ANÁLISIS: ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO

Brizuela González Juan Eduardo

Benítez Ruiz Antonio

Universidad Iberoamericana Puebla

Estructura de Datos y Algoritmos Avanzados

21 de febrero de 2022

Índice

1. Algoritmos de búsqueda

Un algoritmo de búsqueda es un conjunto de instrucciones que están diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos.

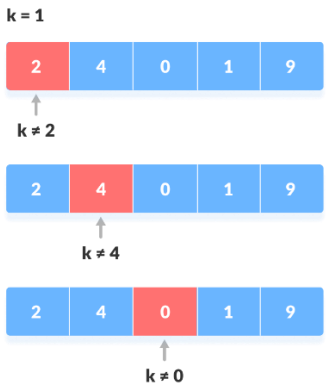
Así, pueden localizarlo o de otra manera, determinar su inexistencia. La búsqueda será mucho más eficiente si la estructura de datos está ordenada. De esta manera, los algoritmos que se presentarán en las siguientes secciones, servirán para realizar búsquedas en vectores ordenados.

1.1. Búsqueda Secuencial

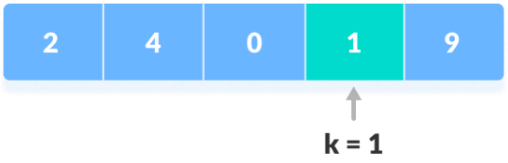
La búsqueda lineal, o secuencial, es un algoritmo donde comenzamos desde un extremo y verificamos cada elemento de la lista hasta encontrar el elemento deseado. Es el algoritmo de búsqueda más simple.

Se trata del único algoritmo posible si el vector no está ordenado. Sin embargo, si el vector está ordenado, éste resulta muy ineficiente.

1. Comienza desde el primer elemento, compara [k] con cada elemento x.



2. Si k == x, devuelve el índice.



3. De lo contrario, regresa “no encontrado”

Pseudocódigo

```
Busquedalineal (matriz, clave)

     para cada elemento de la matriz

          si elemento == valor

               devolver su índice
```

Complejidad de Tiempo	O(n)
Complejidad Espacial	O(1)

La búsqueda secuencial se usa si el arreglo no está ordenado, para buscar en arreglos muy pequeños.

1.2. Búsqueda Secuencial por Centinela

El algoritmo de búsqueda secuencial resulta muy ineficiente puesto que no se aprovecha del hecho de que el vector esté ordenado. Es posible modificarlo para que, teniendo en cuenta esa circunstancia, finalice en cuanto localice el elemento o se determine que éste no existe. A esta modificación se le conoce como búsqueda secuencial con centinela.

- Si el vector está ordenado no es necesario recorrerlo completamente.
- En cuanto el valor buscado es encontrado se puede finalizar el recorrido y en el momento en que aparece un valor mayor que el buscado también se puede detener la ejecución.

```
entero funcion busquedaSecuencialCentinela (v ∈ vector(MAX) de elemento, valor
   ∈ elemento)
variables
  i ∈ entero

  i←1

  mientras v(i)<valor y i<MAX hacer
    i←i+1
  fin mientras

  si v(i)=valor entonces
    busquedaSecuencialCentinela←i
  si no
    busquedaSecuencialCentinela←-1
  fin si
fin funcion
```

[***Pseudocódigo Búsqueda Secuencial por Centinela.*** (s.f.).]

Complejidades de tiempo

- Complejidad en el peor de los casos: **O(n).** se da cuando el elemento a buscar es el último del vector o mayor que todos los elementos del vector.
- Complejidad del mejor caso: **O(1).** se produce cuando el elemento a buscar es el primero del vector o menor que todos los elementos del vector.
- Complejidad de caso promedio: **O(n).** Por término medio, el algoritmo emplea del orden de n/2 iteraciones

Complejidad Espacial

- **O(1)**

La búsqueda secuencial con centinela no precisa recorrer el vector por completo, pero aun así resulta ineficiente con arreglos muy grandes.

1.3. Búsqueda Binaria

La búsqueda binaria es un algoritmo de búsqueda para encontrar la posición de un elemento en una matriz ordenada. En este enfoque, el elemento siempre se busca en el medio de una parte de una matriz. La búsqueda binaria solo se puede implementar en una lista ordenada de elementos. Si los elementos aún no están ordenados, debemos ordenarlos primero.

