	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jesús Cruz Navarro
Asignatura:	Estructuras de Datos y Algoritmos 2
Grupo:	1
No de Práctica(s):	4
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	32
Semestre:	2021-1
Fecha de entrega:	28 de octubre de 2020
Observaciones:	

CALIFICACIÓN: _____

PRÁCTICA 4: BÚSQUEDA 1

Objetivo: El estudiante identificará el comportamiento y características de algunos algoritmos de búsqueda por comparación de llaves.

1. Pruebas de los algoritmos vistos en clase.

a. Cada algoritmo debe estar en una función. Estos son:

i. Búsqueda Lineal *(Mejorado)*

ii. Búsqueda Binaria *(Recursivo)*

b. Desarrollar un programa para probar los algoritmos implementados anteriormente y registrar sus tiempos de ejecución con listas de tamaño n con elementos en el rango **[0, n/2].** Ambos algoritmos deben recibir el mismo conjunto de elementos (es decir, si es una lista aleatoria, los algoritmos deben recibir el mismo conjunto aleatorio). El programa debe imprimir al final **n**, **key**, y el tiempo de ejecución de cada algoritmo. Busque la llave **key** de su preferencia.

c. Se debe probar con listas de diferentes tamaños **(n=0.2M, 0.5M, 1M)** y registrar los tiempos de ejecución para el caso promedio. **NO CONSIDERAR EL ORDENAMIENTO COMO PARTE DEL TIEMPO DE EJECUCIÓN DE LA BÚSQUEDA BINARIA.**

Dado que los tiempos pueden variar de una ejecución a otra, realice las pruebas **3** veces y obtenga el promedio de los tiempos y genere una gráfica para comparar los tiempos de ambos algoritmos en función del tamaño de la lista (**n** vs **t**, es decir, las **n** en el eje **X** y los tiempos en segundos en el eje **Y**).

En lugar de usar **time.time()** y use **time.perf_counter()** que tiene mayor precisión. Su uso es el mismo.

El programa realizado consta de distintas funciones auxiliares, las cuales se mostrarán a continuación.

Primero, se definieron algunas constantes importantes: los algoritmos a utilizar, sus nombres, y los valores de **n** a analizar:

```
ALG1 = linearSearch.linearSearch
ALG2 = binarySearch.binarySearch
ALG1_NAME = "Linear search"
ALG2_NAME = "Binary search"

# Valores de n a analizar
X_ARR = [200000, 500000, 1000000]
```

Constantes

searchAndComputeTime

Esta función ordena busca el elemento proporcionado **(toSearch)** en la lista **arr**, utilizando el algoritmo de búsqueda proporcionado **(searchAlg)**, y calcula el tiempo utilizado por éste.

```
# Función que busca el elemento deseado en la lista
# utilizando el algoritmo proporcionado
def searchAndComputeTime(searchAlg, arr, toSearch):

    # Buscar el elemento y calcular el tiempo de ejecución
    t1 = time.perf_counter()
    searchAlg(arr, toSearch)
    t2 = time.perf_counter()

    # Devolver el tiempo utilizado
    return t2 - t1
```

Función searchAndComputeTime

searchAndGetTimes

Esta función busca el elemento proporcionado en la lista **arr**, con los dos algoritmos de búsqueda analizados y retorna una tupla con los tiempos utilizados por cada uno. Nótese que, para el segundo algoritmo (**binarySearch**), se genera una copia ordenada de la lista.

```python
def searchAndGetTimes(arr, toSearch):
    print("N =", len(arr))
    print("key =", toSearch)

    firstAlgTime = searchAndComputeTime(ALG1, arr, toSearch)
    print(ALG1_NAME, ":", firstAlgTime)

    sortedCopy = arr.copy()
    sortedCopy.sort()

    secondAlgTime = searchAndComputeTime(ALG2, sortedCopy, toSearch)
    print(ALG2_NAME, ":", secondAlgTime, "\n")

    return firstAlgTime, secondAlgTime
```

