Programación

Unidad 8: Algoritmos de Ordenamiento y Búsqueda

“Dos peligros amenazan constantemente al mundo: orden y desorden.”

~ Paul Valéry

Búsqueda

A menudo nos veremos en la necesidad de encontrar un valor en un arreglo. Esto puede ser utilizado tanto para saber si algún valor ya está presente o recuperar algún registro para trabajar o imprimir sus datos.

Veremos 3 métodos de búsqueda distintos, cada uno tiene sus particularidades y en algunos casos algunas precondiciones que debe cumplir el arreglo para poder usarse.

Búsqueda Secuencial

Este algoritmo de búsqueda es el más sencillo de comprender, pero al mismo tiempo es el menos eficiente. Se reduce simplemente a recorrer el vector hasta encontrar el valor buscado.

Algoritmo

Como dijimos, recorremos el vector y por cada elemento evaluamos si es igual al valor buscado. Cabe destacar que lo que se devuelve no será el valor (que ya lo teníamos) sino la posición. En caso que no se encuentre el valor en el vector, devolveremos -1 :

```
1   int buscarSecuencial(int vec[], int, cant, int valor) {
2       for(int i = 0; i < cant; i++) {
3           if (vec[i] == valor) {
4               return i;
5           }
6       }
7
8       //Si no salio todavia, es que no estaba en el vector
9       return -1;
    }
```

Si bien es un algoritmo muy sencillo, como dijimos anteriormente es el más costoso en términos de eficiencia, ya que si el valor que buscamos está muy cerca del final, habremos recorrido casi la totalidad de los elementos.

Búsqueda por Posición Única Predecible (PUP)

Existen algunos casos donde podremos hacer una búsqueda directa apoyándonos en alguna clave que pueda ser equivalente a la posición de un elemento dentro del vector.

Supongamos que tenemos los registros de venta diaria de un mes, formados por los siguientes datos:

- Día
- Venta Diaria

Como nosotros sabemos que cantidad de días hay como máximo (31) y los días son consecutivos, entonces podemos hacer una equivalencia entre el número del día en el mes y la posición que ocupará la venta en el vector. Por lo tanto podemos utilizar un vector de **float** donde cada posición guardará la venta del día que le corresponde:

```
1   int main() {
2       float ventas[31];
3
4       for(int i = 0; i < 31; i++) {
5           cout << “Ingrese la venta del día “ << i + 1 << “: “;
6           cin >> ventas[i];
7       }
8   }
```

En este ejemplo, le pediremos al usuario que ingrese la venta de un día determinado, y si se mira con atención veremos que la equivalencia que estamos encontrando es:

posición = **día - 1**

Es decir la venta del día 4 estará en el vector en la posición 3, la del día 1 en la posición 0 y así sucesivamente.

Ahora supongamos que en realidad el usuario no tiene las ventas ya sumarizadas por día, sino que tiene un registro de venta por cada venta realizada:

- Día
- Valor venta

Si queremos poder realizar la carga de los datos para luego imprimir las ventas diarias y determinar cuál es el día de mayor venta, haremos uso y encontraremos el mayor beneficio del vector con PUP:

```
1   int main() {
2       float ventas[31];
3       int dia;
4       float vta;
5       cout << “Ingrese las ventas del mes” << endl;
6       for(int i = 0; i < 31; i++) {
7           cout << “Día: “;
8           cin >> dia;
9           cout << “Venta: ”;
10          cin >> vta;
11          ventas[dia - 1] = ventas[dia - 1] + vta; // Acumulo la venta leida
12      }
13  }
```

Vemos que durante la carga utilizaremos el día como clave para ubicarnos en el vector en la posición que le corresponda a ese día.

De esta forma tendremos el vector cargado con los acumulados de venta diario, sin necesidad de haber buscado por cada venta la posición en el vector de forma secuencial.