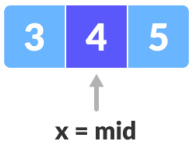
1. Se buscará x == 4

2. Se coloca bajo y alto en las posiciones más baja y alta respectivamente.

3. Encuentra el elemento *mid* a la mitad de la matriz, es decir. arr[(low + high)/2] = 6

4. Si x == mid, entonces devuelve mid. De lo contrario, compare el elemento a buscar con m.
5. Si x > mid, compare x con el elemento medio de los elementos del lado derecho de mid.
6. De lo contrario, compara x con el elemento medio de los elementos en el lado izquierdo de mid. Esto se hace configurando alto a alto = medio - 1.

7. Se repiten los pasos 3 a 6 hasta que el bajo encuentre al alto



8. x == 4

Complejidades de tiempo

- Complejidad en el peor de los casos: **O(log n).**
- Complejidad del mejor caso: **O(1).**
- Complejidad de caso promedio: **O (log n).**

Complejidad espacial

- La complejidad del espacio es **O(1)** porque se utiliza una variable adicional para el intercambio.

Se usa durante la depuración y cuando el arreglo es grande y ordenado.

2. Algoritmos de Ordenamiento

En computación, un algoritmo de ordenamiento pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación de la entrada que satisfaga la relación de orden dada. Estos ordenamientos, deberán de ser eficientes debido a que su importancia radica en optimizar el uso de otros algoritmos (como los de búsqueda) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

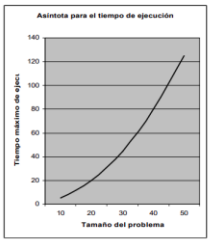
Desde los comienzos de la computación, el problema del ordenamiento ha atraído gran cantidad de investigación, tal vez debido a la complejidad de resolverlo eficientemente a pesar de su planteamiento simple y familiar. Por ejemplo, *BubbleSort* (Ordenamiento Burbuja) fue analizado desde 1956. Aunque muchos puedan considerarlo un problema resuelto, nuevos y útiles algoritmos de ordenamiento se siguen inventado hasta el día de hoy. diversos algoritmos de ordenación fueron desarrollados a lo largo de los años y siempre existió un intenso debate entre los desarrolladores sobre cuál de todos los algoritmos de ordenación era el más rápido. El debate finalizó abruptamente en 1960 cuando Sir Charles Antony Richard Hoare, nativo de Sri Lanka y ganador del premio Turing en 1980, desarrolló el algoritmo de ordenación *QuickSort* casi por casualidad mientras ideaba la forma de facilitar la búsqueda de palabras en el diccionario.

En las siguientes secciones, se explicarán a detalle los algoritmos mencionados además del método de la sacudida. Siendo esto: una descripción breve del algoritmo y su pseudocódigo. Además, se expondrán los casos acerca de sus complejidades de tiempo, siendo éstos su complejidad en el peor, mejor y caso promedio. Finalmente, se mencionarán sus aplicaciones y cuáles y cuándo son las condiciones óptimas de usar éstos.

2.1. Ordenamiento Burbuja

El método de la burbuja va comparando elementos adyacentes y empuja los valores más livianos hacia arriba (los más pesados van quedando abajo). La idea básica de este algoritmo consiste en considerar todos los intercambios posibles hasta deshacer todas las inversiones existentes en el vector

```
desde i←1 hasta MAX-1 hacer
  desde j←MAX hasta i+1 con paso -1 hacer
    si v(j)<v(j-1) entonces
      temp←v(j)
      v(j)←v(j-1)
      v(j-1)←temp
    fin si
  fin desde
fin desde
```



[***Eficiencia del Método Burbuja: Pseudocódigo y Gráfica.*** (s.f.).]

Complejidades de tiempo

- Complejidad en el peor de los casos: **O(n2).** Si queremos ordenar en orden ascendente y la matriz está en orden descendente, ocurre el peor de los casos.
- Complejidad del mejor caso: **O(n).** Si la matriz ya está ordenada, entonces no hay necesidad de ordenar.
- Complejidad de caso promedio: **O (n2).** Ocurre cuando los elementos de la matriz están desordenados (ni ascendentes ni descendentes).