Función searchAndGetTimes

computeAvgTimesArr

Esta función retorna una lista de tuplas con los tiempos promedio de búsqueda para cada **n**.

```python
def computeAvgTimesArr():
    # Lista con las tuplas de tiempos promedios para cada n.
    AVG_TIMES = []
    # Calcular los tiempos promedio para cada n y agregar la tupla respectiva a la list
    for n in X_ARR:
        # Lista que contendrá 3 tuplas con los tiempos obtenidos para n
        TIMES_FOR_N = []
        # Calcular 3 veces el tiempo de los algoritmos
        for i in range(3):
            arr = [random.randint(0, n // 2) for j in range(n)]
            TIMES_FOR_N.append(searchAndGetTimes(arr, random.randint(0, n // 2)))

        # Calcular el promedio de tiempo para cada algoritmo
        firstMean = numpy.mean([t[0] for t in TIMES_FOR_N])
        print("Tiempo promedio", ALG1_NAME, "para", n, "elementos:", firstMean)
        secondMean = numpy.mean([t[1] for t in TIMES_FOR_N])
        print("Tiempo promedio", ALG2_NAME, "para", n, "elementos:", secondMean, "\n")
        # Agregar la tupla de tiempo promedio obtenida
        AVG_TIMES.append((firstMean, secondMean))
    return AVG_TIMES
```

Función computeAvgTimesArr

testAndPlot

Esta función obtiene los tiempos promedios para cada **n**, e imprime la gráfica con la información necesaria utilizando la librería *matplotlib*. El parámetro **title** debe contener el título a mostrar en la gráfica.

```python
# Función que se encarga de realizar el test para
# el caso correspondiente.
# Obtiene una lista de tuplas con los tiempos promedios para cada n,
# y finalmente la grafica.
def testAndPlot(title):
    print(title, "\n")
    AVG_TIMES = computeAvgTimesArr()

    fig, ax = plt.subplots()
    ax.set_title(title)
    ax.set_xlabel("Tamaño del arreglo")
    ax.set_ylabel("Tiempo de ejecución")
    ax.plot(X_ARR, [y[0] for y in AVG_TIMES])
    ax.plot(X_ARR, [y[1] for y in AVG_TIMES])
    ax.legend([ALG1_NAME, ALG2_NAME])
```

Función testAndPlot

Finalmente, se ejecuta la función anterior y se muestra la gráfica generada:

```python
testAndPlot("Algoritmos de búsqueda")
plt.show()
```

Ejecución de los tests y graficación

A continuación, se muestran algunas capturas de pantalla de la salida producida por la ejecución del programa:

```
N = 200000
key = 5840
Linear search : 0.009561399999999942
Binary search : 1.81000000001319e-05

N = 200000
key = 57864
Linear search : 0.004902799999999985
Binary search : 2.1799999999849717e-05

N = 200000
key = 45890
Linear search : 0.0056951999999999
Binary search : 2.019999999980371e-05

Tiempo promedio Linear search para 200000 elementos: 0.0067197999999999425
Tiempo promedio Binary search para 200000 elementos: 2.0033333333261776e-05
```

Salida para N = 200000

```
N = 500000
key = 31214
Linear search : 0.01669570000000009
Binary search : 1.870000000003813e-05

N = 500000
key = 52215
Linear search : 0.01568360000000002
Binary search : 1.9599999999897477e-05

N = 500000
key = 183849
Linear search : 0.014969899999999647
Binary search : 2.0000000000131024e-05

Tiempo promedio Linear search para 500000 elementos: 0.015783066666666585
Tiempo promedio Binary search para 500000 elementos: 1.9433333333355545e-05
```

