utilizando los recursos de análisis de algoritmos.

Para que los algoritmos compitan en igualdad de condiciones, los datos de entrada tendrán las mismas características y serán las más restrictivas de ambos.

Entrada. Una secuencia ordenada de n números, $A = [a_1, a_2, \cdots, a_n]$

Salida. El índice j tal que $a[j] =$ número a encontrar.

```
def timeFunc(func):
    start = time.time()
    index = func([1, 2, 3, 4, 5, 6, 7, 8, 9], 1)
    end = time.time()
    print(f"Actual {index} Expected 0 {end - start}")


timeFunc(search)
```

Figura 1: Marco para probar experimentalmente.

2.1. Búsqueda secuencial

```python
def search(arr, number):
    index = 0
    for i in arr:
        if number == i:
            return index
        index += 1
    return -1
```

Figura 2: Búsqueda secuencial.

Las operaciones dentro del bucle son tiempos constantes, es decir, para n muy grandes, $O(1)$:

$$T(n) = \sum_{i=0}^{n} O(1) = O(n) \tag{9}$$

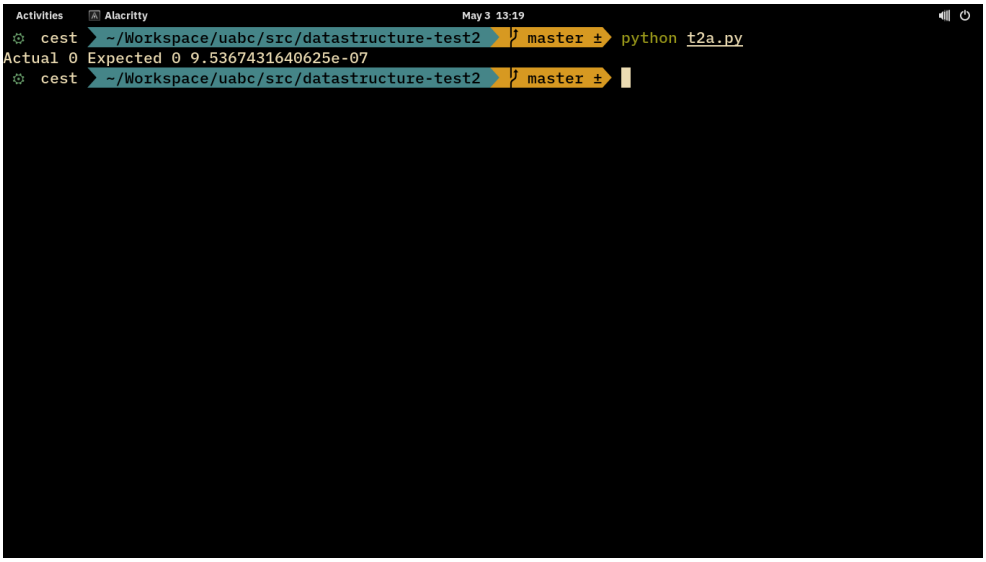

Figura 3: Resultados.

2.2. Búsqueda binaria

```python
def binarySearch(arr, number):
    lowerBound = 0
    upperBound = len(arr) - 1
    while lowerBound <= upperBound:
        index = int((upperBound + lowerBound) / 2)
        if arr[index] <= number:
            lowerBound = index + 1
        if arr[index] >= number:
            upperBound = index - 1
    if arr[index] == number:
        return index
    return -1
```

Figura 4: Búsqueda binaria.

Las operaciones dentro del bucle suceden en tiempos constantes, es decir, para n muy grandes, $O(1)$. El algoritmo tiene el mismo comportamiento (7), de los n elementos, solamente se necesitan $\lfloor log_2(n) + 1 \rfloor$ elementos.

$$T(n) = \sum_{i=0}^{\lfloor log_2(n)+1 \rfloor} O(1) = O(log_2(n)) \tag{10}$$

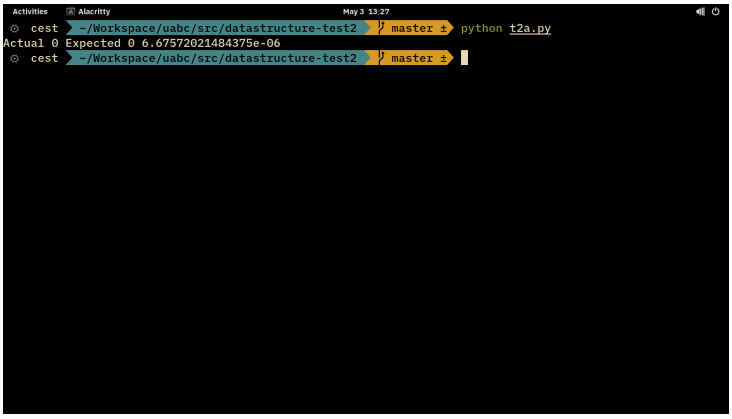

Figura 5: Resultados.

2.2.1. Invariante de lazo

Inicialización. Para $inferior = 0$ y $superior = n$, es cierto que $inferior \leq superior$ y que $i \in [inferior, superior]$.

Mantenimiento. Los límites inferior y superior se reducen a la mitad con respecto a la iteración anterior, $i \in [inferior, superior]$. Como los números tratados son conjuntos ordenados, se garantiza que a_i sea mayor, menor o igual al buscado, y por tanto, los límites inferior y superior convergen hacia la igualdad. Preservándose la invariante del bucle.

Terminación. El algoritmo finaliza cuando $inferior > superior$, el cual sucede porque convergen hacia su igualdad, y necesariamente el número es mayor, menor o igual al buscado, a saber, converge hacia el número y regresa su índice (que es igual al límite inferior y superior) o sale de la secuencia (fuera de los límites).

2.3. Conclusiones sobre la eficiencia

El algoritmo de búsqueda binaria es mas rápido que el secuencial, para n grandes, porque $O(n) > O(log_2(n))$. Sin embargo, en los valores extremos una búsqueda secuencial por el límite inferior o el superior tiene una eficiencia $O(1)$ mientras que el algoritmo de búsqueda $O(log_2(n)$.

Referencias

[1] Cormen, T. H., &; Leiserson, C. E. (2009). Introduction to Algorithms, 3rd edition. In Introduction to algorithms, 3rd edition.