Complejidad espacial

- La complejidad del espacio es **O(1)** porque se utiliza una variable adicional para el intercambio.

El problema de este algoritmo radica en que ejecuta siempre el mismo número de pasos independientemente de si el vector está ordenado o no. ***Por ello, la ordenación burbuja se usa si la complejidad no importa, el arreglo por ordenar es corto, y/o se prefiere código corto y simple.***

2.2. Método de la Sacudida

El método de la sacudida es una variación del método de la burbuja. También es conocido como Burbuja Bidireccional. En cada iteración se realizan dos etapas:

- En la primera etapa (de derecha a izquierda) se trasladan los elementos más pequeños hacia la parte izquierda del vector, almacenando en una variable la posición del último elemento intercambiado.
- En la segunda (de izquierda a derecha) se trasladan los elementos más grandes hacia la parte derecha del vector, almacenando en otra variable la posición del último elemento intercambiado.

```
subrutina sacudida (v e vector(MAX) de elemento)
variables
  i,izq,dcha,k e entero
  temp e elemento

inicio
  izq←2
  dcha←MAX
  k←MAX

  repetir
    desde i←dcha hasta izq con paso -1 hacer
      si v(i-1)>v(i) entonces
        temp←v(i-1)
        v(i-1)←v(i)
        v(i)←temp
        k←i
      fin si
    fin desde
    izq←k+1

    desde i←izq hasta dcha hacer
      si v(i-1)>v(i) entonces
        temp←v(i-1)
        v(i-1)←v(i)
        v(i)←temp
        k←i
      fin si
    fin desde
    dcha←k-1
  hasta que izq>dcha
fin
```

[***Pseudocódigo Método de la Sacudida.*** (s.f.).]

- El algoritmo termina: cuando en una etapa no se producen intercambios, o bien, cuando el contenido de la variable “izquierda” ha sobrepasado a la variable “derecha”.

Complejidades de tiempo

- Complejidad en el peor de los casos: **O(n^2).** ocurre cuando los elementos de la matriz deben ordenarse en orden inverso
- Complejidad del mejor caso: **O(n).** Si la matriz ya está ordenada, entonces no hay necesidad de ordenar.
- Complejidad de caso promedio: **O (n2).** Ocurre cuando los elementos de la matriz están desordenados (ni ascendentes ni descendentes).

Complejidad espacial

- La complejidad del espacio es **O(1)**

2.3. Quicksort

Quicksort es un algoritmo de ordenamiento basado en el enfoque *divide y vencerás* donde una matriz se divide en subarreglos seleccionando un elemento pivote.

1. Al dividir la matriz, el elemento pivote debe colocarse de tal manera que los elementos menores que el pivote se mantengan en el lado izquierdo y los elementos mayores que el pivote estén en el lado derecho del pivote.
2. Los subarreglos izquierdo y derecho también se dividen utilizando el mismo enfoque. Este proceso continúa hasta que cada subarreglo contiene un solo elemento.
3. En este punto, los elementos ya están ordenados. Finalmente, los elementos se combinan para formar una matriz ordenada.

Complejidades de tiempo

- Complejidad en el peor de los casos**: O(n2).** Ocurre cuando el elemento pivote seleccionado es el elemento más grande o el más pequeño. Esta condición conduce al caso en el que el elemento pivote se encuentra en un extremo de la matriz ordenada.
- Complejidad del mejor caso: **O(n*log n).** Ocurre cuando el elemento pivote es siempre el elemento medio o cerca del elemento medio.
- Complejidad de caso promedio**: O(n*log n).** Ocurre cuando las condiciones anteriores no se dan.