Salida para N = 500000

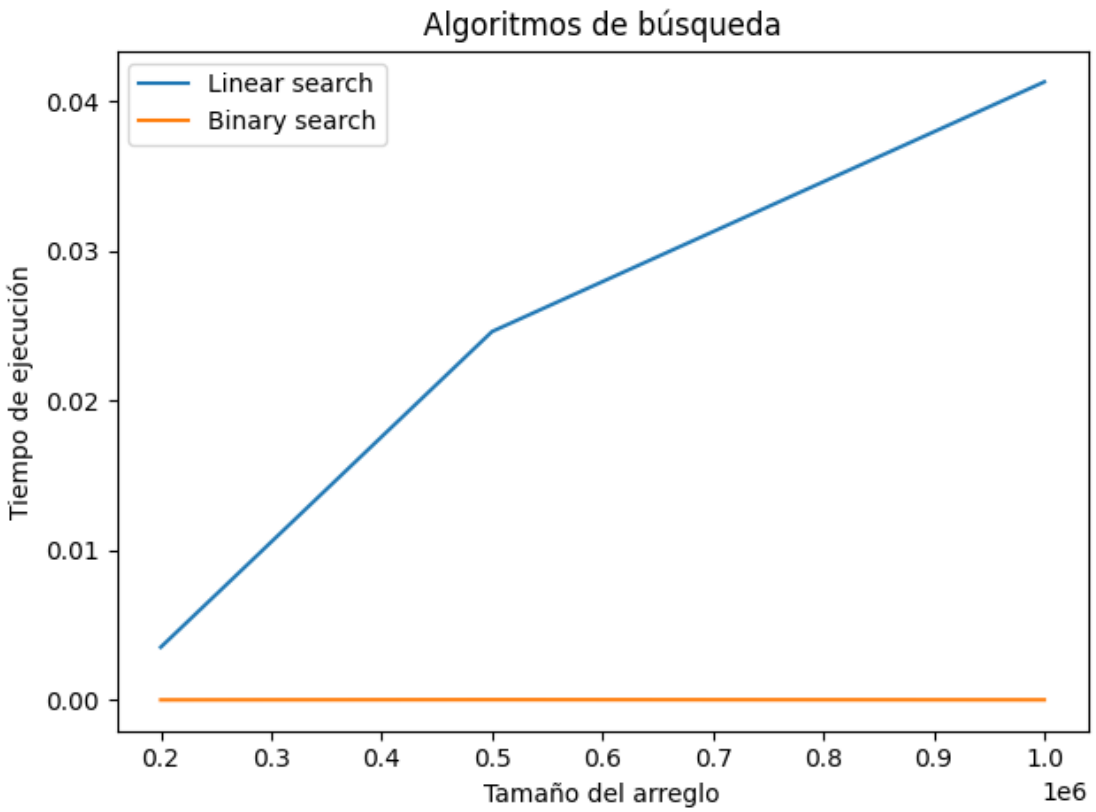
```
N = 1000000
key = 2599
Linear search : 0.027306799999999853
Binary search : 1.8400000000085015e-05

N = 1000000
key = 58414
Linear search : 0.00845729999999989
Binary search : 2.0499999999756824e-05

N = 1000000
key = 3756
Linear search : 0.006443499999999602
Binary search : 1.7200000000272553e-05

Tiempo promedio Linear search para 1000000 elementos: 0.014069199999999782
Tiempo promedio Binary search para 1000000 elementos: 1.870000000003813e-05
```

Salida para N = 1000000



Gráfica generada

Como se logra apreciar, la curva correspondiente al rendimiento de **búsqueda lineal** tiene una forma similar a la de una función lineal, lo cual coincide con la complejidad analizada en clase. Por otro lado, la forma de la curva correspondiente a **búsqueda binaria** se asemeja a la de una función constante. Sin embargo, lo anterior probablemente se deba a la baja cantidad de mediciones realizadas, ya que sabemos que su complejidad es logarítmica. Evidentemente, los tiempos de ejecución de **búsqueda lineal** fueron mayores a los de **búsqueda binaria**.