Como vemos existen algunas precondiciones para poder hacer uso de la búsqueda por PUP:

- Equivalencia entre alguna clave del dato y la posición: debe haber alguna forma de convertir una clave en una posición. La fórmula para convertir las claves a posiciones es siempre la misma:

  posicion = clave - valor_inicio_rango

- Se debe definir un rango conocido de claves: Los valores posibles de las claves deben ser conocidos de antemano
- El rango debe estar completo: para que no haya agujeros en el vector el rango de las claves debe ser completo, deben estar todos los valores.

Ejemplo: además del día (que vemos que cumple con las tres condiciones) pensemos algunos ejemplos más:

- Horas del día: Sabemos que son números del 0 al 23
- Meses del año: Rango del 1 al 12
- Legajos: Si sabemos que para alguna carga se cargan los datos de alumnos con legajos consecutivos del 50 al 150.

En el caso de que el rango no esté completo, igualmente podremos usar PUP pero siempre teniendo en cuenta que no podemos asegurar que los valores sean relevantes en todas las posiciones.

Búsqueda binaria

Si tuviéramos un vector ordenado, en el que no podemos establecer PUP, no tendríamos otra forma de encontrar cualquier elemento que haciendo una búsqueda secuencial. Pero si lo analizamos un poco, el hecho de que el vector ya esté ordenado debería ayudarnos a reducir el tiempo de búsqueda porque sabemos que cuanto más grande, más cerca del final estará (siempre que esté ordenado en forma ascendente). Para esto utilizaremos otro algoritmo que es la búsqueda binaria.

La búsqueda binaria es un algoritmo de búsqueda rápida con una complejidad de tiempo de ejecución de (log n). Este algoritmo de búsqueda funciona sobre el principio de dividir y conquistar. Para que este algoritmo funcione correctamente, el vector debe estar ordenado.

La búsqueda binaria busca un elemento en particular comparando el elemento más central del array. Si se produce una coincidencia, se devuelve el índice del elemento. **Si el elemento intermedio es mayor** que el elemento buscado, entonces el elemento se busca en el **sub-vector a la izquierda** del elemento central. **De lo contrario, el elemento se busca en el sub-vector a la derecha** del elemento central. Este proceso también continúa en el sub-vector hasta que el tamaño del sub-vector se reduce a cero.

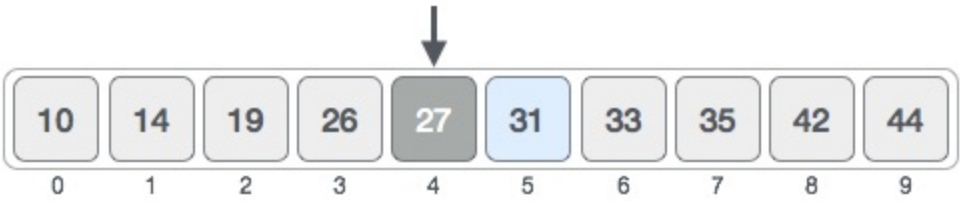
¿Cómo funciona la búsqueda binaria?

Para que una búsqueda binaria funcione, es obligatorio que el vector de datos se encuentre ordenado. Veamos un ejemplo gráfico. El siguiente es nuestro vector ordenado y supongamos que necesitamos buscar la ubicación del valor **31** utilizando la búsqueda binaria.

Primero, determinaremos la mitad del vector usando esta fórmula:

```
medio = inicio + (fin - inicio) / 2
```

Aquí resulta, 0 + (9 - 0) / 2 = 4 (valor entero de 4.5). Entonces, **4** es la mitad del vector.



Ahora **comparamos el valor almacenado en la posición 4 con el valor que se está buscando**, es decir, 31. Encontramos que el valor en la ubicación 4 es 27, que no es el buscado. **Como el valor que buscamos es mayor que 27 y tenemos un vector ordenado, ya sabemos que el valor objetivo debe estar en la parte superior del vector.** Es decir, podemos desestimar todo lo que está a la izquierda de la posición 4 que acabamos de evaluar.