Complejidad espacial

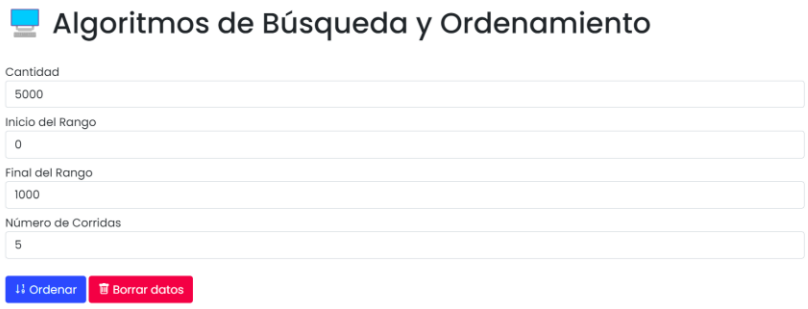
- La complejidad del espacio para Quicksort es **O(log n).**

Desde hace años, Quicksort es el método de ordenación más utilizado y en la práctica, es el algoritmo de ordenación más rápido conocido. Muchos lenguajes de programación, ya lo tienen incluido como método. ***El algoritmo Quicksort se utiliza cuando el lenguaje de programación es bueno para la recursividad, la complejidad del tiempo importa y la complejidad del espacio importa. Por lo tanto, para la mayoría de los casos, se debería utilizar este algoritmo***

3. Descripción de la interfaz de usuario

El proyecto desarrollado fue creado con HTML y JavaScript. Además, se utilizó la biblioteca FileSaver.js; la cual, se usó para generar un archivo de texto con el arreglo ordenado generado. Finalmente, se implementó el *framework* Bootstrap para agilizar el proceso del diseño del sitio.

Se le presenta al usuario la *landing page* donde se le solicita la cantidad o instancia de números a generar, el inicio y fin del rango de éstos, y finalmente, el número de corridas o ejecuciones que el programa realizará.



[Brizuela, J. (2022). Interfaz de Usuario**]**

Cuando el usuario haya picado el botón de ordenar, se generará una tabla con los tiempos de los tres algoritmos de ordenamiento vistos y sus respectivos promedios dependiendo de las corridas.

[Brizuela, J. (2022). Interfaz de Usuario**]**

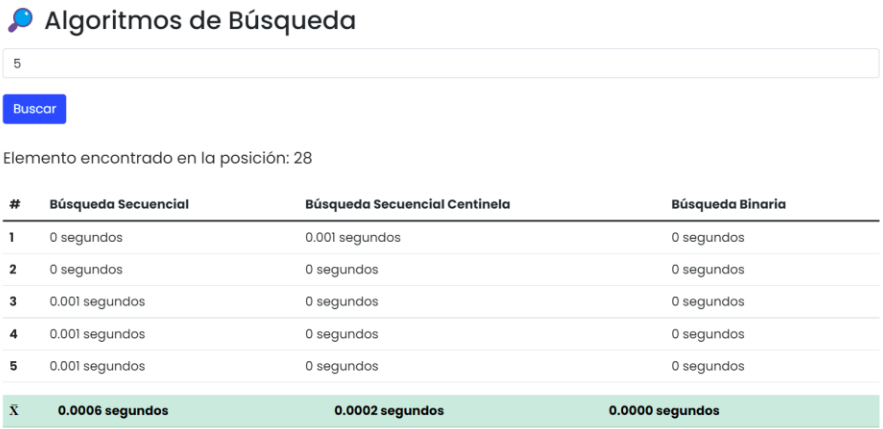
Una vez ordenado, se le presentará un indicador al usuario y se generará un archivo de texto incluyendo el arreglo.

[Brizuela, J. (2022). Interfaz de Usuario**]**

A su vez, se generará una nueva sección para los algoritmos de búsqueda. Solicitando al usuario el elemento por buscar del arreglo generado ordenado.

[Brizuela, J. (2022). Interfaz de Usuario**]**

Una vez que el usuario le haya dado el elemento por buscar, el sitio devolverá el índice del elemento si fue encontrado. De lo contrario, devolverá “elemento no encontrado”. Finalmente, se generará una tabla como en la sección de los algoritmos de ordenamiento.



[Brizuela, J. (2022). Interfaz de Usuario**]**

4. Análisis de Tiempos de Ejecución

Para los experimentos de tiempo de ejecución, se utilizó Python 3.9.7. Cada celda se llenó con el promedio de tiempo de 20 corridas.

```
for j in range(len(limites)):
    cantidad = limites[j]
    print("Tamaño del arreglo:", cantidad)
    for i in range(20):
        generadorNumerosAleatorios(cantidad, inicio, fin, myarr)
        start = time.time()
        quickSort(myarr, 0, len(myarr) - 1)
        end = time.time()
        tiempo = end - start
        print(str(i + 1) + ".", "El tiempo de ejecucción de quick sort fue de:", tiempo)
        tiempos.append(tiempo)
        myarr.clear()
    promedio = sum(tiempos) / 20
    print("Promedio de ejecucción con tamaño " + str(cantidad) + " : " + str(promedio))
    print("")
```

ALGORITMOS DE ORDENAMIENTO

Tamaño de la instancia	**Burbuja**	**Sacudida**	**QuickSort**
50.00	4.91E-05	0	0
100.00	0.00059706	0.00039951	9.97E-05
500.00	0.012167692	0.01082112	0.00084649
1,000.00	0.047870874	0.04388142	0.0009973
5,000.00	1.284585476	1.16932251	0.00698185
10,000.00	5.092366695	4.50495243	0.01595736
20,000.00	20.657938	17.7635088	0.03191447
50,000.00	132.8716598	109.33063	0.08976007
100,000.00	523.390589	435.978621	0.18749738
200,000.00	2141.157521	1751.9284	0.41691399
500,000.00	13215.25215	10919.6776	1.09405589

[Brizuela, J. (2022). Tabla de Resultados Algoritmos de Ordenamiento**]**

Como se puede apreciar en la tabla, Quicksort fue el claro ganador siendo 9,980 veces más rápido que el método de la sacudida, y 12,079 veces más rápido que el ordenamiento burbuja en la última prueba.

ALGORITMOS DE BÚSQUEDA

Tamaño de la instancia	**Secuencial**	**Sec. Centinela**	**Binaria**
50.00	0.000997305	0.00099635	0.00098586
100.00	0.000997305	0.00099707	0.00099707
500.00	0.000997305	0.0009973	0.0009973
1,000.00	0.000997305	0.00098562	0.00099611
5,000.00	0.000995636	0.00199604	0.00100923
10,000.00	0.000992775	0.00199556	0.00098991
20,000.00	0.001994848	0.00398779	0.00099683
50,000.00	0.004986763	0.00897694	0
100,000.00	0.008975506	0.01797605	0.00099945
200,000.00	0.022773266	0.03690147	0.00099635
500,000.00	0.057845831	0.08743191	0.00098372

[Brizuela, J. (2022). Tabla de Resultados Algoritmos de Búsqueda**]**

De la misma manera, los algoritmos “divide y vencerás” ganaron en la categoría de búsqueda, siendo la búsqueda binaria en la última instancia 89 veces más rápida que la búsqueda secuencial con centinela y 58 veces más rápido que la búsqueda secuencial normal.

Cabe destacar que se buscaron los últimos elementos del rango dado por el usuario. Por ejemplo: teniendo un rango de 0 a 1000, se buscó el elemento 1000.

Conclusión

Muchas actividades del día a día, requieren que diferentes colecciones de elementos utilizados se pongan en un orden específico. Un ejemplo claro es en la universidad donde los estudiantes de una clase se ordenan por sus apellidos o por los números de cuenta. El trabajo realizado pretendió dar a conocer los algoritmos de búsqueda y ordenamiento más populares, así como exponer la eficiencia que tiene uno hacia otro. Ambos algoritmos, son empleados profusamente en computación, ahí recae su gran importancia y su apoyo de uno con el otro.

Referencias

Alfaro, T. (s.f.). *Algoritmos de Búsqueda y Ordenamiento* Departamento de Informática Universidad Técnica Federico Santa María.

Binary Search (2022). Programiz. Recuperado el 20 de febrero de 2022 de: https://www.programiz.com/dsa/binary-search

Bubble Sort (2022). Programiz. Recuperado el 20 de febrero de 2022 de: https://www.programiz.com/dsa/bubble-sort

Estructuras de Datos en Java: Algoritmos de ordenación y búsqueda. **(s.f.).** Recuperado de: https://sites.google.com/a/espe.edu.ec/programacion-ii/home/a1arreglos/algoritmos-de-ordenacion-y-busqueda

Linear Search (2022). Programiz. Recuperado el 20 de febrero de 2022 de: https://www.programiz.com/dsa/linear-search

Quicksort (2022). Programiz. Recuperado el 20 de febrero de 2022 de: https://www.programiz.com/dsa/quick-sort