2. Búsqueda de los m vecinos más cercanos en un radio k.

> **a.** Diseñar y desarrollar un algoritmo que permita, utilizando búsqueda binaria y búsqueda lineal, encontrar los m elementos más cercanos a una distancia máxima r de un valor x, en una colección de enteros aleatorios. X puede no existir dentro de la colección. Si se encuentra menos elementos que m, regresar solo estos (ver ejemplo 2).
>
> **b.** El algoritmo debe estar en una función **BusquedaVecinos(arreglo, x, r, m)** y regresar una lista con los elementos encontrados. Si no hay elementos, regresa una lista vacía.
>
> **TIP:** Modifique la búsqueda binaria para, en caso de no existir el elemento, regrese siempre el índice en lugar de **-1** (sería el lugar donde debería estar el número buscado). Con este índice, busque hacia adelante **(i-1, i-2, ...)** y hacia atrás **(i+1, i+2, ...)** a los elementos que estén dentro del rango. Repita esto hasta que la lista de salida esté llena (su **len** sea igual a **m**), o hasta que los elementos **izq** y derecha estén fuera del rango. Tenga cuidado con índices fuera de la lista.

Primero, se implementó el algoritmo de **búsqueda binaria** modificado propuesto: en lugar de retornar **-1** en caso de no encontrarse el elemento, se retornó el índice donde se encontraría el elemento en caso de existir, es decir, el valor de **start**:

```python
# Wrapper de búsqueda binaria modificada
# Esta versión retorna el índice donde el elemento
# debería estar, en caso de no encontrarse
def modifiedBinarySearch(arr, toSearch):
    return binarySearchAux(arr, 0, len(arr) - 1, toSearch)


def binarySearchAux(arr, start, end, toSearch):
    if start > end:
        # Retornar índice correspondiente
        return start

    halfIndex = (start + end) // 2
    if arr[halfIndex] == toSearch:
        return halfIndex
    elif arr[halfIndex] > toSearch:
        return binarySearchAux(arr, start, halfIndex - 1, toSearch)
    else:
        return binarySearchAux(arr, halfIndex + 1, end, toSearch)
```

Búsqueda binaria modificado

Posteriormente, se implementó el algoritmo para encontrar a los vecinos más cercanos cumpliendo con las restricciones planteadas. Su funcionamiento es el siguiente:

Primero, se crea una copia ordenada del arreglo proporcionado, para poder efectuar la búsqueda correspondiente; se ejecuta el algoritmo de **búsqueda binaria modificado** sobre ésta y se almacena el índice obtenido. Además, se creó una función auxiliar para calcular la diferencia entre el elemento a buscar y el elemento en el índice proporcionado:

```python
def findClosestNeighbors(arr, num_neighbors, max_range, toSearch):
    # Asegurarnos de que el arreglo está ordenado
    # para ejecutar la búsqueda
    copy = arr.copy()
    copy.sort()
    index = modifiedBinarySearch(copy, toSearch)
    neighbors = []

    def getDifference(idx):
        return abs(copy[idx] - toSearch)
```

Obtención del índice y definición de función getDifference

Después, se crean dos punteros: **leftPointer**, que contiene el índice del elemento justo a la izquierda del elemento buscado, y **rightPointer**, que contiene el índice del elemento buscado, en caso de haber sido encontrado. En el caso contrario, contiene la posición donde se encontraría el elemento en caso de existir.

Además, se obtiene la diferencia de los elementos presentes en ambos punteros. El puntero de la **derecha** siempre contiene un elemento, por lo tanto, es posible obtener su diferencia sin problema alguno. Sin embargo, es posible que el puntero **izquierdo** sobrepase los límites del arreglo, por lo tanto, se inicializa con un valor de **infinito** y se reasigna en caso de ser válido el índice.

```
# Índices de los posibles elementos
# más cercanos a agregar
leftPointer = index - 1
rightPointer = index

# Diferencia entre los elementos anteriores
# y el elemento buscado
differenceRight = getDifference(rightPointer)
differenceLeft = math.inf
if leftPointer ⩾ 0:
    differenceLeft = getDifference(leftPointer)
```

Asignación de índices y diferencias

Posteriormente, se agregan los vecinos correspondientes. El ciclo mostrado se ejecuta mientras se cumplan dos condiciones: aún no se ha obtenido el número de vecinos solicitados **y** aún hay elementos potenciales a ser agregados. En cada iteración, se agrega el elemento con menor diferencia al arreglo **neighbors**, y se actualiza el puntero correspondiente. También se reasigna la diferencia apropiada: en caso de existir un elemento en el puntero, se obtiene utilizando la función **getDifference**, de lo contrario, se le asigna el valor de **infinito**, para que, de esta forma, la diferencia siempre sea mayor al rango posible y no se cumpla la condición en las iteraciones posteriores. Es decir, en este punto, ya no es posible agregar algún vecino en el lado correspondiente del puntero.

```python
while len(neighbors) < num_neighbors and (differenceLeft ⩽ max_range or differenceRight ⩽ max_range):
    # En este punto ambos vecinos candidatos están dentro del rango

    # La distancia al vecino izquierdo es menor o igual
    if differenceLeft ⩽ differenceRight:
        neighbors.append(copy[leftPointer])
        leftPointer -= 1
        # Si se ha llegado al principio del arreglo,
        # invalidar la diferencia izquierda
        if leftPointer == -1:
            differenceLeft = math.inf
        else:
            differenceLeft = getDifference(leftPointer)
    # La distancia al vecino derecho es mayor
    else:
        neighbors.append(copy[rightPointer])
        rightPointer += 1
        # Si se ha llegado al final del arreglo,
        # invalidar la diferencia derecha
        if rightPointer == len(copy):
            differenceRight = math.inf
        else:
            differenceRight = getDifference(rightPointer)

neighbors.sort()
return neighbors
```

Adición de los vecinos correspondientes al arreglo neighbors

Por último, se ordena el arreglo **neighbors** y se retorna.

```python
neighbors.sort()
return neighbors
```

Se retornan los vecinos obtenidos

Finalmente, se ejecuta el algoritmo con las entradas mostradas en la guía de la práctica, obteniendo la salida esperada:

```python
print(findClosestNeighbors([10, 30, 60, 72, 80, 82, 84, 85, 86, 88, 90, 91, 92, 94, 97, 100], 10, 10, 84))
print(findClosestNeighbors([3, 5, 22, 40, 42, 43, 45, 47, 55], 5, 2, 41.5))
```

Ejecución de los casos de la guía de la práctica

```
[80, 82, 84, 85, 86, 88, 90, 91, 92, 94]
[40, 42, 43]
```

Salida del programa

CONCLUSIONES

Los algoritmos de búsqueda se encargan de resolver un problema ubicuo en las Ciencias de la Computación. Como en la mayoría de los casos donde se analizan distintos algoritmos para resolver un mismo problema, existen diferencias en sus rendimientos respectivos: en este caso, la complejidad asintótica de búsqueda lineal supera a la de búsqueda binaria, por ende, el rendimiento de este último algoritmo es mejor. Lo anterior se vio claramente reflejado en los tiempos de ejecución obtenidos en la práctica.

Sin embargo, también es importante mencionar que, en el caso de búsqueda binaria, los datos de entrada deben contar con cierta característica: deben encontrarse ordenados. Por otro lado, en búsqueda lineal esta restricción no aplica. Estas situaciones influyen en la decisión sobre la utilización de determinado algoritmo sobre otro; en cada caso de uso específico deben analizarse las restricciones y características del problema para determinar si es factible su utilización o no.