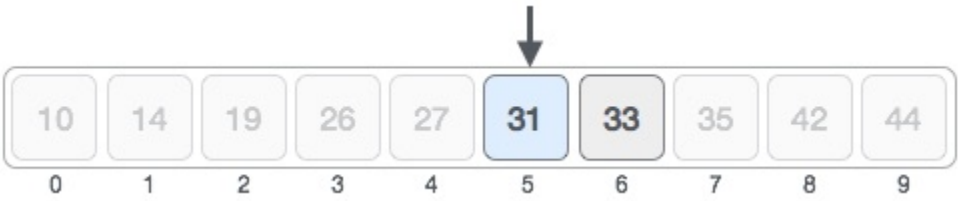
Cambiamos nuestro valor de **inicio** a **medio + 1** y volvemos a encontrar el nuevo valor medio.

```
inicio = medio + 1
medio = inicio + (fin - inicio) / 2
```

Nuestro nuevo medio ahora es 7. Comparamos el valor almacenado en la ubicación 7 con nuestro valor objetivo 31.

El valor almacenado en la ubicación 7 no coincide, es más que lo que estamos buscando. Por lo tanto, el valor debe estar en la parte inferior de esta ubicación.

Por lo tanto, calculamos el medio de nuevo. Esta vez son las 5.



Comparamos el valor almacenado en la ubicación 5 con nuestro valor objetivo. Encontramos que es un partido.

Concluimos que el valor objetivo 31 se almacena en la ubicación 5.

La clave está en que la búsqueda binaria divide a la mitad los elementos de búsqueda y, por lo tanto, reduce el recuento de comparaciones que se realizarán a números muy inferiores comparado con la búsqueda secuencial.

Veamos el algoritmo en C/C++

```
1   int busquedaBinaria(int arr[], int n, int x) {
2       // Cuando arranco evalúo todo el vector de 0 a n - 1
3       int inicio = 0;
4       int fin = n -1;
5
6       while (fin >= inicio) {
7           int mitad = inicio + (fin - inicio) / 2;
8
9           // Si el elemento es el del medio, devolvemos la posicion
10          if (arr[mitad] == x)
11              return mitad ;
12
13          // Si el elemento es menor entonces solo puede estar en la primer mitad
14          if (arr[mitad] > x) {
15              fin = mitad - 1; //Cambio el limite superior
16          } else {
17              inicio = mitad + 1;  // Cambio el limite inferior
18          }
19      }
20
21      // Si llegamos hasta aca es que el elemento no estaba
22      return -1;
23  }
```

Algoritmos de Ordenamiento

Como hemos visto en el caso de la búsqueda binaria, tener el vector ordenado nos permite realizar algunos algoritmos que de otra forma serían mucho más trabajosos. Además de que puede ser un requerimiento emitir algún listado ordenado.

Para poder ordenar un vector existen infinidad de algoritmos, los más eficientes intentarán realizar toda la operatoria sobre el mismo vector, dado que la memoria asignada de forma

estática queda asignada hasta que el programa finalice, y no es eficiente tener duplicado el vector simplemente como estructura auxiliar para el ordenamiento.

