Instituto Politécnico Nacional

Escuela Superior de Cómputo

Análisis de algoritmos

Profesora. **Luz María Sánchez García**

PRÁCTICA 3

ALGORITMO DIVIDE Y VENCE

Grupo: 3CM13

Equipo: CODEART

Integrantes:

1. Shim Kyuseop
2. Ortiz Jiménez José Antonio
3. Quintero Maldonado Iván

Fecha de entrega: 2 de abril de 2022

Planteamiento del problema

Analizar el comportamiento de búsqueda binaria vs búsqueda binaria aleatoria con el enfoque “divide y vencerás”.

Búsqueda binaria

La búsqueda binaria es un algoritmo de búsqueda que se utiliza en una matriz ordenada **dividiendo repetidamente el intervalo de búsqueda por la mitad**. La idea de la búsqueda binaria es usar la información de que la matriz está ordenada y reducir la complejidad del tiempo a O (log n).

Los pasos básicos para realizar la Búsqueda Binaria son:

- Comience con un intervalo que cubra todo el arreglo.
- Si el valor de la clave de búsqueda es menor que el elemento en el medio del intervalo, reduzca el intervalo a la mitad inferior.
- De lo contrario, redúcelo a la mitad superior.
- Verifique repetidamente hasta que se encuentre el valor o el intervalo esté vacío.

Búsqueda binaria aleatoria

Nos dan una matriz ordenada A[] de n elementos. Necesitamos encontrar si x está presente en A o no. En la búsqueda binaria siempre usamos el elemento medio, aquí elegiremos al azar un elemento en el rango dado.

Generador de números aleatorios

Es importante saber de donde vienen estos números "aleatorios", ya que podemos esperar una secuencia después de ciertas repeticiones (millones) y podría tener una variación en tiempo (dados ciertos parámetros).

Un generador de números aleatorios (RNG por sus siglas en inglés) es un dispositivo informático o físico diseñado para producir secuencias de números sin orden aparente.

Los algoritmos para la generación de valores uniformemente distribuidos están presentes en todas las calculadoras y lenguajes de programación, y suelen estar basados en congruencias numéricas del tipo:

$$x_{n+1} \equiv (ax_n + c) \pmod{m}$$

El éxito de este tipo de generadores de valores de una variable aleatoria depende de la elección de los cuatro parámetros que intervienen inicialmente en la expresión anterior:

- El valor inicial o *semilla*: x0
- La constante multiplicativa: α
- La constante aditiva: c
- El número *m* respecto al cual se calculan los restos

Actividades

1. El programa de los algoritmos ordenamientos: búsqueda binaria y búsqueda binaria aleatoria

   Consulte las secciones Prueba y Anexo

2. El programa debe utilizar funciones recursivas

   Consulte las secciones Prueba y Anexo

3. **Análisis temporal**

   Búsqueda binaria

```python
def binary_search(list: List[int], left:int, right:int, target:int):
    if right >= left:
        middle = left + (right - left) // 2
        if list[middle] == target:
            print(f'El elemento se encuentra en el índice: {middle}')
            return
        elif list[middle] > target:
            return binary_search(list, left, middle - 1, target)
        else:
            return binary_search(list, middle + 1, right, target)
    else:
        print("El elemento no está en la lista")
        return
```

   Operación básica: comparaciones de list[middle] con target

   Análisis temporal: La eucación de recurrencia es

$$T(n) = T\left(\frac{n}{2}\right) + 3$$

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$a = 1,\ \ b = 2,\ \ f(n) = 1$$

$$n^{\log_2 1} = n^0 = 1$$

$$f(n) = O(n^{\log_2 1})$$

$$Por\ lo\ tanto, es\ caso\ 2\ en\ teorema\ maestro$$

$$T(n) = O(1 * \log\ n) = O(log\ n)$$

Búsqueda binaria aleatoria

```python
def random_binary_search(list: List[int], left:int, right:int, target:int):
    if right >= left:
        middle = random.randint(left, right)
        if list[middle] == target:
            print(f'El elemento se encuentra en el índice: {middle}')
            return
        if list[middle] > target:
            return random_binary_search(list, left, middle - 1, target)
        else:
            return random_binary_search(list, middle + 1, right, target)
    print("El elemento no está en la lista")
    return
```

Operación básica: comparaciones de list[middle] con target

Análisis temporal:

Tal que generamos el pivote aleatoriamente, puede ser cualquier número en rango de 1 a n. Supongamos que cada pivote tiene la misma posibilidad, cada uno tiene 1/n de posibilidad. Entonces, la ecuación de recurrencia sería:

$$T(n) = \frac{[T(1) + T(2) + \cdots + T(n)]}{n} + O(1)$$

$$n \cdot T(n) = [T(1) + T(2) + \cdots + T(n)] + n\ \ \cdots eq1$$

$$Sustituyendo\ n = n - 1$$

$$(n-1)T(n) = [T(1) + T(2) + \cdots + T(n-1)] + (n-1)\ \cdots eq2$$

$$Restando\ eq1\ y\ eq2$$

$$n \cdot T(n) - (n-1)T(n) = T(n) - 1$$

$$(n-1)T(n) = (n-1)T(n-1) + 1$$

$$T(n) = \frac{(n-1)T(n-1)+1}{n-1}$$

$$T(n) = \frac{1}{n-1} + T(n-1)$$

$$Si\ calculamos\ T(n-1)\ con\ misma\ mecanica,$$

$$T(n-1) = \frac{1}{n-2} + T(n-2)$$

$$Entonces,$$

$$T(n) = \frac{1}{n-1} + \frac{1}{n-2} + T(n-2)$$

$$T(n) = \frac{1}{n-1} + \frac{1}{n-2} + \frac{1}{n-3} + T(n-3)$$

$$\vdots$$

$$T(n) = \frac{1}{n-1} + \frac{1}{n-2} + \frac{1}{n-3} + \cdots + \frac{1}{2} + 1 \ \in \ O(\log n)$$

4. Calcular la cantidad de instrucciones que tarda cada algoritmo

Las tablas son iguales porque su complejidad temporal es O(log n) en ambos casos. Sin embargo, existiría la diferencia de número de instrucciones que ejecuta porque en búsqueda binaria aleatoria se define valor de pivote aleatoriamente. Entonces el número exacto de instrucciones podría variar dependiendo valor de pivote generado.

Búsqueda binaria

K	INSTRUCCIONES
100	6.64385619
200	7.64385619
300	8.22881869
400	8.64385619
500	8.965784285

600	9.22881869
700	9.451211112
800	9.64385619
900	9.813781191
1000	9.965784285
10000	13.28771238
100000	16.60964047
1000000	19.93156857

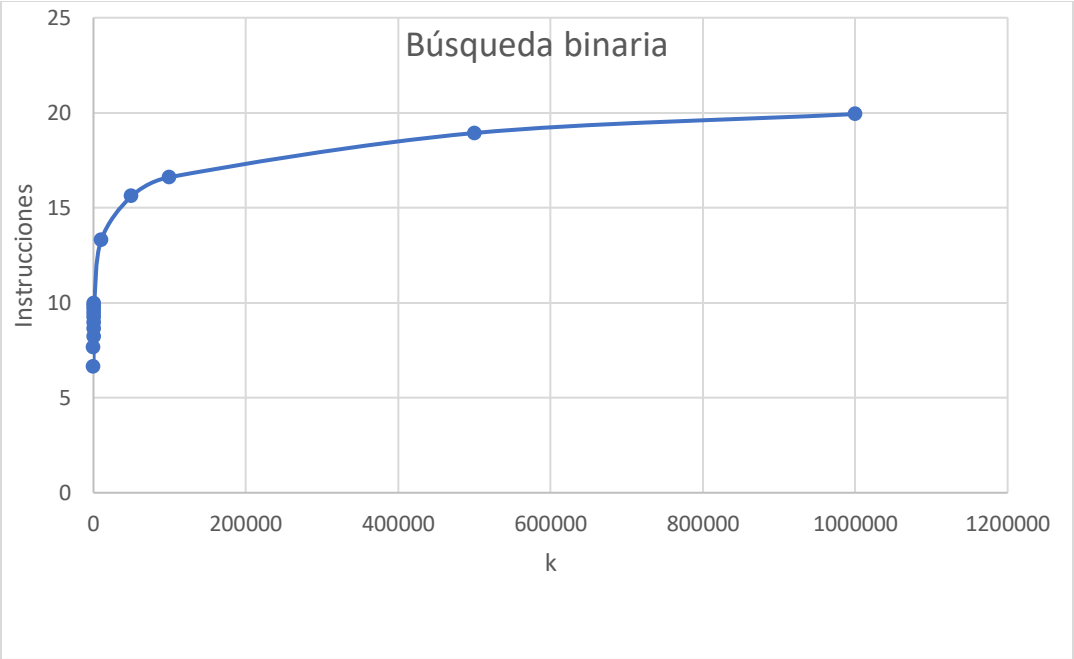
Búsqueda binaria aleatoria

K	INSTRUCCIONES
100	6.64385619
200	7.64385619
300	8.22881869
400	8.64385619
500	8.965784285
600	9.22881869
700	9.451211112
800	9.64385619
900	9.813781191
1000	9.965784285
10000	13.28771238
100000	16.60964047
1000000	19.93156857

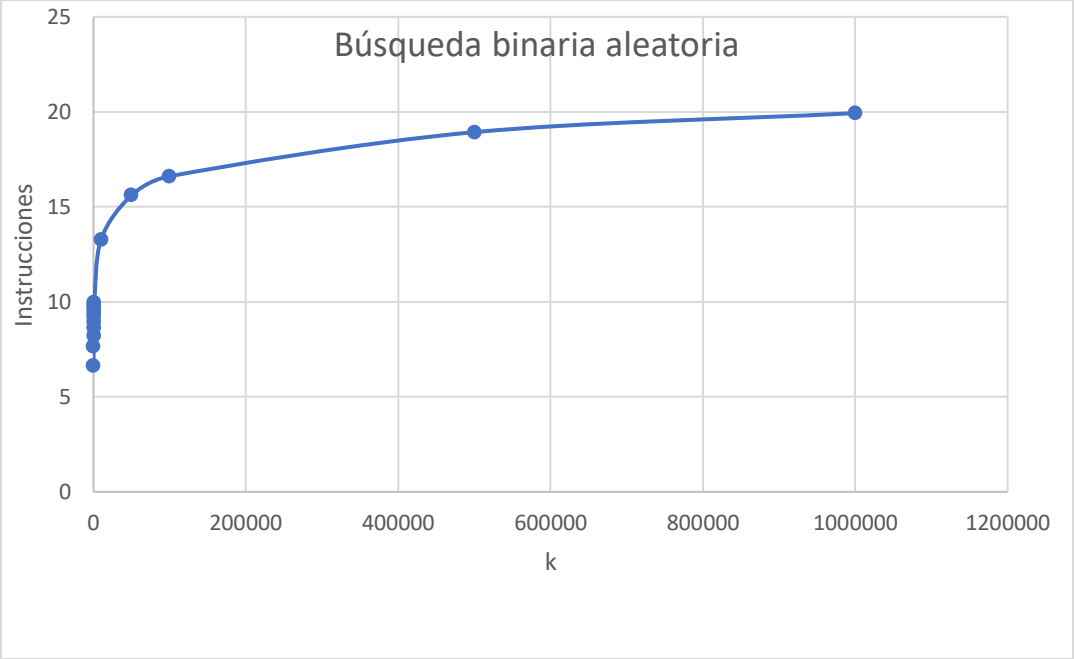
5. Gráficas de comportamiento temporal de cada algoritmo

Las gráficas son iguales porque su complejidad temporal son O(log n) en ambos casos. Sin embargo, existiría la diferencia de tiempo de ejecución porque en búsqueda binaria aleatoria, se define valor de pivote aleatoriamente. Entonces su tiempo de ejecución exacta va ser poco diferente a búsqueda binaria normal aunque ambos si pertenecen a O(log n)

Búsqueda binaria



Búsqueda binaria aleatoria



6. Análisis espacial

   En ambos algoritmos tendrá O(n) de análisis espacial, por list(array) de 1x1.

7. Calcular cantidad de celdas de memorias necesarias para cada algoritmo

   Ambos tienen complejidad espacial de O(n), entonces las tablas son iguales.

Búsqueda binaria

K	CELDAS
100	100
200	200
300	300
400	400
500	500
600	600
700	700
800	800
900	900
1000	1000
10000	10000
50000	50000
100000	100000
500000	500000
1000000	1000000

Búsqueda binaria aleatoria

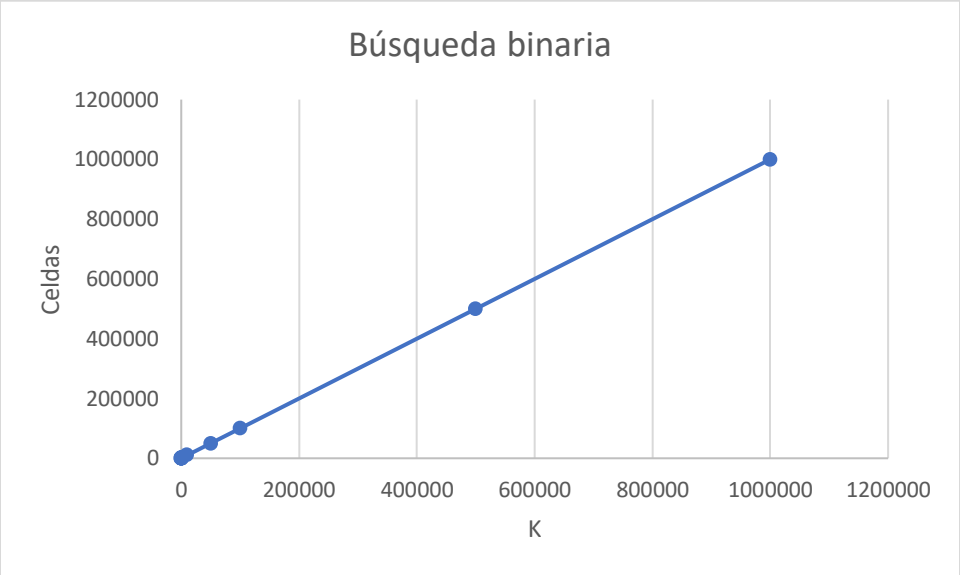
K	CELDAS
100	100
200	200
300	300
400	400
500	500
600	600
700	700
800	800

900	900
1000	1000
10000	10000
50000	50000
100000	100000
500000	500000
1000000	1000000

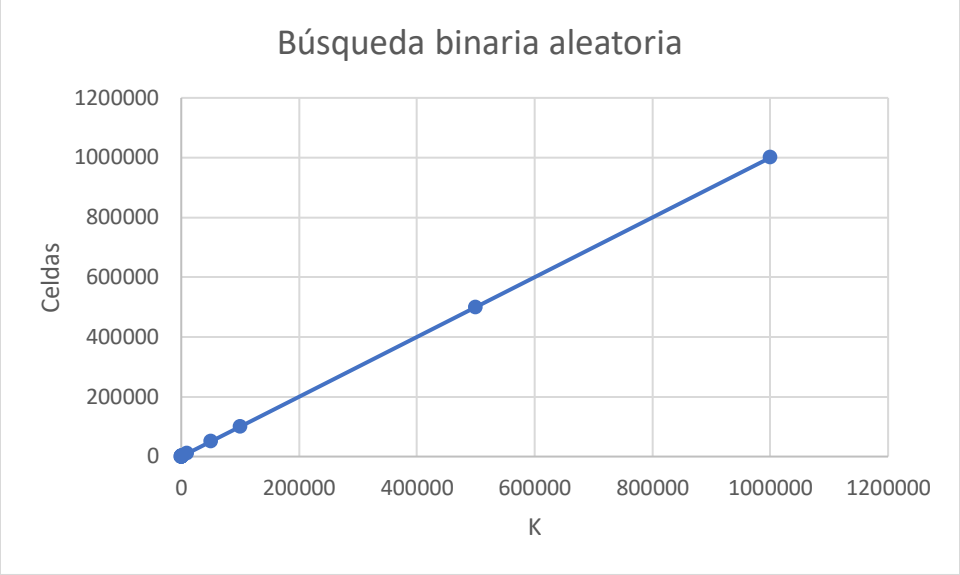
8. Gráficas de comportamiento espacial de cada algoritmo

   Ambos tienen complejidad espacial de O(n), entonces las gráficas son iguales.

Búsqueda binaria



Búsqueda binaria aleatoria



9. **Indique el entorno controlado para realizar las pruebas experimentales**

- Sistema operativo basado en Linux
- Python 3

10. Preguntas

a) Para cada algoritmo ¿Existe alguna solución que no sea recursiva?

Si, existe una forma iterativa para cada uno.

b) ¿Cuál de los 2 algoritmos es el más fácil de implementar?

Búsqueda binaria.

c) ¿Cuál de los 2 algoritmos es el más difícil de implementar?

Búsqueda binaria aleatoria, por la parte del índice aleatorio. Son casi idénticos.

d) ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?

Si, ambos tiene tiempo de ejecución similar, porque sus complejidades temporales pertenecen a O(log n). Dependiendo de valor de pivote que genera, búsqueda binaria aleatoria puede ser lento o también rápido que búsqueda binaria normal.

e) ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?

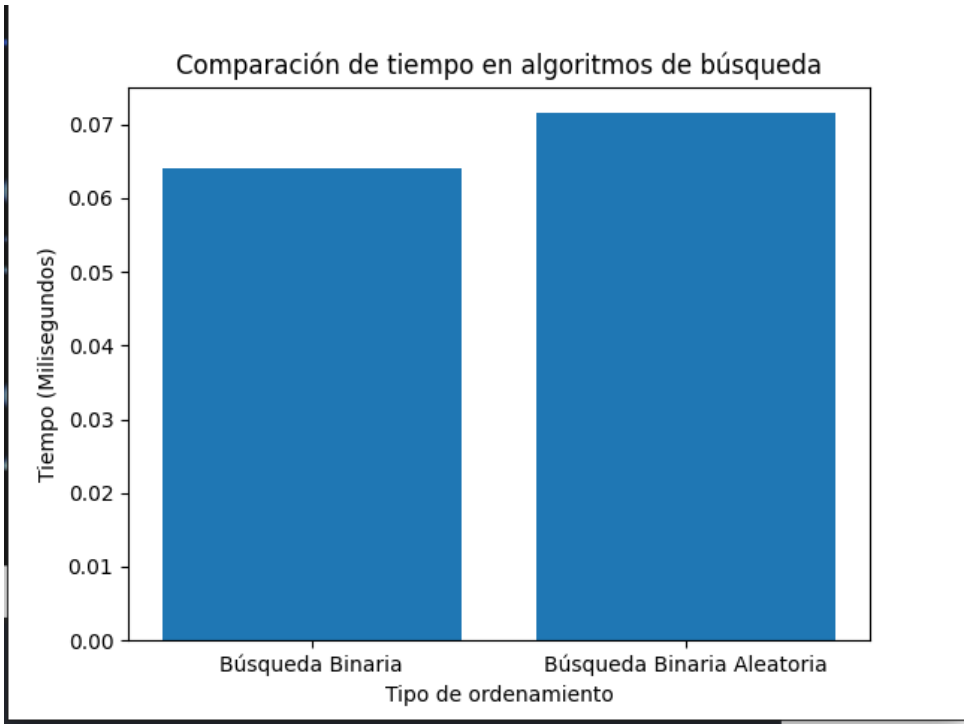
Usar "*typing"* para tener un fuerte tipado, usar la misma versión de python y utilizar VS Code para un mejor desarrollo. Además de investigar cómo funciona "random" para entender mejor el algoritmo.

Anexo

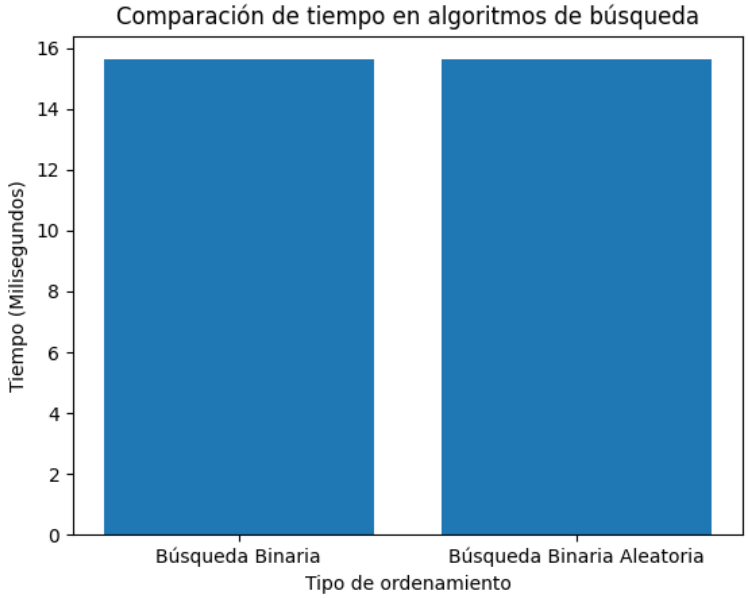
```python
from typing import List
import matplotlib.pyplot as plt
from time import time
import sys
import random
# Variables for the algorithms
start = 0
end = 100000000
step = 1
target = 1
# Change Recursion maximum
sys.setrecursionlimit(10000)
# Insertion Sort Algorithm
def binary_search(list: List[int], left:int, right:int, target:int):
    if right >= left:
        middle = left + (right - left) // 2
        if list[middle] == target:
            print(f'El elemento se encuentra en el índice: {middle}')
            return
        elif list[middle] > target:
            return binary_search(list, left, middle - 1, target)
        else:
            return binary_search(list, middle + 1, right, target)
    else:
        print("El elemento no está en la lista")
        return
# Selection Random Binary Search
def random_binary_search(list: List[int], left:int, right:int, target:int):
    if right >= left:
        middle = random.randint(left, right)
        if list[middle] == target:
            print(f'El elemento se encuentra en el índice: {middle}')
            return
        if list[middle] > target:
            return random_binary_search(list, left, middle - 1, target)
        else:
            return random_binary_search(list, middle + 1, right, target)
    print("El elemento no está en la lista")
    return
# Time Flag
cpu_start_time = 0
# Creation of list
list = [i for i in range(start,end+1,step)]
list_last_index = len(list)-1
# Binary Search
print("Búsqueda binaria")
cpu_start_time = time()
binary_search(list, 0, list_last_index, target)
cpu_binary_search = (time() - cpu_start_time) * 1000
print(f'Milisegundos: {cpu_binary_search}')
# Binary Search
print("Búsqueda binaria aleatoria")
cpu_start_time = time()
random_binary_search(list, 0, list_last_index, target)
cpu_random_binary_search = (time() - cpu_start_time) * 1000
print(f'Milisegundos: {cpu_random_binary_search}')
# Chart
plt.bar(["Búsqueda Binaria", "Búsqueda Binaria Aleatoria"],
        [cpu_binary_search, cpu_random_binary_search])
plt.title("Comparación de tiempo en algoritmos de búsqueda")
plt.xlabel("Tipo de ordenamiento")
plt.ylabel("Tiempo (Milisegundos)")
plt.show()
```

Pruebas

```
antonio@antonio:~/Desktop/Antonio/Algoritm
Búsqueda binaria
El elemento se encuentra en el índice: 1
Milisegundos: 0.06413459777832031
Búsqueda binaria aleatoria
El elemento se encuentra en el índice: 1
Milisegundos: 0.07152557373046875
```




```
~#@> python .\main.py
Búsqueda binaria
El elemento se encuentra en el índice: 1
Milisegundos: 15.623807907104492
Búsqueda binaria aleatoria
El elemento se encuentra en el índice: 1
Milisegundos: 15.625476837158203
```



Bibliografía

- GeeksforGeeks. (2022b, marzo 9). *Binary Search*. https://www.geeksforgeeks.org/binary-search/?ref=gcse

- *Algoritmo de búsqueda binaria aleatoria*. (2022, 25 enero). GeeksforGeeks. Recuperado 2 de abril de 2022, de https://www.geeksforgeeks.org/randomized-binary-search-algorithm/

- colaboradores de Wikipedia. (2021, 28 julio). *Generador de números aleatorios*. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Generador_de_n%C3%BAmeros_aleatorios