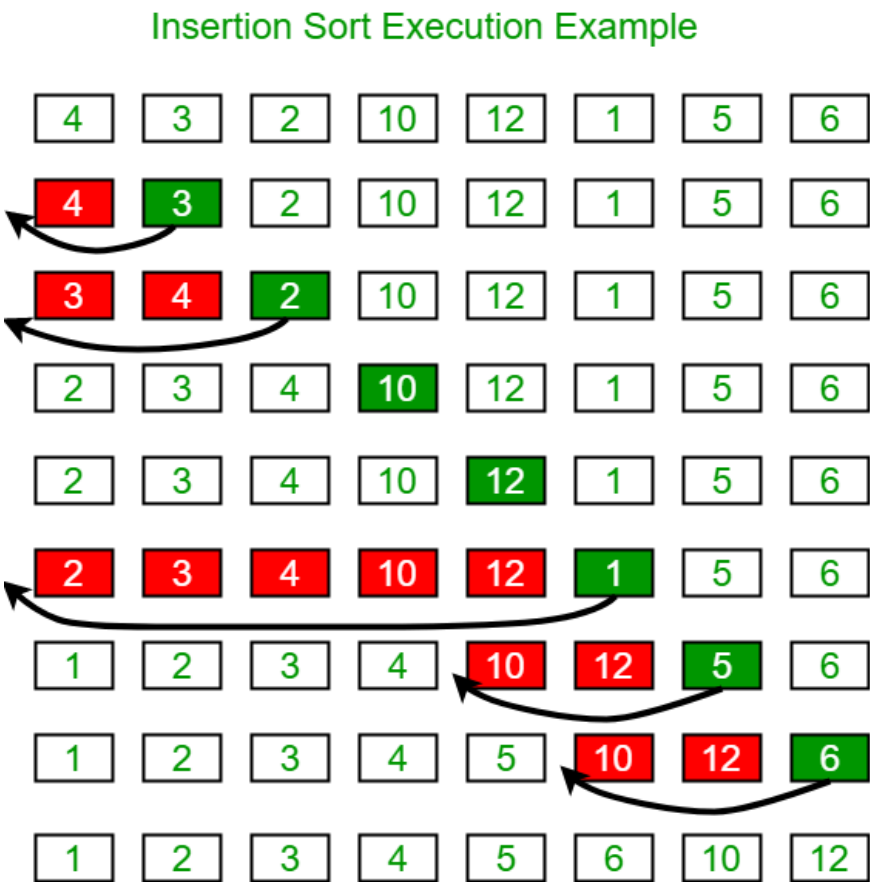
Nosotros veremos solamente dos algoritmos, siendo el más eficiente Burbujeo Mejorado.

Insertion Sort

Este algoritmo es uno de los más sencillos (y por lo tanto no tan eficiente!) consiste en ordenar los elementos similar a como hacemos con los naipes, vamos insertando el valor en donde le corresponde mirando los naipes que tengo hasta el momento.

Veamos un ejemplo gráfico:

Como vemos lo que hace es ubicar cada elemento en donde le corresponde en el subvector a la izquierda del elemento.



Veamos otro ejemplo paso a paso:

12, 11, 13, 5, 6

Iteramos para i = 1 (arrancamos desde el 2do elemento) hasta 4 (último elemento del array)

i = 1. Como 11 es más chico que 12, mueve 12 e inserta el 11 en su lugar

11, 12, 13, 5, 6

i = 2. 13 queda en su posición porque todos los elementos de A[0..I-1] son más chicos

11, 12, 13, 5, 6

i = 3. El 5 se mueve al principio de todo y todos los otros elementos del 11 al 13 se desplazan un lugar a la derecha.

5, 11, 12, 13, 6

i = 4. 6 se moverá a la posición después del 5 y los elementos del 11 al 13 se mueven un lugar.

5, 6, 11, 12, 13

Cuidado, si bien pareciera que solo necesitamos recorrer el vector una vez, en realidad cada vez que tuvimos que desplazar elementos, tuvimos que recorrer un subvector y esto varias veces, por eso decimos que no es el más eficiente de los algoritmos.

Veamos el código

```
1   void insertionSort(int arr[], int n) {
2      int i, key, j;
3      for (i = 1; i < n; i++)
4      {
5          key = arr[i];
6          j = i - 1;
7
8          /* Movemos los elementos de arr[0..i-1], que son
9          mas grandes que key, a la posicion siguiente
10         a su posicion actual */
11         while (j >= 0 && arr[j] > key)
12         {
13             arr[j + 1] = arr[j];
14             j = j - 1;
15         }
16         arr[j + 1] = key;
17     }
18 }
```

Burbujeo

Este método de ordenamiento consiste en comparar cada par de valores adyacentes, e intercambiarlos en caso que no estén en el orden buscado

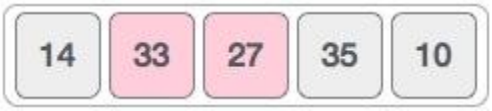
¿Cómo funciona?

Tomemos un vector desordenado para nuestro ejemplo. El método de burbujeo precisará O(n2) repeticiones así que lo mantendremos lo más sencillo posible

Burbujeo comienza con los primeros dos elementos, comparándolos y verificando cuál es más grande.

En este caso el valor 33 es más grande que 14, por lo cual ya está ordenado. Luego nos movemos un lugar y compararemos el 33 con el 27.

Y descubrimos que el 27 es más chico que el 33, por lo cual estos dos valores deben ser intercambiados.



El nuevo vector debería quedar de la siguiente forma:

Luego nos volvemos a mover un lugar y comparamos el 33 con el 35. Que al ya estar ordenados, quedan donde están.

Luego evaluamos el 35 con el 10

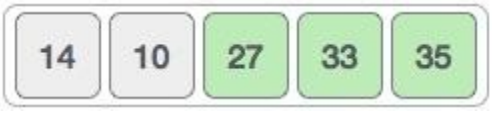
Como el 10 es más chico necesitamos hacer el intercambio ya que no están en orden:

Así llegamos al final del vector, al final de esta primer iteración el vector debería lucir así

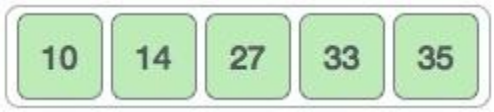
Como decíamos, vemos que al final de la primer iteración, el valor 35 ya encontró su lugar definitivo en el vector ordenado. Es decir en la primer iteración ya logramos tener un elemento ordenado. Por lo tanto ahora al hacer la segunda pasada, no necesitaremos llegar hasta ese valor para evaluar, sino que deberemos evaluar los valores hasta N - 2

Veamos como va quedando el vector al final de cada iteración en las siguientes pasadas. Luego de la segunda iteración

Vemos que ahora ya son dos elementos los ordenados, el 33 y el 35 ya están en donde quedarán finalmente. En la tercer iteración quedarán el 27, 33, 35 ya ordenados:



Luego el vector quedará finalmente ordenado:



Veamos el código para realizar el algoritmo en su versión más sencilla (y menos eficiente)

```
1   void burbujeo(int arr[], int n)
2   {
3      int i, j, aux;
4      for (i = 0; i < n-1; i++) {
5
6          // Los ultimos i elementos ya estan ordenados
7          for (j = 0; j < n-i-1; j++) {
8              if (arr[j] > arr[j+1]) {
9                  aux = arr[j];
10                 arr[j] = arr[j+1];
11                 arr[j+1] = aux;
12             }
13          }
14      }
15  }
```

El inconveniente de este algoritmo es que si a la segunda pasada, el vector ya está completamente ordenado, nosotros **igualmente hacemos n- 1 iteraciones**. Para evitar esto lo que haremos es agregar una bandera o flag, que nos permita saber si pudimos completar una pasada completa sin hacer intercambios de valores**, es decir si el vector ya está ordenado**.

Burbujeo Mejorado

Como dijimos, mejoramos el burbujeo agregando el flag ordenado, al iniciar cada pasada lo ponemos en **true** si en esa pasada hubo al menos un valor que no estaba en orden, se cambia a **false.**

```
1   void burbuejoMejor(int arr[], int n)
2   {
3      int i, j, aux;
4      int i = 0;
5      bool ordenado = false;
6      while (i < n && !ordenado) {
7         ordenado = true;  // Arranco asumiendo que si esta ordenado
8         // Los ultimos i elementos ya estan ordenados
9         for (j = 0; j < n-i-1; j++) {
10            if (arr[j] > arr[j+1]) {
                  aux = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = aux;
                  ordenado = false; // Cambio el flag si hice un swap
              }
          }
          i++;
      }
  }
```

Bibliografía utilizada

Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Programación en C. Segunda Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA. S.A.U., 2005. ISBN-84: 481-9844-1.

● Brian W. Kernighan, Rob Pike. La práctica de la programación. Pearson Educación. Méjico (2000)

● Debugging with CodeBlocks: http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks